UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LIN YU HAN

ESTUDO COMPARATIVO DA PROGRAMAÇÃO PARALELA DE ALGORITMOS EM CPU COM *GO* E GPU COM *CUDA*

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2014

LIN YU HAN

ESTUDO COMPARATIVO DA PROGRAMAÇÃO PARALELA DE ALGORITMOS EM CPU COM *GO* E GPU COM *CUDA*

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. André Koscianski

PONTA GROSSA

2014

TERMO DE APROVAÇÃO

ESTUDO COMPARATIVO DA PROGRAMAÇÃO PARALELA DE ALGORITMOS EM CPU COM GO E GPU COM CUDA

por

LIN YU HAN

Este(a) Trabalho de Conclusão de Curso foi apresentado(a) em 11 de junho de 2014 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O(a) candidato(a) foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
André Koscianski
Prof.(a) Orientador(a)

_____
Gleifer Vaz Alves
Membro titular

_____
Ionildo José Sanches
Membro titular

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

Em dedicação à família, aos amigos e a todos que me apoiaram e acreditaram em mim desde sempre.

AGRADECIMENTOS

Os parágrafos a seguir expressam de maneira sincera o agradecimento a todas as pessoas que colaboraram com essa jornada da minha vida.

Agradeço primeiramente à minha mãe Shih Mei Ying, pelo esforço e paciência de todos esses anos cuidando da minha pessoa com muito amor e carinho.

Ao meu pai Lin Jung Tang, que mesmo pelas dificuldades físicas, nunca deixou faltar nada na vida em nossa família, sempre batalhando com muito amor e coragem. Merece minha admiração total, sempre me guiando com sua sabedoria, é uma honra ser filho de uma pessoa com tamanha força de vontade.

Este trabalho não poderia ser concluído sem o patrocínio indireto vindo dos dois, pai e mãe. Muito Obrigado.

Agradeço ao Prof. Dr. André Koscianski que proporcionou o patrocínio através da orientação repleta de inspiração e conhecimento.

Obrigado também à todos os amigos que compartilharam suas experiências e pelos os que incentivaram e se preocuparam com essa minha jornada.

Agradeço por último a minha pessoa, pela força de vontade, determinação e pela competência que mantive durante toda a pesquisa.

Desejo a todas as pessoas agradecidas aqui muito sucesso.

No! Try not! Do... or do not. There is no try.
(Yoda, 1980)

RESUMO

LIN, Yu Han. **Estudo Comparativo da Programação Paralela de Algoritmos em CPU com *Go* e GPU com *CUDA***. 2014. 94 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2014.

Processadores *multicore,* em outras palavras, com mais de um núcleo, se tornaram de uso comum. Entretanto o estudo da programação paralela não é muito explorado. Existem diversos problemas que podem ser solucionados de maneira mais eficiente explorando concorrência e paralelismo. Este trabalho apresenta um estudo da programação paralela nas arquiteturas *multicore*. Duas abordagens diferentes são analisadas e comparadas. O uso da nova linguagem de programação desenvolvida pela Google, a *Go*, que traz uma abordagem para a programação paralela em CPUs *multicore*, e a plataforma de programação paralela *CUDA* que utiliza as placas gráficas das GPUs da NVIDIA para o processamento paralelo. Dentre os problemas computacionais que se beneficiam do paralelismo, têm-se os algoritmos genéticos.

Palavras-chave: Paralelismo. *Go*. *CUDA*. *Multicore*. Concorrência.

ABSTRACT

LIN, Yu Han. **Comparative Study of Parallel Programming of Algorithms on CPU with *Go* and GPU with *CUDA***. 2014. 94 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Federal Technology University - Parana. Ponta Grossa, 2014.

Multicore processors, in other words, with more than one core, have become common usage. However, the study of parallel programming is not much explored. There are many problems that can be solved more efficiently exploiting parallelism and concurrency. This paper presents a study of parallel programming on multicore architectures. Two different approaches are analyzed and compared. The use of new programming language developed by Google, Go, which brings an approach to parallel programming on multicore CPUs, and the CUDA parallel programming plataform which uses the graphics cards from NVIDIA GPUs for parallel processing. Among the computational problems that benefit from parallelism, there are also genetic algorithms.

Keywords: Parallelism. Go. CUDA. Multicore. Concurrency.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API	*Application Programming Interface*
CPU	*Central Processing Unit*
CUDA	*Computer Unified Device Architecture*
GPGPU	*General Purpose Graphics Processing Unit*
GPU	*Graphics Processing Unit*
LWP	*Light-weight process*
MPI	*Message Passing Interface*

SUMÁRIO

1 INTRODUÇÃO

Sistemas computacionais complexos normalmente podem ser divididos em partes distintas que podem ser processadas simultaneamente, e consequentemente podem ter diversas instruções em execução paralela. Além disso, as técnicas de paralelismo podem ser utilizadas em muitos problemas computacionais.

A evolução do hardware tem sido rápida comparada ao desenvolvimento de software (PIKE, 2012). Com o objetivo de melhorar o desempenho dos sistemas de modo geral, surgiram as arquiteturas *multicore*, que unem em um mesmo *chip* múltiplas unidades de processamento para a realização das tarefas computacionais. Uma forma eficiente de explorar essas arquiteturas de múltiplos núcleos é o desenvolvimento de aplicações com um grau de paralelismo em sua modelagem e utilizar recursos que permitam o desenvolvimento de programas com vários fluxos de execução concorrentes (*threads*) (CAMARGO, 2012).

Entretanto, deve-se avaliar se é interessante aplicar o paralelismo em determinadas situações. Por exemplo, não há necessidade de criar um editor de texto, ou um simples *player* de música que utilize os oito núcleos de um processador *octacore*. De fato, o processamento paralelo tem se tornado mais disponível, porém a programação específica sobre essas plataformas e a identificação de problemas que se beneficiam de paralelismo ainda não é muito difundida.

As linguagens de programação atuais não têm contribuído para a área de paralelismo. Baseada nesse fator, a *Google* lançou sua própria linguagem de programação, a *Go*. Desenvolvida para ajudar a resolver problemas que a *Google* enfrenta (PIKE, 2012), *Go* proporciona recursos para a programação concorrente e paralela; propõe uma abordagem para a construção de sistemas de software em máquinas de múltiplos núcleos.

Outro grande avanço na área da programação paralela foi a ferramenta desenvolvida pela *NVDIA*, a *CUDA* (*Computer Unified Device Architecture*), que diferentemente dos outros recursos como *Cilk* e *OpenMP*, utiliza o processamento das GPUs (*Graphics Processing Units*) que até então era utilizada apenas em tarefas gráficas em três dimensões, para realizar grandes processos aritméticos. *CUDA* é uma arquiteutra de computação paralela e um modelo de programação que

aumenta o desempenho computacional ao fazer o uso da unidade de processamento gráfico (NVIDIA, 2013).

O projeto tem como principal objetivo o estudo da programação paralela com a linguagem *Go* em CPU (*Central Processing Unit*) e da plataforma *CUDA* em GPUs, trazendo uma visão geral e comparativa dessas duas ferramentas. Para isso serão escolhidos problemas computacionais que possam se beneficiar de processamento paralelo, entre algumas possibilidades, têm-se multiplicação de matrizes, cálculo da sequência de Fibonacci e os algoritmos genéticos. O objetivo do trabalho não inclui comparações de desempenho com relação ao tempo de execução ou tempo total de processamento.

1.1 JUSTIFICATIVA

Há diversas situações em ciência que tratam de otimização e busca em grande escala. Simulações meteorológicas, por exemplo, são executadas sobre cálculos numéricos para ocorrência de variáveis climáticas de longos períodos e em extensas regiões geográficas, acarretando um processamento computacional de tempo bastante elevado. Quando se depara com problemas desse tipo, é necessário verificar se as atividades realizadas no algoritmo podem ser executadas de maneira concorrente e mapeadas sobre os recursos de processamento disponíveis, de forma a melhorar algum índice de desempenho, como por exemplo, reduzir o tempo de processamento total.

Com suporte a construções concorrentes sobre as unidades de processamento central, a linguagem *Go* foi projetada para programação básica (*e.g.* sistemas operacionais) com objetivo de ter as características desejáveis em uma linguagem atual (*e.g.* orientação a objeto, paralelização e concorrência).

Outra linha de investigação surgiu com as placas de vídeo. Muitos pesquisadores começaram a usar as GPUs para a realização de cálculos; percebendo isso, a indústria lançou a plataforma *CUDA*, com compilador específico, beneficiando problemas numéricos com o paralelismo. A programação paralela envolve conhecimento de arquitetura paralela (*e.g.* acesso à memória, composição/distribuição das unidades de processamento) e de sincronização de processos (*e.g.* tabela de processos, mapas de memória e multiprocessamento).

1.2 PROBLEMA

Programação concorrente e paralela são tópicos não muito tradicionais entre programadores. Existem diversas plataformas de desenvolvimento diferentes, englobando bibliotecas e hardware, com características particulares.

O problema abordado neste trabalho consiste em implementar problemas paralelizáveis, utilizando duas tecnologias distintas, trazendo uma visão geral dessas tecnologias e visando estabelecer um quadro comparativo entre elas.

Essa comparação pretende apresentar principalmente a perspectiva de projeto de programação de uma solução, ou seja, as dificuldades relacionadas a utilizar cada alternativa para resolver o problema proposto.

1.3 OBJETIVO

Nessa seção serão abordados o objetivo geral e os objetivos específicos do trabalho.

1.3.1 Objetivo Geral

Este trabalho visa uma análise de duas ferramentas de programação paralela por meio de experimentos, visando elaborar um quadro geral de comparação entre elas. Esta análise é direcionada a comparação de aspectos de implementação, como dificuldade de programar em cada plataforma.

1.3.2 Objetivos Específicos

Os objetivos específicos estão listados abaixo, apresentando os principais focos do estudo.

- Estudo da linguagem *Go*.
- Estudo da plataforma *CUDA*.

- Implementação de programas para familiarização com a linguagem *Go* e com a plataforma *CUDA*.
- Realizar uma comparação de dificuldades e vantagens de implementação dos objetos de estudo.

2 REFERENCIAL TEÓRICO

Nesta seção serão abordados conceitos em relação ao que está sendo estudado, cujo objetivo é apresentar definições e características das ferramentas que serão usadas (*Go* e *CUDA*) e um embasamento teórico de algoritmos que se beneficiam de paralelismo. Conceitos que envolvem indiretamente com programação também serão abordados, como uma visão geral sobre paralelismo, processos e *threads*.

2.1 PROCESSOS

Um processo pode ser definido como uma entidade de um programa que possui um contador que especifica a próxima instrução a ser executada (SILBERSCHATZ *et al*, 2004). Em geral, pode ser considerado um programa em execução, ou seja, um programa que está sendo executado em um dos processadores virtuais do sistema operacional (TANENBAUM e STEEN, 2007). Entretanto, um programa de computador pode ser composto por uma série de processos, sendo cada processo equivalente a uma unidade de trabalho.

A capacidade de um sistema operacional executar simultaneamente dois ou mais processos é chamado de multiprocessamento. No computador são executados vários processos de maneira concorrente, o sistema operacional deve monitorar essas atividades, assim como o gerenciamento de memória (TANENBAUM e STEEN, 2007).

Os processos quando executados, passam por uma série de fases, cada fase é chamado de estado. Um processo executado pode estar nos seguintes estados (SILBERSCHATZ *et al*, 2004):

- Novo: estado em que o processo é criado;
- Em execução: execução das instruções;
- Em espera: espera por algum evento (I/O ou recepção de um sinal);
- Pronto: o processo está pronto para ser executado por um processador;
- Terminado: o processo é encerrado.

Com uma arquitetura que proporciona o multiprocessamento, o desenvolvedor possui a capacidade de implementar multiprogramação. A

multiprogramação tem como objetivo garantir a execução constante de processos (processos concorrentes) de forma a maximizar a utilização da CPU (SILBERSCHATZ *et al*, 2004).

Existem duas modalidades de processos concorrentes: aqueles que não afetam a execução de outros processos, ou seja, que podem não compartilhar dados com outros processos são chamados de processos independentes; e aqueles onde a execução afeta os outros processos e que por ventura possa a vir compartilhar dados com os outros processos, são chamados de processos cooperativos (SILBERSCHATZ *et al*, 2004).

Um ambiente que proporciona processos cooperativos possui muitas vantagens (SILBERSCHATZ *et al*, 2004). Essas características vão ser identificadas durante o desenvolvimento do trabalho, pois o problema estudado envolve o compartilhamento de informações, processamento mais rápido com a divisão de tarefas em subtarefas (ganho de velocidade proveniente da arquitetura *multicore*) e modularidade. A modularidade refere-se à divisão das funções do programa em processos separados, ou em *threads*.

Os processos cooperativos compartilham dados, para isso é necessário que haja comunicação entre eles. Um exemplo clássico é o problema do produtor-consumidor (SILBERCHATZ *et al*, 2004). O produtor gera a informação e o consumidor usa essas informações geradas.

O programa trabalha com um *buffer* para a comunicação de processos. Com um *buffer* ilimitado, o produtor pode sempre produzir novas tarefas e armazená-las no *buffer*, e o consumidor pode ter que esperar por novas tarefas a serem consumidas; com um *buffer* limitado, o consumidor deve esperar encher o *buffer* para começar a consumir, e o produtor deve esperar que o *buffer* esvazie para inserir dados novamente (SILBERCHATZ *et al*, 2004).

Uma forma de implementar o problema do produtor-consumidor é usando as primitivas *send* e *receive* para a troca de mensagens. Essa troca pode ser síncrona ou assíncrona. Em um *send* síncrono o processo que envia a mensagem fica bloqueado até que o processo receptor receba a mensagem, já o *send* assíncrono envia a mensagem e já retoma com suas atividades sem esperar pelo receptor. Em um *receive* síncrono o processo receptor fica bloqueado até que a mensagem fique disponível, e o *receive* assíncrono trabalha tanto com uma mensagem válida quanto uma mensagem nula (SILBERCHATZ *et al*, 2004).

A linguagem *Go* trabalha com os conceitos de *buffers* limitados/ilimitados e de comunicação *send/receive* em um de seus mecanismos de comunicação e sincronização para execuções concorrentes, os canais (*channels*).

Baseado no fato de que um processo é uma unidade de trabalho em um sistema, pode-se implicitamente descrever que cada processo em um determinado momento executa uma de suas atividades para realizar uma ação específica. Essas ações são chamadas de *threads*, usadas também para o controle de execução dos processos (SILBERCHATZ *et al,* 2004).

2.2 *THREADS*

Um processo leve (LWP – *light-weight process*) ou simplesmente *thread*, é uma unidade de utilização da CPU composta por um ID que compartilha com outras *threads* do mesmo processador os recursos deste, como seção de código (SILBERCHATZ *et al*, 2004).

Quando se lida com múltiplas *threads*, o desenvolvedor da aplicação deve gerenciar de forma adequada o acesso de dados para não ocorrer *deadlocks*. Essas situações acontecem quando uma *thread* permanece bloqueada no aguardo de um sinal de outra *thread* (que nunca é enviado), ou seja, falha na comunicação e sincronização dos processos leves.

2.3 PROGRAMAÇÃO PARALELA

Conforme apresentado nas seções anteriores, processos concorrentes cooperativos são aqueles em que a execução influencia os outros processos.

O objetivo de usar a paralelização é aumentar o desempenho da aplicação com a execução nos vários núcleos disponíveis na máquina.

Existem duas abordagens de paralelismo: a autoparalelização e programação paralela. A autoparalelização ocorre quando uma aplicação é desenvolvida sequencialmente não modelada para ser processada paralelamente, porém o compilador tenta automaticamente paralelizar essa aplicação para que possa se aproveitar da estrutura do hardware (*e.g. multicore*) (YANO, 2010).

Por outro lado, a programação paralela desenvolve o paralelismo desde o algoritmo, logo, esta abordagem fornece maior ganho de desempenho, porém exige um maior esforço no desenvolvimento (YANO, 2010).

2.4 *GO*

Go é uma linguagem de programação criada pela *Google* e tornou-se um projeto *open source* em 2009 (GO AUTHORS, 2010).

> A linguagem de programação *Go* é um projeto *open source* para fazer programadores mais produtivos. *Go* é expressivo, conciso, limpo e eficiente. Seus mecanismos de concorrência o torna mais fácil de escrever programas que tira o máximo proveito de máquinas *multicores* e de rede, enquanto seu novo tipo de sistema permite a construção de um programa flexível e modular. *Go* compila rapidamente o código para a máquina e tem a conveniência da coleta de lixo e do poder da reflexão em tempo de execução. É rápida, de tipagem estática, linguagem compilada que se comporta como uma tipagem dinâmica, linguagem interpretada (GO AUTHORS, 2013, p. 1).

Uma das características da linguagem é o suporte para a programação concorrente. Quando o problema é de natureza paralela, podem-se usar métodos de concorrência para manipular as tarefas executadas de modo que atinjam um grau de paralelismo (GO AUTHORS, 2013).

Para a programação concorrente e paralela, o *Go* utiliza as *Goroutines* e *Channels*.

2.4.1 *Goroutines*

Goroutines são semelhantes a *threads*. São funções que executam concorrentemente com outras *goroutines* em um mesmo espaço de endereço (GO AUTHORS, 2012). Qualquer função em *Go* pode ser invocada como uma rotina normal ou como uma *goroutine* concorrente através da palavra chave *go* na frente da rotina para a execução concorrente.

A Figura 1 mostra duas chamadas para a função de impressão de texto, uma com a palavra chave ‘*go’* (linha 2) e a outra sem.

```
1  func main() {
2      go fmt.Println("Hello de uma goroutine")
3      fmt.Println("Hello da goroutine principal")
4  }
```

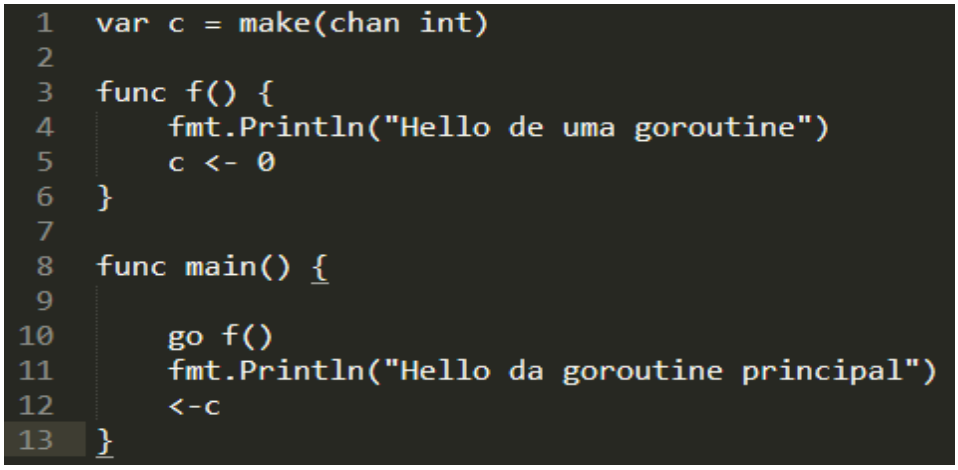
Figura 1 – Código exemplo escrito em *Go* com *goroutine*
Fonte: Autoria própria

A chamada usando '*go*' será executada concorrentemente. Uma *goroutine* é mais leve do que uma *thread* (DOXSEY, 2013), o que significa que ela consome menos recursos da máquina. Um programa escrito em Go roda em uma *goroutine* singular, porém essa *goroutine* pode criar outras *goroutines*, executadas concorrentemente.

A execução do programa mostrado na Figura 1, apresenta como uma *goroutine* funciona. O programa principal imprime na tela o texto da linha 3 e termina, não dando tempo da *goroutine* da linha 2 ser escalonada, ou seja, a *goroutine* criada executa juntamente com a *goroutine* principal, porém esta termina a execução do programa antes da *goroutine* terminar sua tarefa. Esse problema pode ser resolvido usando canais.

2.4.2 *Channels*

O *channel* (canal) é o principal mecanismo da linguagem *Go* para a execução sincronizada de *goroutines* (GO AUTHORS, 2012). São condutos que permitem enviar e receber valores entre *goroutines*, de maneira sincronizada (funciona como um sinalizador de recepção e transmissão de valores). *Channels* são alocados através do prefixo *make* e também possuem um tipo (inteiro, flutuante, etc.) como uma variável normal.



```
1  var c = make(chan int)
2
3  func f() {
4      fmt.Println("Hello de uma goroutine")
5      c <- 0
6  }
7
8  func main() {
9
10     go f()
11     fmt.Println("Hello da goroutine principal")
12     <-c
13 }
```

Figura 2 – Exemplo de sincronização channel
Fonte: Autoria própria

A Figura 2 mostra um código com *goroutine* e canal. Essa implementação resulta na impressão da frase "*Hello da goroutine principal*" e logo na sequência a frase "*Hello de uma goroutine*":

linha 1: Declaração do *channel* (c) do tipo inteiro.

linha 3: Declaração da função *f()*.

linha 4: Chamada da função para imprimir a frase "Hello de uma goroutine".

linha 5: Envia-se um sinal para o canal (c), neste caso poderia ser qualquer número. A *goroutine* sabe que pode passar por esse canal.

linha 10: Cria-se uma *goroutine* para a função *f*.

linha 12: Neste ponto, o canal espera a *goroutine* terminar a execução da função *f*. Depois que a *goroutine* terminar a sua tarefa, o programa é finalizado.

O exemplo acima mostra um canal sem *buffer*. Canais sem *buffer* fazem com que o transmissor fique bloqueado até que o receptor tenha recebido todo o valor; no caso do exemplo da Figura 2, o valor zero (ou sinal) é recebido depois que a tarefa é terminada. Se o canal é com *buffer*, o transmissor é bloqueado se algum valor for copiado para o *buffer*. Se o *buffer* estiver cheio, o transmissor vai espera até que algum receptor receba todo o valor. O receptor fica bloqueado até ter algum dado para receber.

2.4.3 Execução em CPUs *multicore*

Se um determinado cálculo puder ser dividido em partes separadas, executadas independentemente, isso pode ser paralelizado. Com *Go*, pode-se usar um canal para sinalizar quando cada parte estiver completa.

A programação paralela em CPU na linguagem *Go* ocorre quando se especifica ao *run-time* quantas *goroutines* serão executadas simultaneamente no código. Para definir isso, utiliza-se do pacote *runtime* a função *runtime.GOMAXPROCS(x)*, onde 'x' é a quantidade de *cores* de CPU utilizados. A linguagem *Go* também fornece a função *runtime.NumCPU()* que informa a quantidade de núcleos de processamento disponíveis na máquina.

```
1  func main() {
2      canal1 := make(chan int)
3      runtime.GOMAXPROCS(runtime.NumCPU())
4
5      for i := 1; i <= 100000; i++ {
6              go func(num int) {
7                  fmt.Println("Executando goroutines", num)
8                  canal1 <- 0
9              }(i)
10     }
11     fmt.Println("=============================================")
12     <-canal1
13 }
```

Figura 3 – Loop utilizando todas as CPUs lógicas da máquina
Fonte: Autoria própria

A Figura 3 mostra um código que usa todos os núcleos da CPU *multicore*. As *goroutines* são disparadas nos núcloes e vão imprimindo o valor correspondente da variável *i*. Na linha onze (11) temos uma *string* a ser impressa na tela que vem antes da sinalização do canal *canal1*. O resultado desse código será uma sequencia de números impressos não estando em ordem crescente (como ocorre geralmente em um laço executado de modo sequencial).

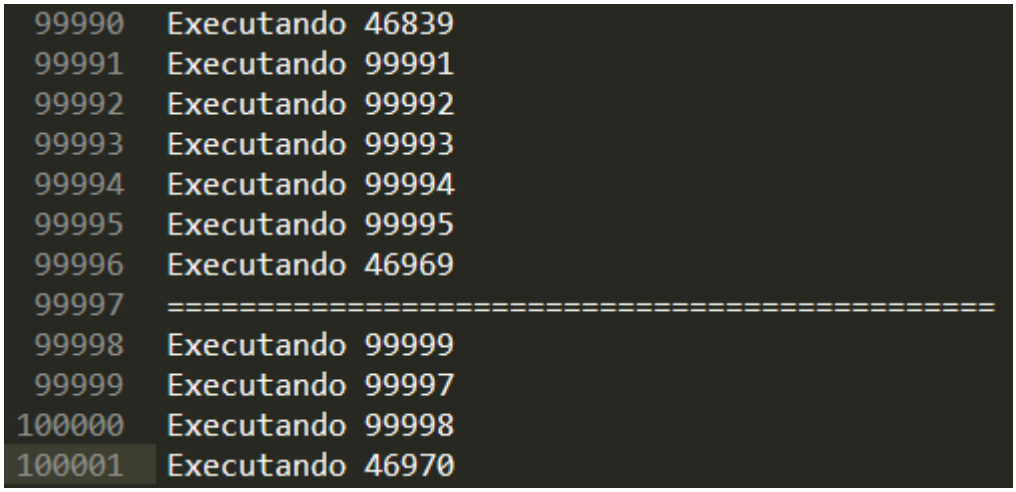


Figura 4 – Resultado das últimas onze linhas do exemplo da Figura 3
Fonte: Autoria própria

A Figura 4 mostra o resultado das últimas onze linhas do código exemplo da Figura 3. As *goroutines*, no decorrer do programa, sabem quais números foram contados pelas outras *goroutines* executadas em outros núcleos, ou seja, há comunicação entre elas, entretanto, não há sincronização.

2.5 ALGORITMOS GENÉTICOS

Algoritmos genéticos são modelos computacionais baseados em mecanismos evolutivos de seleção, reprodução e mutação. São métodos de busca e otimização que simulam os processos naturais de evolução (PESSINI, 2003). Através desses algoritmos podem-se codificar soluções para um problema específico de modo a manter informações importantes da estrutura de dados. (WHITLEY, 1994).

Uma implementação de um algoritmo genético começa com uma população inicial de cromossomos, sendo que cada um deles representa uma possível solução para um determinado problema. Esses cromossomos são submetidos a uma avaliação (*fitness*) que define se o determinado cromossomo representa uma melhor solução para o problema, garantindo-lhe mais chances de reprodução do que os cromossomos que não representam uma melhor solução.

A partir de um elemento da população, operações aleatórias análogas à mutação, geram novos elementos. Todos são submetidos à avaliação e os melhores classificados são usados para um novo ciclo.

A população inicial de cromossomos não pode ser pequena para evitar a tendência de produzir divergências prematuras (WHITLEY, 1994). Porém o uso de uma população relativamente grande é diretamente proporcional ao seu custo computacional, ou seja, quanto maior a população, maior será o tempo e o uso de processamento para obter a solução.

Como foi citado anteriormente, algoritmos genéticos simulam processos naturais de evolução. Parte da metáfora biológica usada para motivar a pesquisa genética é que ela é de natureza paralela, ou seja, em uma perspectiva natural de população, milhões de indivíduos podem evoluir simultaneamente (WHITLEY, 1994). Logo, o grau de paralelismo utilizado em uma pesquisa genética é diretamente proporcional ao tamanho da população utilizada, ou seja, pode-se explorar o paralelismo em algoritmos genéticos.

O objetivo deste trabalho não é realizar um estudo específico sobre algoritmos genéticos, mas sim aproveitar a oportunidade de aplicar paralelismo. Ao trabalhar com algoritmos genéticos em problemas específicos podem-se usar representações adaptadas a fim de atingir aquilo que for mais apropriado (COPPIN, 2010). Em termos de prática, os algoritmos genéticos têm um amplo impacto sobre problemas de otimização, entretanto, muito trabalho ainda tem de ser feito para mostrar em quais especificações os algoritmos genéticos funcionam bem (RUSSEL *et al*, 2004), o que foge ao escopo deste trabalho.

2.5.1 Algoritmos Genéticos e Paralelismo

Um algoritmo genético pode ser paralelizado de forma que cada cromossomo tenha seu próprio processador para executar os cálculos (PESSINI, 2003). No entanto, alguns algoritmos genéticos são inspirados em processos evolutivos paralelos de uma população de indivíduos, ou seja, o modelo natural do algoritmo ser propriamente para a execução paralela.

Existem vários modelos que viabilizam o processo de paralelização (WHITLEY, 1994). Neste trabalho serão analisadas três modelos diferentes de explorar o paralelismo em algoritmos genéticos. São eles: algoritmo genético canônico, modelo da Ilha e algoritmos genéticos celulares.

2.5.1.1 Algoritmo genético canônico

Este modelo opera sobre a implementação geral dos algoritmos genéticos com uma pequena modificação. Começa com uma população de cromossomos inicial, representando possíveis soluções ao problema. Para cada geração a seguir de cromossomos, será realizada uma seleção através de uma avaliação com uma função, chamada de *fitness*. Os indivíduos com melhores valores de adaptação possuem maior probabilidade de reprodução. É possível modificar as características da população através de mutações. Após consecutivas aplicações dos operadores (seleção, cruzamento e mutação) espera-se que os cromossomos tornem-se uma boa solução, não sendo necessariamente a ótima (WHITLEY, 1994).

A modificação para este modelo é que a seleção não é utilizada a função *fitness*, mas sim por meio de um torneio onde dois indivíduos são escolhidos e a mais adaptado entre os dois é selecionado.

Com essa abordagem, um processador *dual core* pode conter dois torneios independentes, em seguida, mantém-se o vencedor de cada torneio que representam a geração intermediária para dar sequência aos outros operadores. O cruzamento e a mutação podem ocorrer em paralelo (WHITLEY, 1994).

2.5.1.2 Algoritmos genéticos celulares

Tendo em vista o seguinte exemplo:

> Supõe-se que temos 2500 (dois mil e quinhentos) núcleos dispostos em uma grade de duas dimensões 50 por 50 (50x50). Cada núcleo só pode se comunicar com seus vizinhos de cima, da esquerda, da direita e de baixo. Os núcleos das bordas da grade se juntam de modo a formar um torus (WHITLEY, 1994, p.33).

Para cada núcleo (ou célula) atribui-se um indivíduo da população. Cada indivíduo escolhe o melhor dos seus vizinhos para a reprodução, ou pode-se também usar algum método probabilístico. Dessa reprodução é gerado apenas um filho que será o novo indivíduo desse núcleo.

Esse modelo pode criar um isolamento por distância (WHITLEY, 1994), isto é, subpopulações uma distante da outra, como o modelo das ilhas (apresentado na

seção seguinte). Através dessa abordagem as populações cuja distância é menor possuem um maior índice de interação, logo, podem trocar material genético mais facilmente.

2.5.1.3 Modelo das ilhas

Neste modelo, a população total é dividida em subpopulações; cada subpopulação reside em uma ilha e cada ilha possui seu próprio sistema de evolução; após algumas gerações, os melhores indivíduos de cada ilha são capazes de migrar para as outras ilhas, compartilhando assim material genético (WHITLEY, 1994).

Com esse modelo, cada subpopulação reside em um núcleo do processador (ilha). O sistema evolução (algoritmo genético) é executado paralelamente. Cada núcleo executa seu próprio algoritmo genético e depois de um determinado tempo é feita a migração dos melhores indivíduos de uma ilha para a outra, ou seja, aquelas que proporcionam a melhor solução para um determinado problema serão mandados para outras ilhas para serem executados em outros algoritmos genéticos.

2.6 ALGORITMO DO RECOZIMENTO SIMULADO

O recozimento simulado (*simulated annealing*) ou têmpera simulada, originou-se da semelhança encontrada do processo físico de resfriamento de metais em estado de fusão com problemas de otimização. A têmpera simulada está relacionada com minimização de custo (RUSSEL *et al*, 2004).

Pode-se utilizar o algoritmo da têmpera simulada para procurar o ponto de mínimo de uma função. Parte-se de um ponto inicial e no laço de repetição mais interno do algoritmo faz-se um movimento aleatório, inicialmente com saltos grandes (metais em altas temperaturas) para depois ir diminuindo os movimentos lentamente (baixando temperaturas) (RUSSEL *et al*, 2004) até achar a solução ótima em que a função tem o menor valor (temperatura ideal para se obter um estado de baixa energia no sólido).

O recozimento dos metais é realizado de maneira muito cuidadosa para se levar a baixos níveis de energia interna do material formando uma estrutura

cristalina; este processo de achar um estado de baixa energia pode ser simulado através de uma solução de um problema de otimização (SOEIRO *et al*, 2009).

A função objetivo (ou custo) corresponde ao nível de energia do sistema que se deseja minimizar; a temperatura do sistema físico não tem equivalente no problema de otimização, será apenas um parâmetro de controle (SOEIRO *et al*, 2009).

2.7 COMBINANDO ALGORITMO GENÉTICO E TÊMPERA SIMULADA

Algoritmos genéticos e algoritmos de têmpera simulada são métodos de busca heurística. As informações do problema são utilizadas para guiar a busca e prevenir o programa de forma que este não perca tempo processando os dados de regiões improváveis de se encontrar a solução (NETO, 2010). Esses algoritmos são usados quando não há uma solução “fechada” ou “pronta” e é necessário buscar a solução do problema mudando as variáveis de entrada.

Uma das estratégias do método heurístico é utilizar uma classificação através de uma função de avaliação. Essa função estrutura os valores de forma que as melhores soluções sejam posicionadas no início de uma fila.

Os métodos heurísticos podem ser modificados, gerando novos algoritmos que podem facilitar a busca de determinado problema (SÖRENSEN, 2010). Baseado nisso, é possível combinar algoritmos genéticos e o algoritmo da têmpera simulada para formar uma nova heurística. Essa ideia é apresentada, por exemplo, em ADLER (1993) e em KOSCIANSKI (2004).

O método utilizado nos estudos deste trabalho aplica a heurística da têmpera simulada nos indivíduos da população de um algoritmo genético. O cromossomo sofre uma mutação baseada em uma temperatura que diminui gradativamente com o tempo; essa temperatura indica quanto o cromossomo variou em relação ao seu estado anterior, de acordo com essa fórmula:

$$crom(t+1) = crom(t) + \tau * rand(-1,+1) \quad ,$$

onde ‘crom’ é um valor real, ‘t’ é o tempo ou iteração do algoritmo, ‘τ’ é a temperatura e ‘rand’ é um gerador de números aleatórios. Um número randômico de menos um a um é gerado e multiplicado à temperatura. Nesta abordagem, não são aplicados os conceitos de *crossover* (reprodução).

2.8 *COMPUTE UNIFIED DEVICE ARCHITECTURE* (*CUDA*)

CUDA é uma arquitetura de programação paralela criada pela NVIDIA com o objetivo de melhorar o desempenho computacional através do uso do processamento gráfico das GPUs. Com essa arquitetura as placas de vídeo que antes eram restritas apenas para a computação gráfica podem ser usadas para propósitos gerais (GPGPU – *General Purpose Graphics Processing Unit*).

Uma GPU que tem a disponibilidade para a programação com *CUDA* pode possuir centenas ou mais *cores* (núcleos) onde podem ser executados milhares de *threads* computacionais (NVIDIA, 2013). Essa arquitetura é composta por *Streaming Multiprocessors* (SMs), a quantidade de SM contida na placa de vídeo varia de uma geração para outra de GPU da NVIDIA CUDA (KIRK *et al*, 2010).

> A programação específica para placas gráficas possui uma abordagem diferente em comparação a programação em CPU devido a diferenças na arquitetura (como registradores de processamento rápido limitados, recursos de compartilhamento de memória) e o uso das *threads* para o mapeamento das tarefas (SUCHARD, 2010, p. 10).

2.8.1 Blocos e Grades

Em *CUDA*, um bloco é uma unidade de organização e mapeamento de *threads* sobre o *hardware*; cada bloco é alocado a um multiprocessador da GPU (NVIDIA, 2010).

Uma grade é uma estrutura onde se definem um conjunto de blocos e *threads* que serão criados e gerenciados pela GPU para uma determinada função (NVIDIA, 2010).

Os blocos (*blocks*) de *threads* são organizados em grades (*grids*). Os blocos podem ser unidimensionais (vetores) ou bidimensionais (matrizes). As grades podem ser unidimensionais, bidimensionais ou tridimensionais.

A Figura 5 é uma representação da grade de tamanho 3x2 contendo blocos de tamanho 4x3. Tantos os blocos quanto os *threads* possuem sua própria identificação (ID).

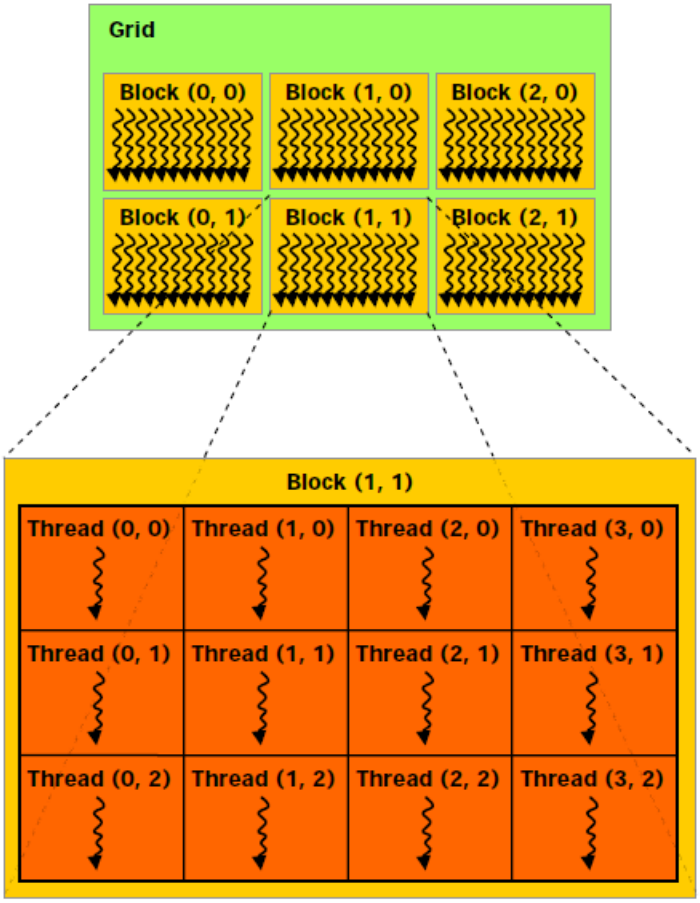


Figura 5 – Imagem representativa de grade, blocos e threads
Fonte: NVIDIA (2010)

2.8.2 Função *Kernel*

Com *CUDA*, programação em GPU é realizada através de uma função chamada *kernel*, essa função é executada na máquina por várias *threads* que rodam em diferentes processadores da placa gráfica (FRANCO, 2009).

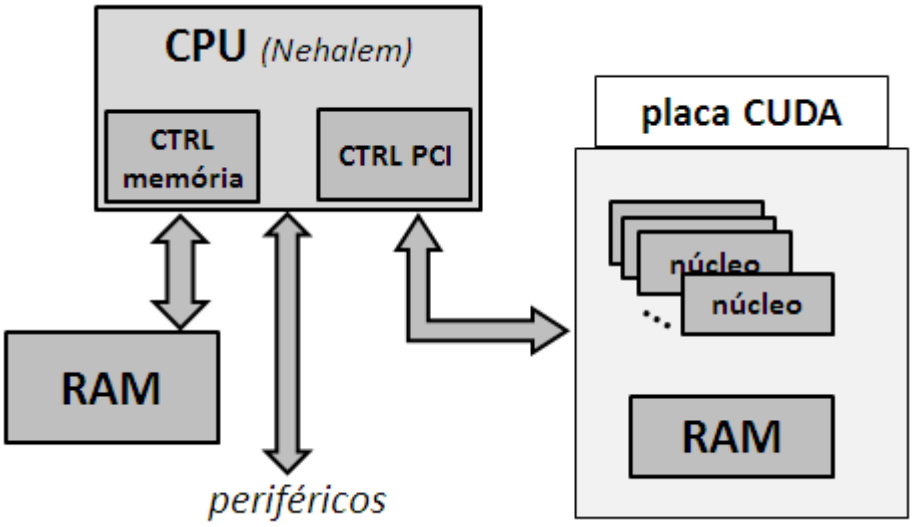


Figura 6 – Modelo de relacionamento das placas
Fonte: Autoria própria

A chamada do *kernel* é realizada pela CPU. Todos os dados e configurações são inicializados pela CPU e copiados para a memória da GPU, após a GPU terminar as atividades, os resultados devem ser retornados para a memória principal. A Figura 6 ilustra o relacionamento dos principais componentes do computador. A transferência de dados entre CPU e GPU é realizada explicitamente pela função *cudaMemcpy()*. Recentemente, a versão seis de *CUDA* facilitou esse processo através de uma tecnologia chamada memória unificada, o que torna esse processo transparente.

Em um bloco, contem-se *threads* que trocam de maneira eficiente os dados através do compartilhamento de memória e a sincronização desse bloco durante a execução. Cada bloco de *thread* é executado em um multiprocessador e cada *thread* possui uma identificação (FRANCO, 2009).

CUDA referencia a CPU como *host* e a GPU como *device*.

Com o *kernel*, nota-se que a programação paralela em *CUDA* se torna simplificada, pois não há necessidade de escrever códigos com *threads* de maneira convencional.

Os *kernels* são executados em paralelo. As *threads* contidas dentro dos blocos se comunicam através do compartilhamento extremamente rápido da memória, essa otimização no acesso da memória é fundamental para um bom desempenho da GPU (SUCHARD, 2010).

A definição das dimensões das grades e dos blocos para a organização das *threads* é realizada com uma extensão de sintaxe da linguagem C, <<<*gridSize,*

blockSize>>>, usada entre a função *kernel* e a lista para a passagem de parâmetros (SUCHARD, 2010).

A sequência de execução básica de um algoritmo em *CUDA* segue os seguintes passos: os dados são processados e inicializados pelo *host* (CPU), em seguida os dados processados são copiados para a memória do *device* (GPU) que após o processamento (execução do *kernel* para as tarefas paralelas), são retornados para a memória principal.

```
1  void somaVetorCuda(int *c, const int *a, const int *b, unsigned int size)
2  {
3      int *dev_a = 0;
4      int *dev_b = 0;
5      int *dev_c = 0;
6
7      cudaMalloc((void**)&dev_c, size * sizeof(int));
8      cudaMalloc((void**)&dev_a, size * sizeof(int));
9      cudaMalloc((void**)&dev_b, size * sizeof(int));
10
11     cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
12     cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
13
14     somaKernel<<<1, size>>>(dev_c, dev_a, dev_b);
15
16     cudaDeviceSynchronize();
17
18     cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
19
20 }
```

Figura 7 – Função das execuções básicas da soma de vetores em *CUDA*
Fonte: Autoria própria

A Figura 7 ilustra um trecho de um programa que realiza a soma paralela de dois vetores, conforme os passos citados anteriormente da implementação básica de um algoritmo em CUDA:

linhas 3, 4, 5: Três vetores são criados (*dev_a, dev_b, dev_c*), um deles (*dev_c*) vai ser o resultado da soma dos outros dois (*dev_a*, *dev_b*).

linhas 7, 8, 9: Aloca o espaço dos três vetores na memória GPU.

linhas 11, 12: Copia os vetores com os dados (*a, b*) da CPU para a GPU.

linha 14: Função *kernel* é chamada para o plano de execução paralela.

Antes da lista de argumentos, define-se o tamanho da grade e o tamanho do bloco, ou seja, neste caso, temos uma grade contendo uma *thread* para cada elemento. A quantidade de elementos está definida pela variável *size* que é passada por parâmetro.

linha 16: Sincronização de todas as tarefas iniciadas por qualquer *thread* dentro do bloco de *thread* (NVIDIA, 2013), ou seja, espera-se a função *kernel* terminar para depois dar continuidade ao programa.

linha 18: Copia o vetor (*dev_c*) que contém a soma dos dois vetores da GPU para o vetor (*c*) da CPU.

```
1  __global__ void somaKernel(int *c, const int *a, const int *b)
2  {
3      int i = threadIdx.x;
4      c[i] = a[i] + b[i];
5  }
```

Figura 8 – Função kernel para execução em paralelo
Fonte: Autoria própria

Tomando como referência o código da Figura 7, temos o código da Figura 8. A função *kernel* é definida usando a declaração __*global*__. Essa qualificação define que a função é executada no *device* e que pode ser chamada tanto do *host* quanto do *device*:

linha 3: Índice da *thread* no bloco de *thread*.

linha 4: Soma dos elementos dos vetores.

2.8.3 Qualificadores de Funções *Cuda*

Esses qualificadores definem como uma função vai ser executada, se é no *device* ou no *host*. Em *CUDA* existem três qualificadores (SUCHARD, 2010):

- __*global*__: visto na seção anterior, qualifica que a função ou variável é executada/armazenada no *device* e ela pode ser chamada do *host* ou do próprio *device*. São especificamente usados para as funções *kernel* (NVIDIA, 2013);
- __*device*__: qualifica que a função ou variável é executada/armazenada no *device* e somente pode ser chamada a partir do *device*;
- __*host*__: qualifica que a função ou variável é executada/armazenada somente no *host* e que só pode ser chamada a partir do *host*.

2.9 FUNÇÃO GRIEWANK

Para efeito de teste do algoritmo genético, optou-se por realizar a otimização de uma função matemática. No presente caso, foi selecionada a função de Griewank, bastante comum em testes de algoritmos de otimização. A função Griewank apresenta vários pontos mínimos diferentes. A Figura 9 mostra a irregularidade que a função pode apresentar.

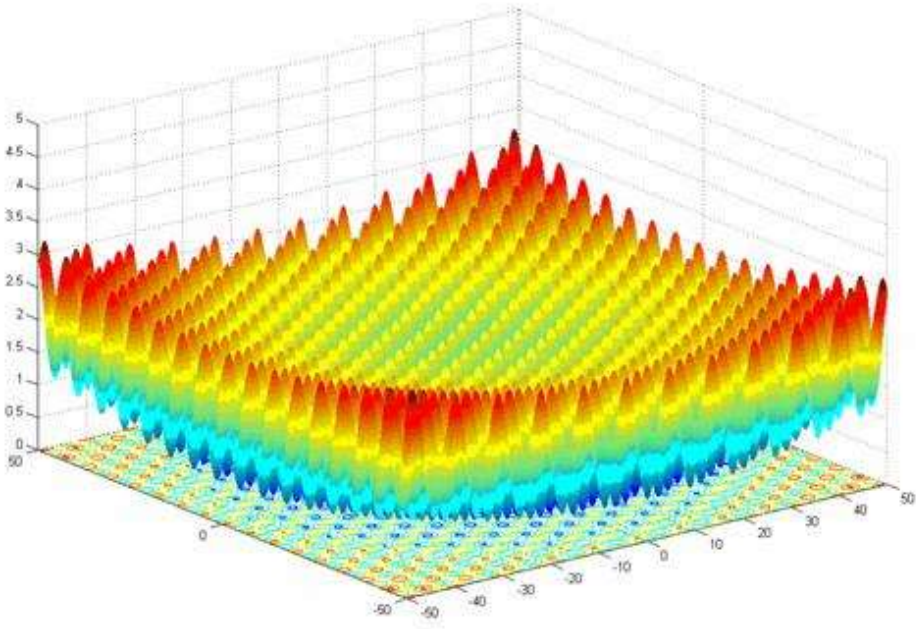


Figura 9 – Gráfico exemplo da função de Griewank
Fonte: HEDAR (2006)

Como se pode observar na Figura 9, a superfície traçada pela função é bastante irregular. A partir de um ponto aleatório, deseja-se encontrar o ponto mínimo através de saltos para encontrar pontos ainda menores. Este processo pode ser demorado, entretanto, ao se utilizar a concorrência no programa, podem-se ter dois ou mais saltos realizados ao mesmo tempo.

A função é multidimensional e calculada por

$$f(x_1, \dots, x_n) = 1 + \frac{1}{4000}\sum_{k=1}^{n} x_k^2 - \prod_{k=1}^{n} \frac{\cos x_k}{\sqrt{k}} \ .$$

Neste estudo utilizamos dimensão *n* = 2.

2.10 ORDENAÇÃO BITÔNICA

O princípio do algoritmo de ordenação bitônica (ou *bitonic sort*) baseia-se no compara-e-troca, ou seja, comparam-se dois elementos de uma sequência de números e caso estiverem fora de ordem, trocam de posição. Este algoritmo é referenciado em vários exemplos de programação *CUDA*.

A ordenação bitônica opera sobre uma sequência de comprimento *n* onde *n* é uma potência de dois. Uma sequência de números também é definida bitônica (*e.g.* $a_1, a_2, a_3, \ldots, a_n$) onde existe um inteiro $1 \leq j \leq n$ tal que $a_1 \leq a_2 \leq \ldots \leq a_j \geq a_{j+1} \geq \ldots \geq a_n$ (GONDA *et al*, 2005).

No algoritmo descrito por Gonda *et al* (2005), assume-se que o número de elementos *n* da sequência a ser ordenada e o número de processadores *p* são potências de dois. A ideia do algoritmo descrito é baseada na operação de divisões bitônicas sucessivas e ordenações locais, até que toda a sequência esteja ordenada.

A implementação utilizada na ordenação bitônica no trabalho é baseada no trabalho de Ayushi Sinha do *Providence College* (SINHA, 2011). O presente trabalho utiliza a ordenação bitônica para o algoritmo genético implementado em CUDA, pois este não possui funções para ordenação de números decimais.

2.11 ESTUDO DE CASOS SELECIONADOS

Os problemas de paralelismo utilizados neste trabalho foram implementados com a finalidade de identificar e entender como as plataformas trabalham a concorrência sobre as arquiteturas *multicore*, bem como entender conceitos técnicos da programação paralela.

A sequência de Fibonacci calculada recursivamente pode ser solucionada adotando o paralelismo. Cada número da sequência é calculado independentemente por uma chamada recursiva. Pode-se realizar isso de modo paralelo, todos os números ao mesmo tempo, e cada chamada com sua pilha. Este método adotado é ineficiente em relação ao uso de memória e ao uso de CPU, mas foi utilizado para finalidades de testes e implementação.

A multiplicação de matrizes por não apresentar dependência de dados pode ser solucionada através de uma técnica paralela. Os elementos da matriz resultante podem ser calculados todos ao mesmo tempo.

O paralelismo em algoritmos genéticos reflete na própria metáfora. Indivíduos existem concorrentemente um com os outros, assim como também os processos climáticos, que podem ocorrer em várias ilhas ao mesmo tempo.

2.12 TRABALHOS RELACIONADOS

O uso das GPUs como propósito geral tem se tornado constante. Muitos pesquisadores têm feito a comparação de desempenho entre o uso das CPUs e GPUs, como Pospíchal *et al* (POSPÍCHAL, 2009), que usou as funções de Rosenbrock, Griewank e Michalewicz.

O trabalho realizado por Lee *et al* (LEE, 2010) trata exatamente dessa comparação. Os autores apresentam uma análise da programação paralela aplicando técnicas de otimização sobre CPU e GPU. Este estudo baseia-se na comparação de vários problemas diferentes. As aplicações escolhidas possuem um nível de paralelismo elevado, que por natureza podem ser modeladas sobre as arquiteturas multicores moderna. Na pesquisa, concluiram que *multithreading*, *cache blocking* e reorganização de acesso à memória SIMD (*Single Instruction, Multiple Data*) são técnicas que possuem um melhor desempenho na CPU; já para GPU especifica como minimização da sincronização global e *buffers* localmente compartilhados como as duas técnicas chaves de melhor desempenho.

Como foi citado nos capítulos anteriores, *CUDA* utiliza blocos para mapeamento de *threads* na GPU. A comunicação das *threads* que ocorrem dentro do bloco ocorre na memória global, e requer uma sincronização (XIAO *et al*, 2010). Essa operação é realizada apenas na GPU e causa uma sobrecarga que compromete o desempenho da aplicação. Baseado nisso, os autores do trabalho citato propõem métodos mais eficientes para essa sincronização, apresentando um modelo de desempenho para a execução dos *kernels*. Este exemplo mostra como o conhecimento da arquitetura e a organização do código pode ter um efeito muito importante na velocidade de execução.

A plataforma *CUDA* é originalmente uma aplicação para GPU, entretanto, é possível utilizar os *kernels* sobre a arquitetura CPU, exemplo disso é o *framework* MCUDA (STRATTON *et al*, 2008).

A programação paralela sobre a CPU já têm sido aplicada através de ferramentas como OpenMP, Pthreads e Cilk, como é o caso dos trabalhos de Bordin *et al* (2010) e Guimarães (2014). Este último realiza uma análise de desempenho com multiplicação de matrizes e cálculo da sequência de Fibonacci como escopo. Assim como *CUDA*, essas ferramentas utilizam uma linguagem de programação existente (como C) e criam mecanismos de programação para explorar as arquiteturas *multicore*. Com uma perspectiva diferente, a linguagem *Go* trata o paralelismo em sua própria estrutura de linguagem, como mostrado no trabalho de Tang *et al* (2010) que estuda a linguagem em dois problemas de computação paralela: integração paralela e programação paralela dinâmica.

As peculiaridades de arquitetura CUDA exigem conhecimento específico para implementação de soluções. A mesma coisa acontece em outras arquiteturas e bibliotecas, embora algumas delas sejam anteriores a CUDA e já são usadas por um certo número de programadores; exemplo disso é o padrão OpenCL. Isso motivou trabalhos de pesquisadores em torno de bibliotecas e *front-ends*, por exemplo, para traduzir código de uma plataforma para outra. Alguns exemplos de trabalhos em torno dessas ideias incluem Harvey e Fabritis (2011), van Dick *et al* (2009), Nawatta e Suda (2011), Klöckner *et al* (2012), Yang e Lin (2011).

Outra maneira de explorar a arquitetura CUDA é usando placas GPU em um *cluster*; um exemplo nesse sentido é apresentado por De La Asunción *et al (2012)*, usando quatro computadores conectados em rede. A comunicação entre os computadores nesse caso empregou MPI (*Message Passing Interface*).

Finalmente, outras arquiteturas de GPU também podem ser exploradas para paralelismo. Exemplificando, o trabalho de Du *et al* (2012) cria uma ferramenta para gerar código OpenCL ou Cuda, de maneira transparente para o programador que deseja empregar placas gráficas que usem um desses padrões.

3 DESENVOLVIMENTO

Neste capítulo será apresentada a programação dos problemas paralelizáveis com as ferramentas utilizadas no decorrer do trabalho.

3.1 METODOLOGIA

As atividades desenvolvidas no presente trabalho são divididas em três etapas. A primeira etapa, já abordada, foi a pesquisa bibliográfica cujo objetivo é buscar um embasamento teórico em relação à programação paralela, em relação às ferramentas de estudo (*Go* e *CUDA*) e como essas ferramentas trabalham. Também se abordou os problemas escolhidos para estudo (como a função de Griewank), os algoritmos genéticos e têmpera simulada.

A segunda etapa consiste na familiarização das plataformas (*Go* e *CUDA*) através do desenvolvimento de aplicações paralelas de tarefas específicas. Isto incluiu: multiplicação de matrizes, o cálculo da sequência de Fibonacci e algoritmos genéticos. Esses algoritmos possibilitam uma visão de como as ferramentas trabalham com o paralelismo. Todos esses algoritmos estão documentados nos apêndices.

Na última etapa é feita uma discussão com um parecer da programação paralela realizada nas duas plataformas.

3.1.1 Método

A realização deste trabalho deu-se através do método de pesquisa de análise comparativa, com o intuito de identificar diferenças e semelhanças no que diz respeito à facilidade de implementação.

3.1.2 Ambiente de programação

Todos os códigos contidos no escopo deste trabalho foram desenvolvidos em um computador de sistema operacional *Windowns 7 64bits*, com o processador *Intel i5 2410M 2.3 GHz TurboBoost 2.9 GHz* com dois núcleos de processamento e

placa de vídeo *NVIDIA GeForce GT 540M TurboCache 1713Mb* com noventa e seis núcleos de processamento. Foi utilizado o pacote binário de *Go* versão do compilador 1.2.2. Os programas em *Go* foram implementados e testados na plataforma *LiteIDE*, e os programas com CUDA implementados no *Visual Studio 2012*, com versão do compilador *CUDA* 5.5.

3.2 PROGRAMAÇÃO PARALELA COM GO

Para o uso desta linguagem, é necessário que o programador esteja familiarizado com o ambiente e com a ferramenta de desenvolvimento. Baseado nisso, os desenvolvedores da linguagem *Go* criaram um tutorial contendo programas que demonstram os diferentes aspectos da linguagem, empregando alguns exercícios para uma compreensão mais aprofundada.

Este tutorial é computado em um ambiente de execução de programas. Este ambiente é chamado de *Go Playground*, um serviço web que recebe um programa em um servidor remoto, para depois devolver o resultado.

Para o estudo e familiarização da linguagem realizou-se o tutorial inteiro. Depois se fez a implementação de alguns exemplos, que serão apresentados nos tópicos a seguir. O objetivo do trabalho não é apresentar um tutorial da nova linguagem, entretanto é de interesse mostrar o modo como foi compreendido e aplicado os recursos que a linguagem nova têm em relação à implementação concorrente.

3.2.1 Paralelismo em *Go*: Processando *Goroutines*

Na seção 2.4.3 foi visto o uso da programação paralela com os recursos da linguagem para a programação concorrente: *goroutine*, *channel* e *gomaxprocs*. Durante a execução do programa, verificou-se o uso de todas as CPUs do computador, porém faltava um ponto importante quando se trata de processos paralelos: a sincronização.

O que ocorre no código demonstrado na seção 2.4.3 é que o canal utilizado funciona apenas como um sinalizador: quando uma *goroutine* do laço termina sua tarefa, o canal diz que ele está liberado para executar outra tarefa (é como se as

goroutines estivessem em uma corrida: a primeira que termina a tarefa mostra na tela).

```
10  func main() {
11          c := make(chan string)
12          runtime.GOMAXPROCS(runtime.NumCPU())
13          for i := 0; i <= 100000; i++ {
14                  go func(num int) {
15                  c <- fmt.Sprintln("Executando goroutines", num)
16                  }(i)
17          }
18          //Espera terminar
19          for i := 0; i <= 100000; i++ {
20              fmt.Print(<-c)
21          }
22  }
```

Figura 10 – Execução de *goroutines*
Fonte: Autoria própria

A Figura 10 mostra o uso do canal como um meio por onde as *goroutines* guardam informações. Ao invés de ser apenas um sinalizador, o canal é usado para transferir dados: o canal guarda todas as informações processadas pelas *goroutines* no primeiro laço de repetição (linhas 13~17) que realizam as tarefas nos vários núcleos da CPU; em seguida, o segundo laço (linhas 19~21) descarrega as informações contidas no canal, apresentando os dados na ordem correta.

3.2.2 Paralelismo em *Go*: Fibonacci Recursivo

O cálculo de Fibonacci inicia-se com os elementos 0 (zero) e 1 (um) e os valores a partir do terceiro número da sequência é o resultado da soma dos dois números anteriores, conforme mostrado na seguinte fórmula:

$$F(n) = \begin{cases} se\ n \leq 1, n \\ se\ não, F(n-1) + F(n-2) \end{cases}$$

Na Figura 11, tem-se o código de uma função recursiva que realiza o cálculo da sequência de Fibonacci na linguagem *Go*.

```
 9 func fibonacci(num int) int {
10     if num == 0 || num == 1 {
11         return 1
12     } else {
13         return fibonacci(num-1) + fibonacci(num-2)
14     }
15 }
```

Figura 11 – Cálculo da sequência de Fibonacci recursivo
Fonte: Autoria própria

Utiliza-se a abordagem de paralelização introduzida no capítulo anterior: passa-se o resultado da função de *fibonacci()* em um canal de comunicação das *goroutines* para realizar o cálculo da sequência.

```
31     for i:=0; i < qtdNum; i++{
32         go func(n int){
33             c <- fibonacci(n+1)
34         }(i)
35     }
36
37     for i:=0; i < qtdNum ; i++{
38         fmt.Println(<-c)//imprime
39     }
```

Figura 12 – Calculando a sequência de Fibonacci com *goroutines*
Fonte: Autoria própria

No código ilustrado na Figura 12, mostra as *goroutines* calculando o número da sequência de Fibonacci definido pela variável *i* concorrentemente com outras *goroutines* nos núcleos da CPU.

3.2.3 Paralelismo em *Go*: Operações com Matrizes

A multiplicação de matrizes é um problema que permite o paralelismo, pois o é solucionado através de um conjunto de dados que podem ser divididos em grupos e subgrupos e as operações podem ser processadas de forma ágil combinando os resultados.

```
39      for l := 0; l < lines; l++ {
40          for c := 0; c < columns; c++ {
41              for i := 0; i < columns; i++ {
42                  matrix3[l][c] += matrix1[l][i] * matrix2[i][c]
43              }
44          }
45      }
```

Figura 13 – Multiplicação de matrizes na linguagem Go
Fonte: Autoria própria

A Figura 13 mostra o trecho de um programa em *Go* que realiza a multiplicação de matrizes em uma implementação sequencial.

```
for l := 0; l < lines; l++ {
    go func(line int) {
        for c := 0; c < columns; c++ {
            for i := 0; i < columns; i++ {
                matrix3[line][c] += matrix1[line][i] * matrix2[i][c]
            }
        }
        canal1 <- true
    }(l)

}
<-canal1
```

Figura 14 – Multiplicação paralela de matrizes na linguagem Go
Fonte: Autoria própria

Na Figura 14 temos o desenvolvimento do algoritmo de multiplicação de matrizes realizada com *goroutines*. A quantidade de *goroutines* inicializadas é equivalente com o número de linhas que a matriz possui (ou seja, uma goroutine para cada linha) e o canal é utilizado como um sinalizador fazendo com que a *goroutine* do programa principal espere todas as *goroutines* inicializadas no laço terminarem suas tarefas.

3.2.4 Algoritmo Genético em *Go*

O algoritmo genético inicial foi implementado na linguagem *Go* com o objetivo de estar mais familiarizado com a linguagem.

Para esta abordagem, utilizou-se a equação $z = x^2 + y^2$ como função objetivo. O algoritmo foi empregado para encontrar o ponto de mínimo. Cada indivíduo (z) do algoritmo é composto por dois genes (x, y). A heurística de algoritmo genético abordada é composta por laços de classificação e mutação.

Classificamos a população em ordem crescente da função de custo (ou seja, o primeiro valor do vetor será sempre o menor de todos) e aplicamos a mutação que consiste na alteração dos genes do cromossomo através da heurística da têmpera simulada. Isto significa que cada gene sofre uma modificação aleatória, fazendo o ponto (x, y) sair do lugar. Na têmpera simulada, isso equivale a um átomo do metal mudar de posição. Após essa mutação, recalcula-se o valor de $z = f(x, y)$ para a reclassificação do indivíduo.

Dentro do código, cada indivíduo foi armazenado em um registro.

```
12  type individuo struct {
13      x float64    //gene 1
14      y float64    //gene 2
15      z float64    //gene do individuo a partir do gene 1 e 2
16  }
```

Figura 15 – Indivíduo criado a partir de *structs*
Fonte: Autoria própria

O método de classificação nada mais é do que ordenar a população inteira de ordem crescente. Para esta operação utiliza-se a função *Sort* da linguagem *Go*.

```
94      sort.Sort(pop)
```

Figura 16 – Função *Sort* para ordenação
Fonte: Autoria própria

A mutação é uma pequena alteração no valor dos genes do indivíduo.

```
49  func gasMutante(individuo gene) gene {
50      //qtdInadala um pequeno valor no intervalo de 0 ~ 0.5
51      individuo.x *= (1.0 - qtdInalada) + (2 * qtdInalada * rand.Float64())
52      individuo.y *= (1.0 - qtdInalada) + (2 * qtdInalada * rand.Float64())
53
54      return individuo
55  }
```

Figura 17 – Função de mutação de genes
Fonte: Autoria própria

Os algoritmos genéticos no modelo das ilhas apresentados mais adiante foram implementados com base deste algoritmo genético inicial como modelo.

3.2.5 Algoritmo Genético do Modelo das Ilhas em *Go*

A função de Griewank foi usada para este algoritmo de otimização. Cada indivíduo é armazenado em um registro e o conjunto de indivíduos é armazenado em outro registro, representando uma ilha inteira.

```
27 type ilha struct{
28     crom[nIndv] cromossomo
29 }
```

Figura 18 – Ilha criada a partir de *structs*
Fonte: Autoria própria

A heurística do genético é aplicada. Cada gene do indivíduo sofrendo uma pequena mutação a cada rodada definida por uma variável chamada *alpha*.

```
70 func (ilha *TIlha) genetico (alpha float64){
71
72     canal := make(chan bool)
73
74     for i := 0; i < nIlhas; i++ {
75         go func (numIlha int){
76             for j:= 0; j < nIndv; j++{
77                 ilha[numIlha].crom[j] = alteraGene(ilha[numIlha].crom[j], alpha)
78                 ilha[numIlha].crom[j] = griewank(ilha[numIlha].crom[j])
79             }
80             sort.Sort(Tpop(ilha[numIlha].crom))
81             canal <- true
82         }(i)
83
84     }
85     <-canal
86 }
```

Figura 19 – Mutação de genes em Go
Fonte: Autoria própria

O paralelismo é aplicado da seguinte maneira: cada *goroutine* criada é responsável por uma ilha, então pode-se dizer que uma *goroutine* é uma ilha. No código ilustrado na Figura 19, a quantidade de *goroutines* criadas (linha 75) é definido pelo número de ilhas (*nIlhas*). Cada *goroutine* faz a mutação dos seus indivíduos (linhas 76~79) e também a classificação (linha 80). Um canal é usado (linha 81) para a sincronização das *goroutines*.

A função *GOMAXPROCS* deve ser inicializada na função principal para que as *goroutines* possam realizar as tarefas sobre os indivíduos nos vários núcleos da CPU.

3.3 PROGRAMAÇÃO PARALELA COM *CUDA*

Diferente da linguagem *Go* que trabalha apenas com a CPU, a plataforma *CUDA* utiliza a GPU. Através desse conceito, realiza-se grande parte do processamento das operações na placa de vídeo.

Os tópicos a seguir apresentam alguns programas implementados em *CUDA* para a familiarização da plataforma.

3.3.1 Paralelismo em *CUDA*: Fibonacci

Assim como em *Go*, o cálculo da sequência de Fibonacci foi implementada. Desejava-se de que cada *thread* criada (assim como cada *goroutine* criada em *Go*) ficasse responsável por calcular um número da sequência.

Entretanto, o programa com recursividade não pôde ser testado, pois a placa de vídeo utilizada na pesquisa não suporta chamada de funções recursivas dentro do *kernel*. Com essa limitação, foi necessário implementar uma versão não recursiva.

```
23  __global__ void kernel(int *d_sequenciaFib){
24      int i = threadIdx.x;
25
26      d_sequenciaFib[i] = fibonacci(i);
27
28      __syncthreads();
29  }
```

Figura 20 – Função kernel de chamada ao Fibonacci
Fonte: Autoria própria

Cada *thread* criada fica responsável por calcular um valor da sequência, armazenando-as em um vetor, como mostrado na figura acima. Cada *thread* chama a função *fibonacci()*, apresentado na Figura 20, o qual retorna o valor correspondente da sequência.

```
7   __device__ int fibonacci(int num){
8
9       if (num == 0 || num == 1){
10          return 1;
11      }else{
12          int aux, a = 0, b = 1;
13          for (int i = 0; i < num; ++i)
14          {
15              aux = a+b;
16              a = b;
17              b = aux;
18          }
19          return aux;
20      }
21  }
```

Figura 21 – Fibonacci implementado em CUDA
Fonte: Autoria própria

Na Figura 21 temos a função *fibonacci()*, com uma versão sequencial do cálculo.

3.3.2 Paralelismo em *CUDA*: Operações com Matrizes

Para a multiplicação de matrizes em *CUDA*, armazena-se uma determinada matriz em um vetor principal. Nesta abordagem, utilizamos matrizes quadráticas, ou seja, o número de linhas da matriz é igual ao número de colunas.

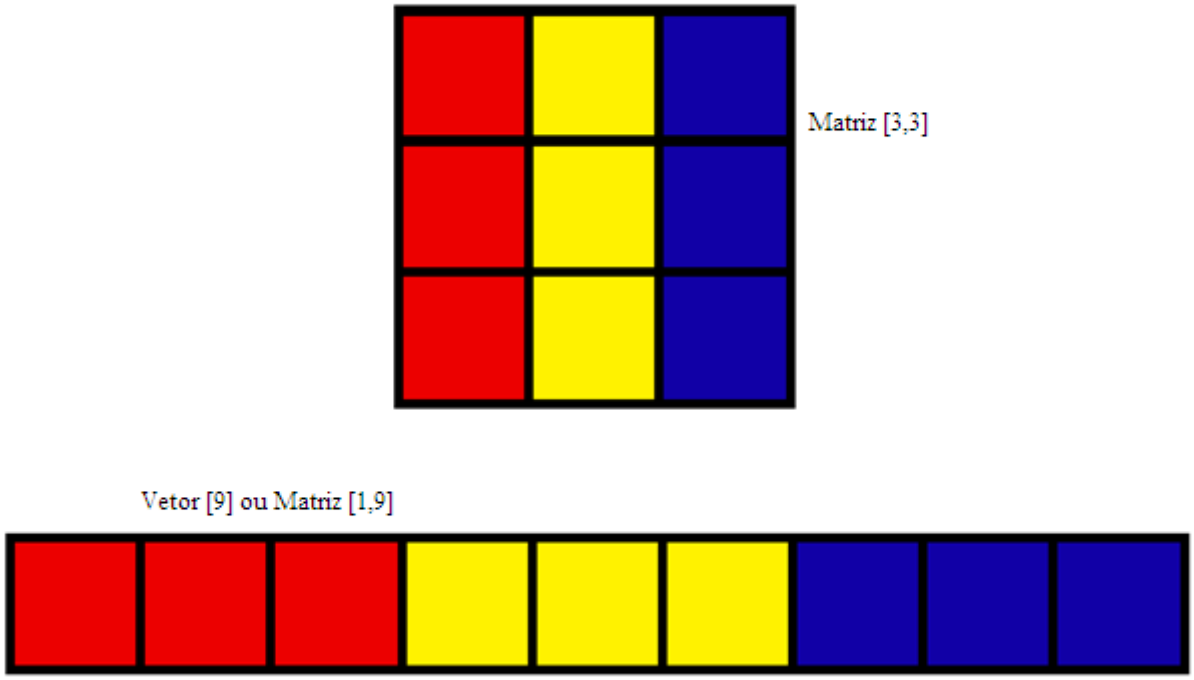


Figura 22 – Representação de Matrizes
Fonte: Autoria própria

Por exemplo, uma matriz quadrática 3x3 equivale a um vetor de nove posições, como mostrado na figura acima.

Dentro da arquitetura *CUDA*, configura-se o *kernel* de modo que cada *thread* criada fique responsável por calcular o resultado da multiplicação de uma linha da matriz resultante.

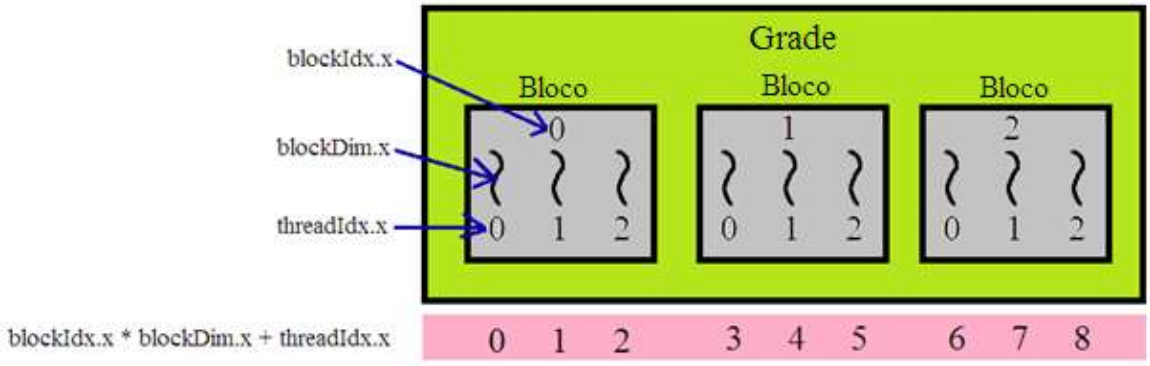


Figura 23 – Representação de uma grade em CUDA
Fonte: Autoria própria

Com o exemplo de uma matriz quadrática de dimensão três, criam-se três blocos (*blockIdx*) de dimensão três (*blockDim*), cada bloco com três *threads* (*threadIdx*). Através da equação

$$thread = blockIdx.x * blockDim.x + threadIdx.x$$

as *threads* são marcadas com um único ID, como mostrado na Figura 23. Um segundo conjunto de *threads* deve ser criado, tendo assim *threads* para o controle das linhas e colunas da matriz.

```
 7  __global__ void matrixMult(float *A, float *B, float *C, int N)
 8  {
 9      //Multiplicação de matrix C=A*B de tamanho NxN
10      //Cada thread computa um único elemento da matrix C
11      int linha = blockIdx.y*blockDim.y + threadIdx.y;
12      int coluna = blockIdx.x*blockDim.x + threadIdx.x;
13
14      float sum = 0.f;
15      for (int n = 0; n < N; ++n)
16          sum += A[linha*N+n]*B[n*N+coluna];
17
18      C[linha*N+coluna] = sum;
19
20      __syncthreads();
21  }
```

Figura 24 – Representação de Matrizes
Fonte: Autoria própria

O código mostrado na Figura 24 apresenta a função *kernel* que realiza a multiplicação de matrizes quadráticas de tamanho *N*.

3.3.3 Algoritmo Genético do Modelo das Ilhas em *Cuda*

O algoritmo genético tem como Griewank de função objetivo. Cada indivíduo é composto pelos seus genes onde é aplicada a heurística de algoritmo genético, aplicando-se mutações e classificações como no algoritmo implementado em *Go*. Foram adotado duas abordagens para esta plataforma.

Na primeira abordagem, relaciona-se o modelo das ilhas com a arquitetura de *CUDA*: cada *thread* é um indivíduo e cada bloco é uma ilha, sendo a grade representando o mundo.

```
141  __global__ void genetico(ilha *ilha, double alpha){
142      int j = threadIdx.x;
143      for (int i = blockIdx.x; i < nIlhas; ++i)
144      {
145          ilha[i].crom[j].x[0] += alpha * (2.0 * curand_uniform(&s) - 1.0);
146          ilha[i].crom[j].x[1] += alpha * (2.0 * curand_uniform(&s) - 1.0);
147          ilha[i].crom[j].y = griewank(ilha[i].crom[j].x);
148          __syncthreads();
149      }
```

Figura 25 – Mutação de genes em CUDA versão 1
Fonte: Autoria própria

Na Figura 25 temos a mutação dos indivíduos desta primeira abordagem. A função *curand_uniform()* (linha 151) retorna um valor randômico calculado pela própria *API*, e a função *__syncthreads()* (linha 153) realiza a sincronização das *threads*.

Neste programa, cada *thread* é responsável por um indivíduo de uma ilha aplicando a mutação (linhas 151~152), o laço que percorre as ilhas (linha 149) faz com que a função *curand_uniform()* não aplique o mesmo valor randômico em todas as ilhas, afinal de contas, com o paralelismo, as *threads* realizam as tarefas todas juntas e ao mesmo tempo, logo, o valor randômico calculado pelo sistema seria igual para todas no determinado tempo que as *threads* trabalham.

```
90          genetico<<<nIlhas, nIndv>>>(d_Ilha, alpha);
```

Figura 26 – Chamada da função *kernel* versão 1
Fonte: Autoria própria

Para a chamada do *kernel*, a quantidade de blocos configurada é o número de ilhas, e a quantidade de *threads* de cada bloco é a quantidade de indivíduos de cada ilha, como mostra a Figura 26.

Esta versão é interessante, entretanto, o número de indivíduos de cada ilha é limitado ao número de *threads* que a GPU suporta em cada bloco.

A segunda versão trabalha com um conceito um pouco diferente: criam-se arquipélagos (blocos), cada arquipélago contém várias ilhas (*threads*) e cada ilha vários indivíduos.

Ao invés de aplicar a mutação de apenas um indivíduo da ilha, cada *thread* fica responsável por aplicar a mutação em uma ilha inteira, então pode-se atribuir que uma thread é uma ilha.

Sendo um bloco contendo várias ilhas (*threads*), a população do mundo é relativamente maior comparada à primeira versão.

```
150  __global__ void genetico(ilha *ilha, double alpha){
151      int numeroIlha = threadIdx.x + blockIdx.x*blockDim.x;
152      for (int i = 0; i < nIndv; ++i)
153      {
154          ilha[numeroIlha].crom[i].x[0] += alpha * (2.0 * curand_uniform(&s) - 1.0);
155          ilha[numeroIlha].crom[i].x[1] += alpha * (2.0 * curand_uniform(&s) - 1.0);
156          ilha[numeroIlha].crom[i].y = griewank(ilha[numeroIlha].crom[i].x);
157      }
158      __syncthreads();
159  }
```

Figura 27 – Mutação de genes em CUDA versão 2
Fonte: Autoria própria

A segunda abordagem é mostrada na Figura 27. As *thread* (*numeroIlha*) são mapeadas nos vários blocos da GPU (linha 158). Cada *thread* fica responsável pelo laço que realiza a mutação dos genes (linhas 159~163).

```
97          genetico<<<nArquipelago, nIlhas>>>(d_World, alpha);
```

Figura 28 – Chamada da função *kernel* versão 2
Fonte: Autoria própria

Nesta versão, temos como constantes o número de indivíduos, o número de ilhas e o número de arquipélagos. A chamada do *kernel* é realizada com a quantidade de arquipélagos (número de blocos) e a quantidade de ilhas (número de *threads* de cada bloco), como mostrado na Figura 28.

A classificação da população em ambas as versões é realizada com a ordenação crescente dos indivíduos.

A plataforma CUDA, assim como a linguagem Go, oferece métodos de ordenação para serem usadas dentro da placa; entretanto, essas funções são limitadas apenas para ordenar números inteiros. Então escolheu-se um algoritmo de ordenação para realizar a classificação da população. Dentre os vários algoritmos de ordenação (*e.g. mergesort*, *heapsort*, *quicksort*) adotou-se a ordenação bitônica paralela, visto na seção 2.10.

```
149 //Bitonic Sort
150 __device__ void BitonicSort(s_individuo * p, int passo, int limite) {
151     int indice = threadIdx.x + blockDim.x * blockIdx.x;
152     int elem = indice^passo;
153     if ((elem)>indice){
154         if ((indice&limite)==0 && p[indice].v>p[elem].v) {
155             s_individuo temp = p[indice];
156             p[indice] = p[elem];
157             p[elem] = temp;
158         }
159         if ((indice&limite)!=0 && p[indice].v<p[elem].v){
160             s_individuo temp = p[indice];
161             p[indice] = p[elem];
162             p[elem] = temp;
163         }
164     }
165 }
```

Figura 29 – Função *BitonicSort* para ordenação
Fonte: SINHA (2011)

Este método de ordenação, já comentado anteriormente, é baseado no trabalho desenvolvido por Ayushi Sinha do *Providence College* (SINHA, 2011). No seu trabalho, a autora apresenta três algoritmos de ordenação (*quick*, *radix*, *bitonic*) e relatou como melhor em desempenho a ordenação bitônica. Algumas modificações foram realizadas para que pudesse ser usada nas aplicações desenvolvida neste trabalho em CUDA.

4 ANÁLISE E RESULTADOS

Este capítulo tem a finalidade de descrever uma análise das implementações realizadas nas duas plataformas, bem como realizar uma visão geral dos problemas escolhidos, das ferramentas e uma comparação com relação à dificuldade de uso de cada tecnologia.

4.1 VISÃO GERAL DAS FERRAMENTAS

A linguagem *Go* foi lançada em 2009 e já começa a ganhar popularidade para a programação comum. Uma grande característica que difere das outras linguagens é o tratamento simples para a concorrência. Nos problemas implementados observou-se que a sincronização das *goroutines* é facilmente tratada através dos canais de comunicação (seja para sinalização ou para a passagem de dados) e o grau de paralelismo realizado também é facilmente controlado por meio da função *GOMAXPROCS*.

O objetivo de trabalhar o paralelismo com essa linguagem foi alcançado, entretanto, é necessário ressaltar que o campo de pesquisa com *Go* é bastante grande. Além de paralelismo, a linguagem tem suporte para serviços web; uma aplicação interessante seria a conexão concorrente de vários clientes em um servidor, realizando o processamento na arquitetura *multicore*.

No caso da plataforma *CUDA*, a GPU dispõe de uma grande quantidade de núcleos, muito maior que uma CPU clássica. O uso das placas gráficas é bastante interessante quando se quer desempenho com relação a otimizações de funções matemáticas e cálculos complexos.

CUDA apresenta uma abordagem bastante clara de manipulação e sincronização de *threads*. Sua arquitetura permite que a programação seja adaptável ao problema paralelizável através do uso de identificadores nos blocos e *threads* da GPU (*blockId, threadId*). A chamada do *kernel* com relação à configuração da dimensão da grade e blocos também se apresentou bastante flexível, ou seja, adaptável ao problema a ser resolvido.

Todo dado que é processado pela GPU, em sua memória local, é oriundo da memória principal do computador. Acesso a arquivos, a funções de bibliotecas

externas, a outros periféricos como placas de rede e disco rígido não são realizados por *CUDA*. Outro aspecto importante a se frisar, é o fato de que *CUDA* é operante apenas em placas criadas pela própria fabricante da *API*, a *NVIDIA*.

O paralelismo realizado no trabalho com esta plataforma operante em GPU também foi alcançado. Durante a implementação dos problemas, observou-se de que a *API* oferece tudo que é necessário para a programação dentro da placa, como por exemplo, geração de números randômicos e o cálculo de seno, cosseno e tangente. Uma das limitações identificadas foi a falta de suporte para recursividade. Essa característica só está disponível em placas CUDA mais recentes.

4.2 COMPARAÇÃO DAS IMPLEMENTAÇÕES

No cálculo da sequência de Fibonacci, identificou-se uma pequena vantagem com *Go* pelo fato da implementação ser mais natural nessa linguagem. Em *CUDA* foi preciso implementar uma versão não recursiva. O paralelismo neste caso se tornou pouco eficiente dada a natureza do problema. O objetivo, conforme já explicado, era comparar a programação nos dois casos.

Com o objetivo de calcular uma sequência inteira de Fibonacci, uma implementação sequencial possuiria um desempenho melhor, já que o valor do próximo número é calculado com os dois números já calculados anteriormente. Já citado anteriormente, a recursividade de Fibonacci não contém dependências de dados. Como *CUDA* não traz suporte à recursividade na placa o qual foi realizada a pesquisa, foi preciso rever a codificação. Para cada número foi preciso realizar um cálculo de maneira sequencial, apenas com objetivo de carregar vários núcleos de cálculo da GPU e para poder simular o mesmo efeito de pilhas.

Enquanto que em *Go*, o cálculo de, por exemplo, algumas dezenas de números é processado em varias *goroutines* divididas em alguns núcleos da CPU, em *CUDA* pode-se ter centenas de núcleos executando um numero maior de *threads* para calcular essas mesmas dezenas.

Na multiplicação de matrizes, *Go* também apresentou maior simplicidade, pois a implementação do problema com estrutura de matriz convencional em *CUDA* não foi estudada adequadamente, pois não se conseguiu uma matriz resultante com o resultado correto da multiplicação (verificado através de uma comparação com

uma multiplicação realizada sequencialmente). Após algumas tentativas sem sucesso de implementação, verificou-se a necessidade de utilizar outra abordagem para realizar a multiplicação de matrizes. A segunda abordagem, que foi a implementação com vetores para a solução do problema, tornou-se adequado para a situação, produzindo os resultados de maneira correta da multiplicação. Embora em ambos os casos, tanto as *goroutines* quanto as *threads* fiquem responsáveis por calcular o resultado de cada posição da matriz, *CUDA* pode realizar a tarefa muito mais rápido, pois opera sobre muito mais núcleos de processamento.

Do ponto de vista de processamento de tarefas temos: em uma matriz 100x100, *Go* cria cem *goroutines* (uma para cada linha) para obter a matriz resultante calculando em, por exemplo, oito núcleos, já em *CUDA* é possível processar cem *threads* em cem núcleos para calcular o resultado da multiplicação.

Os algoritmos genéticos em ambos os casos apresentaram-se de maneira bastante interessante. Não se encontrou muita dificuldade com relação à implementação da abordagem de usar *threads* e *goroutines* para controlar a população de uma única ilha, aplicando a metáfora de que as ilhas operam situações ambientais independentemente uma das outras. Em *CUDA* foi realizada outra abordagem que se beneficia da arquitetura apresentada pela plataforma, onde blocos da grade representavam as ilhas e as *threads* representavam os indivíduos. A metáfora da coexistência de pessoas em um mundo foi alcançado adotando essa segunda abordagem, onde cada *threads* realizava as tarefas sobre um cromossomo.

Ao comparar o algoritmo genético implementado nas duas ferramentas, o desempenho de *CUDA* é relativamente maior. Por exemplo, em um problema com cinquenta ilhas, pode-se processar as cinquenta *threads* em cinquenta núcleos da GPU. Já em *Go*, as mesmas cinquenta ilhas seriam processadas por cinquenta *goroutines* nos poucos núcleos da CPU.

Baseado em todas as análises feitas, tem-se que a vantagem da concorrência realizada pela linguagem *Go* não é limitada ao hardware, ou seja, se a CPU for *multicore* é possível realizar o paralelismo, e a implementação também é bastante simples, o mesmo não ocorre com a plataforma *CUDA*, que não opera em qualquer GPU, apenas aquelas produzidas pela indústria criadora da ferramenta, e sua implementação é simples dependendo do tipo de problema.

De modo geral, em ambas as ferramentas, o tratamento para a programação concorrente é facilmente adaptável. Para problemas que exigem cálculos extensos, é preferível a utilização de *CUDA*, pois opera em muito mais núcleos.

4.3 CARACTERÍSTCAS GERAIS

Os quadros a seguir especificam algumas características dos objetos de estudo que foram identificadas no decorrer da pesquisa.

Característica	Descrição
1. Plataforma operante	Qualquer CPU *multicore*
2. Softwares requeridos (no sistema operacional *Windows*)	Necessária instalação da distribuição binária de Go e do ambiente *LiteIDE*
3. Linguagem/Sintaxe	Linguagem de programação própria, muito semelhante a C; utiliza uma nova sintaxe para a programação
4. Funções para o paralelismo	Funções: *Goroutine*/*Channels*/*Gomaxprocs*
5. Uso de memória	Memória principal e memória *cache*.
6. Adaptação quanto programação	Necessária adaptação de sintaxe e de novas funções
7. Aplicabilidade	A linguagem é de propósito geral, projetada para substituir C++ em programação de sistemas.

Quadro 1 – Características gerais da linguagem Go
Fonte: Autoria própria.

De modo geral, a linguagem *Go*, como já foi citado nos capítulos anteriores, veio com propósito geral de ser uma nova linguagem rápida e eficiente para a programação de sistemas. Mesmo sendo uma linguagem muito parecida com C, novas abordagens foram identificadas (*e.g.* retorno duplo, orientação a objetos, paralelismo). Programadores habituados com C provavelmente necessitariam de tempo para a adaptação.

Característica	Descrição
1. Plataforma operante	GPU da NVIDIA capaz de operar *CUDA*
2. Softwares requeridos (no sistema operacional *Windows*)	Necessária instalação do *Toolkit* CUDA e ambiente *Eclipse* ou *Visual Studio 2012*
3. Linguagem/Sintaxe	C/C++, Fortran, Pyton; utiliza as linguagens já existentes para a programação.
4. Funções para o paralelismo	Funções *kernel* e identificadores de *threads*
5. Uso de memória	Necessário transferir os dados da memória principal para a memória da GPU e vice versa
6. Adaptação quanto à programação	Necessária adaptação de novas funções
7. Aplicabilidade	As GPUs são indicadas especialmente para processamento matemático, em que um problema possa ser particionado. Não oferecem recursos como acesso à rede ou acesso à arquivos.

Quadro 2 – Características gerais da plataforma *CUDA*
Fonte: Autoria própria.

A plataforma *CUDA* utiliza linguagens de programação como C/C++ para a programação. Programadores podem escolher entre as linguagens suportadas pela plataforma para explorar o paralelismo das placas gráficas.

5 CONCLUSÃO

Este capítulo apresenta as considerações finais do trabalho fazendo uma relação dos resultados obtidos com o objetivo geral e específicos.

Através das análises feitas, pode-se concluir que a utilização dessas duas ferramentas depende muito do tipo de problema. A vantagem clara que a GPU tem sobre a CPU é a quantidade de núcleos, logo, quanto mais núcleos a placa tiver, maior o poder de processamento e melhor vai ser o desempenho. Por outro lado, a vantagem da CPU é o seu relacionamento com os outros componentes do computador.

O objetivo geral do trabalho foi alcançado com sucesso. Com os experimentos e estudos realizados nas duas ferramentas foi possível identificar suas as características principais e as dificuldades de implementação. Com a análise através dos dois quadros produzidos é possível escolher qual das duas ferramentas o programador vai escolher caso tenha que trabalhar com o paralelismo.

Hoje é muito comum ter computadores com uma CPU *multicore*, podendo usar a linguagem *Go* para a programação, por outro lado, têm de ser feito um tempo a mais de adaptação com a linguagem.

Já o mesmo não acontece com *CUDA*, que utiliza alguma linguagem existente para a programação em GPU, sem falar no grande poder de processamento proporcionado pelo grande números de núcleos a mais comparada à CPU. Porém, *CUDA* trabalha apenas com placas gráficas específicas, e sua limitação de não poder acessar outros componentes do computador o torna incapaz de produzir solução de muitos problemas.

Os conceitos técnicos da programação paralela foram atingidos com as duas plataformas. A dificuldade de implementação das duas ferramentas é levado em consideração do problema e também do quão familiarizado o programador está com os ambientes de programação.

Quanto aos objetivos específicos do estudo da linguagem *Go* e da plataforma *CUDA* foi realizada através de pesquisas na literatura; essa pesquisa também permitiu a identificação de problemas que permitem a paralelização, como o Fibonacci recursivo, multiplicação de matrizes e algoritmo genéticos paralelos.

O objetivo da familiarização das duas ferramentas foi alcançado através da implementação de alguns problemas identificados no objetivo específico citado anteriormente.

Quanto ao último objetivo específico, a comparação da implementação dos objetos de estudo foi alcançada através da análise dos problemas solucionados com a concorrência, apresentando também os quadros que fazem um comparativo com das características de cada ferramenta.

As dificuldades encontradas não interferiram muito nos resultados desse trabalho. Para um programador acostumado com a linguagem C, a programação em *CUDA* não deve apresentar nenhuma dificuldade. O desafio está em entender como funciona a chamada do *kernel* e como configurar a grade e os blocos para determinado tipo de problema. A linguagem *Go* foi a ferramenta que houve mais programas implementados, justamente por causa da sua nova sintaxe, levando ao programador acostumado com a linguagem C estar mais atento durante a programação.

É possível concluir que esse trabalho foi finalizado com resultados satisfatórios. Os códigos incluídos têm por objetivo auxiliar o desenvolvimento de trabalhos futuros que envolvem desempenho de sistemas e serve também de iniciativa para o estudo aplicado da programação concorrente.

5.1 TRABALHOS FUTUROS

Com base nos resultados obtidos, outros problemas podem servir como escopo para trabalhos futuros. São eles:

- Implementação de algoritmos genéticos canônicos e algoritmos genéticos celulares com *CUDA*;
- Desenvolvimento de serviços web com *Go*;
- Estudo de métodos de ordenação paralelas;
- Estudos que envolvem otimizações com problemas inerentes ao paralelismo;
- Desenvolvimento de sistemas que aplicam as duas ferramentas.
- Programação paralela em múltiplas GPUs

A programação paralela na área computacional envolve vários problemas; sua importância se deve ao desempenho e a utilização da arquitetura *multicore*.

REFERÊNCIAS

ADLER, Dan. Genetic algorithms and simulated annealing: A marriage proposal. In: **Neural Networks, 1993., IEEE International Conference on**. IEEE, 1993. p. 1104-1109.

BORDIN, M. V.; RITTER, H.E. Sessão de Iniciação Científica – **Ferramentas de Programação Paralela para Arquiteturas Multicore.** Passo Fundo: ERAD, 2010. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2010/0025.pdf>. Acesso em: 8 set. 2013.

CAMARGO, C. A. S. **Uma Revisão de Ferramentas para Programação *Multithread* Baseadas no Modelo de Paralelismo de Tarefas.** Pelotas, 2012. 59 p. Dissertação (mestrado) - Universidade Federal de Pelotas. Departamento de Ciência da Computação. Pelotas, 2012. Disponível em: <http://inf.ufpel.edu.br/site/wp-content/uploads/2012/04/Uma-Revisão-de-Ferramentaspara-Programação-Multithread-Baseadas-no-Modelo-de-Paralelismo-de-Tarefas.pdf>. Acesso em 8 set. 2013.

COPPIN, Ben. **Inteligência artificial.** Rio de Janeiro, RJ: LTC, 2010.

DE LA ASUNCIÓN, M., et al. An MPI-CUDA implementation of an improved Roe method for two-layer shallow water systems. **Journal of Parallel and Distributed Computing**, v. 72, n. 9, p. 1065-1072, 2012.

DU, Peng et al. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. **Parallel Computing**, v. 38, n. 8, p. 391-407, 2012.

FENG, Wu-chun; XIAO, Shucai. To GPU synchronize or not GPU synchronize?. In: **Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on**. IEEE, 2010. p. 3801-3804.

FRANCO, Joaquín, et al. **A parallel implementation of the 2D wavelet transform using CUDA.** Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on. IEEE, 2009. Disponível em: <http://ditec.um.es/caps/pdfs/franco_pdp09.pdf>. Acesso em 12 dez. 2013.

GO AUTHORS. **The Go Programming Language**. 2010. Disponível em: <http://golang.org/doc/>. Acesso em: 12 out. 2013.

GOMES, Leandro AS; NEVES, Bruno S.; PINHO, Leonardo B. **CUDA x OpenMP x Pthreads: Implicações no custo total de uma solução de distribuição segura de vídeos.** ERAD, 2012.

GONDA, Luciano; MONGELLI, Henrique. **Uma Implementação de Algoritmos BSP/CGM para ordenação**. WSCAD, 2005.

GUIMARÃES, G. M.; OLIVEIRA, J. O. de C. **Comparação entre facilitadores para programação paralela em processadores multicore.** Ponta Grossa, 2014. Trabalho de Conclusão de Curso – Universidade Tecnológica Federal do Paraná.

HARVEY, Matt J.; DE FABRITIIS, Gianni. Swan: A tool for porting CUDA programs to OpenCL. **Computer Physics Communications**, v. 182, n. 4, p. 1093-1099, 2011.

HEDAR, Abdel-Rahman; FUKUSHIMA, Masao. Derivative-free filter simulated annealing method for constrained continuous global optimization. **Journal of Global Optimization**, v. 35, n. 4, p. 521-549, 2006.

HERLIHY, Maurice; SHAVIT, Nir. **The art of multiprocessor programming.** Burlington, MA: Elsevier Science, 2008. 508 p. ISBN 9780123705914.

HUGHES, Cameron; HUGHES, Tracey. **Professional multicore programming:** design and implementation for C++ developers. Indianapolis, IN: Wiley, 2008. 620 p. ISBN 9780470289624.

KIRK, D. B.; WEN-MEI, W. H.. **Programming massively parallel processors:** hands-on approach. Morgan Kaufmann. 2010.

KLÖCKNER, Andreas et al. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. **Parallel Computing**, v. 38, n. 3, p. 157-174, 2012.

KOSCIANSKI, A. **Modèles de Mécanique pour le Temps Réel**. Tese doutorado. INSA de Rouen, França, 2004.

LEE, Victor W. et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: **ACM SIGARCH Computer Architecture News**. ACM, 2010. p. 451-460.

NAWATA, Takehiko; SUDA, Reiji. APTCC: Auto Parallelizing Translator From C To CUDA. **Procedia Computer Science**, v. 4, p. 352-361, 2011.

NVIDIA. **GETTING STARTED GUIDE FOR MICROSOFT WINDOWS**: Installation and Verification on Windows. Disponível em: <http://docs.nvidia.com/cuda/pdf/CUDA_Getting_Started_Windows.pdf>. Acesso em: 01 dez. 2013.

NVIDIA. **CUDA C PROGRAMMING GUIDE:** Design Guide. 2013. Disponível em: <http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf>. Acesso em: 04 jan. 2014.

PESSINI, Evandro Carlos, et al. **Algoritmos Genéticos Paralelos:** Uma Implementação Distribuída Baseada em Javaspaces. Dissertação (mestrado) – Universidade Federal de Santa Catarina. Departamento de Ciência da Computação. Florianópolis, 2003. Disponível em: <https://repositorio.ufsc.br/xmlui/bitstream/handle/123456789/85977/238259.pdf?sequence=1>. Acesso em 2 jan. 2014.

PIKE, Rob. *et al*. **Go at Google:** Language Design in the Service of Software Engineering. Disponível em: <http://talks.golang.org/2012/splash.article>. Acesso em: 09 nov. 2013.

PIKE, Rob. **Go Concurrency Patterns.** Heroku's Waza conference. 2012. Disponível em: <http://talks.golang.org/2012/concurrency.slide>. Acesso em: 31 dez. 2013.

POSPÍCHAL, Petr; JAROS, Jiri. Gpu-based acceleration of the genetic algorithm. **GECCO competition**, 2009.

RUSSELL, Stuart J.; NORVIG, Peter. **Inteligência artificial.** Rio de Janeiro, RJ: Elsevier, 2004. 1021 p. ISBN 9788535211771.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer; GAGNE, Greg. **Fundamentos de sistemas operacionais.** 6. ed. Rio de Janeiro: LTC, 2004.

SINHA, Ayushi. **Sorting on CUDA,** Providence College, 2011. Disponível em: <http://digitalcommons.providence.edu/student_scholarship/7/>. Acesso em 26 mar. 2013.

SOEIRO, F. J. C. P; BECCENERI, J. C.; SILVA, N. A. J. Recozimento Simulado (Simulated Annealing). In: Antônio J. Silva Neto; José Carlos Becceneri. (Org). **Técnicas de Inteligência Computacional Inspiradas na Natureza - Aplicação em Problemas Inversos em Transferência Radiativa.** São Carlos: SBMAC - Sociedade Brasileira de Matemática Aplicada e Computacional, 2009, v.41, p.43-50.

SÖRENSEN, Kenneth. Metaheuristics—the metaphor exposed. **International Transactions in Operational Research**, 2012.

STRATTON, John A.; STONE, Sam S.; WEN-MEI, W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In: **Languages and Compilers for Parallel Computing**. Springer Berlin Heidelberg, 2008. p. 16-30.

SUCHARD, M. A., et al. **Understanding GPU programming for statistical computation:** Studies in massively parallel massive mixtures. Journal of Computational and Graphical Statistics 19.2 (2010). Disponível em: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2945379/>. Acesso em 12 dez. 2013.

TANENBAUM, Andrew S.; STEEN, Maarten van. **Sistemas distribuídos:** princípios e paradigmas. 2. ed. São Paulo, SP: Pearson Prentice Hall, 2007.

TANG, Peiyi. Multi-core parallel programming in go. In: **Proceedings of the First International Conference on Advanced Computing and Communications**. 2010. p. 64-69. Disponível em: <http://www.ualr.edu/pxtang/papers/ACC10.pdf>. Acesso em: 31 dez. 2013.

VAN DYK, Danny et al. HONEI: A collection of libraries for numerical computations targeting multiple processor architectures. **Computer Physics Communications**, v. 180, n. 12, p. 2534-2543, 2009.

WHITLEY, Darrell. **A genetic algorithm tutorial.** Statistics and computing 4.2 (1994): 65-85. Disponível em: <http://bipad.cmh.edu/ga_tutorial1994.pdf>. Acesso em 8 dez 2013.

XIAO, Shucai; FENG, Wu-chun. Inter-block GPU communication via fast barrier synchronization. In: **Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on**. IEEE, 2010. p. 1-12.

YANG, Chao-Tung; HUANG, Chih-Lin; LIN, Cheng-Fang. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters.**Computer physics communications**, v. 182, n. 1, p. 266-269, 2011.

APA

YANO, Luis Gustavo Abe. **Avaliação e comparação de desempenho utilizando tecnologia CUDA**. 2010. Tese de Doutorado. Universidade Estadual Paulista.

APÊNDICE A - Programa 1: Processando Goroutines

O código a seguir mostra o programa apresentado na seção 3.2.1:

```
package main
import (
    "fmt"
    "runtime"
    //"math/rand"
    //"time"
)
func main() {
    c := make(chan string)
    runtime.GOMAXPROCS(runtime.NumCPU())
    for i := 0; i <= 100000; i++ {
        go func(num int) {
            c <- fmt.Sprintln("Executando goroutines", num)
        }(i)
    }
    //Espera terminar
    for i := 0; i <= 100000; i++ {
        fmt.Print(<-c)
    }
}
```

APÊNDICE B - Programa 2: Fibonacci Recursivo em Go

O código a seguir mostra o programa apresentado na seção 3.2.2:

```
package main
import (
    "fmt"
    "runtime"
    "time"
)
func fibonacci(num int) int {
    if num == 1 || num == 2 {
        return 1
    } else {
        return fibonacci(num-1) + fibonacci(num-2)
    }
}
func main() {
    qtdNum := 40 //qtd de números
    c := make(chan int)
    runtime.GOMAXPROCS(runtime.NumCPU())
    start := time.Now()
    //paralelo
    for i := 0; i < qtdNum; i++ {
        go func(n int) {
            c <- fibonacci(n + 1)
        }(i)
    }
    //espera terminar
    for i := 0; i < qtdNum; i++ {
        fmt.Println(<-c) //imprime
    }
    /*
        //sequencial
        for i:=0; i < qtdNum; i++{
            fmt.Println(fibonacci(i+1))
        }
    */
    fmt.Println(time.Since(start)) //tempo que leva para calcular o fibonacci
    fmt.Println()
}
```

APÊNDICE C - Programa 3: Multiplicação de Matrizes em Go

O código a seguir mostra o programa apresentado na seção 3.2.3:

```
package main
import (
    "fmt"
    "math/rand"
    "runtime"
    "time"
)
var lines = 100
var columns = 100
func main() {
    rand.Seed(time.Now().UnixNano())
    runtime.GOMAXPROCS(runtime.NumCPU())
    matrix1 := make([][]float64, lines)
    for i := 0; i < lines; i++ {
        matrix1[i] = make([]float64, columns)
    }
    matrix2 := make([][]float64, lines)
    for i := 0; i < lines; i++ {
        matrix2[i] = make([]float64, columns)
    }
    matrix3 := make([][]float64, lines)
    for i := 0; i < lines; i++ {
        matrix3[i] = make([]float64, columns)
    }
    for i := 0; i < lines; i++ {
        for j := 0; j < columns; j++ {
            matrix1[i][j] = rand.Float64()
            matrix2[i][j] = rand.Float64()
        }
    }
    canal1 := make(chan bool)
    start := time.Now()
    /*sequencial
    for l := 0; l < lines; l++ {
        for c := 0; c < columns; c++ {
            for i := 0; i < columns; i++ {
                matrix3[l][c] += matrix1[l][i] * matrix2[i][c]
            }
        }
    }*/
    for l := 0; l < lines; l++ {
        go func(line int) {
            for c := 0; c < columns; c++ {
                for i := 0; i < columns; i++ {
                    matrix3[line][c] += matrix1[line][i] * matrix2[i][c]
                }
            }
            canal1 <- true
        }(l)
    }
    <-canal1
    fmt.Println(time.Since(start))
    for i := 0; i < lines; i++ {
```

```
        for j := 0; j < columns; j++ {
            fmt.Printf("%f ", matrix3[i][j])
        }
        fmt.Println("")
    }
    //fmt.Println(matrix3)
}
```

APÊNDICE D - Programa 4: Algoritmo Genético em Go

O código a seguir mostra o programa apresentado na seção 3.2.4:

```
package main
import (
    "fmt"
    "math/rand"
    "runtime"
    "sort"
    "time"
    //"bufio"
)
type gene struct {
    x float64 //posicao x
    y float64 //posicao y
    z float64 //valor da funcao
}
const QThreads = 2
const Quantos = 10
const Quantos2 = 2 * Quantos
type Tpop [Quantos]gene
var Gpop1 Tpop //populacao
var qtdInalada float64
var c chan int
const NCPU = 4
//------------------------------
// Methods required by sort.Interface.
func (s Tpop) Len() int {
    return Quantos
}
func (s Tpop) Less(i, j int) bool {
    return s[i].z < s[j].z
}
func (s Tpop) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}
//------------------------------------------
// esta funcao recebe um gene, aplica mutacao
//  e devolve um novo gene
func gasMutante(individuo gene) gene {
    //qtdInadala um pequeno valor no intervalo de 0 ~ 0.5
    individuo.x *= (1.0 - qtdInalada) + (2 * qtdInalada * rand.Float64())
    individuo.y *= (1.0 - qtdInalada) + (2 * qtdInalada * rand.Float64())
    return individuo
}
//------------------------------------------
// esta funcao aplica mutacao a varios genes,
//  ou seja uma populacao inteira.
// depois cada gene eh avaliado e a populacao
//  e organizada, do pior ate o melhor.
func (apop *Tpop) chernobyl(sizepop int) {
    for i := 0; i < sizepop; i++ {
        apop[i] = gasMutante(apop[i])
        apop[i].z = apop[i].x*apop[i].x + apop[i].y*apop[i].y
    }
    sort.Sort(apop)
```

```
}
//----------------------------------------
func main() {
    var aux gene
    rand.Seed(time.Now().UnixNano())
    runtime.GOMAXPROCS(NCPU)
    qtdInadala = 0.1
    var x1 gene
    x1.x = 10
    x1.y = 10
    x1.z = 100
    for i := 0; i < Quantos; i++ {
        Gpop1[i] = x1
    }
    start := time.Now()
    counter := 0
    for Gpop1[0].v > 0.0000001 {
        Gpop1.chernobyl(Quantos)
        counter++
    }
    fmt.Printf("Ilha 1 - %d) %f %f %f \n", counter, Gpop1[0].x, Gpop1[0].y, Gpop1[0].v)
    fmt.Println(time.Since(start))
}
```

APÊNDICE E - Programa 5: Algoritmo Genético no Modelo das Ilhas implementado em Go

O código a seguir mostra o programa apresentado na seção 3.2.5:

```
package main
import (
    "fmt"
    "math"
    "math/rand"
    "sort"
    "time"
    "runtime"
)
const nIndv = 148;
const nIlhas = 32;
//--------Griewank----------//
//contantes griewank' function
const n = 2
const fr = 4000
//------------------------//
type cromossomo struct {
    x[2] float64
    y float64
}
type ilha struct{
    crom[nIndv] cromossomo
}
type Tpop [nIndv]cromossomo
type TIlha [nIlhas]ilha //ilhas
type TPopT [nIlhas*nIndv]cromossomo //população total do mundo
var Ilha TIlha
var PopT Tpop
//-----------------------------------------//
// Methods required by sort.Interface.
func (s Tpop) Len() int {
    return nIndv
}
func (s Tpop) Less(i, j int) bool {
    return s[i].y < s[j].y
}
func (s Tpop) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}
//-----------------------------------------//
func griewank(individuo cromossomo) cromossomo{
    var sum, prod float64
    sum = 0
    prod = 1
    for i := 0; i<n; i++{
        sum += individuo.x[i]*individuo.x[i]
        prod *= math.Cos(individuo.x[i]/math.Sqrt(float64(i+1)))
    }
    individuo.y = (sum/fr)-prod+1
    return individuo
}
func alteraGene (individuo cromossomo, alpha float64) cromossomo{
    individuo.x[0] += alpha * (2.0 * rand.Float64() - 1.0)
```

```
    individuo.x[1] += alpha * (2.0 * rand.Float64() - 1.0)
    return individuo
}
func (ilha *TIlha) genetico (alpha float64){
    canal := make(chan bool)
    for i := 0; i < nIlhas; i++ {
        go func (numIlha int){
            for j:= 0; j < nIndv; j++{
                ilha[numIlha].crom[j] = alteraGene(ilha[numIlha].crom[j], alpha)
                ilha[numIlha].crom[j] = griewank(ilha[numIlha].crom[j])
            }
            sort.Sort(Tpop(ilha[numIlha].crom))
            canal <- true
        }(i)
    }
    <-canal
    /*
    //sequencial
    for numIlha := 0; numIlha < nIlhas; numIlha++ {
        for j:= 0; j < nIndv; j++{
            ilha[numIlha].crom[j] = alteraGene(ilha[numIlha].crom[j], alpha)
            ilha[numIlha].crom[j] = griewank(ilha[numIlha].crom[j])
        }
        sort.Sort(Tpop(ilha[numIlha].crom))
    }
    */
}
func main(){
    runtime.GOMAXPROCS(runtime.NumCPU())
    rand.Seed(time.Now().UnixNano())
    var P_old cromossomo
    alpha := 1.0
    loops := 0
    for i:=0; i<nIlhas; i++{
        for j:=0; j<nIndv; j++{
            Ilha[i].crom[j].x[0] = rand.Float64()
            Ilha[i].crom[j].x[1] = rand.Float64()
            Ilha[i].crom[j].y = rand.Float64()
        }
    }
    P_old.y = PopT[0].y
    start := time.Now()
    for loops < 100 || P_old.y - PopT[0].y > 0.01{
        Ilha.genetico(alpha)
        //------ reunir os nIlhas/2 melhores-------//
        for i:=0; i < nIlhas; i++{
            for j:=0; j<int(math.Sqrt(nIndv)); j++{
                PopT[i] = Ilha[i].crom[j];
            }
        }
        sort.Sort(PopT)
        for i:=0; i<nIlhas; i++{
            for j:=0; j<nIlhas; j++{
                Ilha[i].crom[j] = PopT[j]
            }
```

```
        }
        fmt.Printf("%d) x1 = %f    x2 = %f    y = %f \n", loops, PopT[0].x[0], PopT[0].x[1],
PopT[0].y)//melhor da rodada
        alpha = 0.95 * alpha;
        loops++
    }
    fmt.Println(time.Since(start))
}
```

APÊNDICE F - Programa 6: Fibonacci em CUDA

O código a seguir mostra o programa apresentado na seção 3.3.1:

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <stdio.h>

const int qtdNum = 40;

__device__ int fibonacci(int num){

    if (num == 0 || num == 1){
        return 1;
    }else{
        int aux, a = 0, b = 1;
        for (int i = 0; i < num; ++i)
        {
            aux = a+b;
            a = b;
            b = aux;
        }
        return aux;
    }
}

__global__ void kernel(int *d_sequenciaFib){
    int i = threadIdx.x;

    d_sequenciaFib[i] = fibonacci(i);

    __syncthreads();
}

int main()
{

    int h_sequenciaFib[qtdNum];
    int *d_sequenciaFib;

    cudaMalloc((void**)&d_sequenciaFib, qtdNum*sizeof(int));

    kernel<<<1, qtdNum>>>(d_sequenciaFib);

    cudaMemcpy(h_sequenciaFib, d_sequenciaFib, qtdNum*sizeof(int),
cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();

    for (int i = 0; i < qtdNum; ++i)
    {
        printf("%d\n", h_sequenciaFib[i]);
    }

    cudaDeviceReset();

    return 0;
}
```

APÊNDICE G - Programa 7: Multiplicação de Matrizes em CUDA

O código a seguir mostra o programa apresentado na seção 3.3.2:

```
#include <iostream>
#define tamBloco 16

__global__ void matrixMult(float *A, float *B, float *C, int N)
{
	//Multiplicação de matrix C=A*B de tamanho NxN
	//Cada thread computa um único elemento da matrix C
	int linha = blockIdx.y*blockDim.y + threadIdx.y;
	int coluna = blockIdx.x*blockDim.x + threadIdx.x;

	float sum = 0.f;
	for (int n = 0; n < N; ++n)
	   sum += A[linha*N+n]*B[n*N+coluna];

	C[linha*N+coluna] = sum;

	__syncthreads();
}

int main()
{

	int N,K;
	K = 10;
	N = K*tamBloco;

	//Criando matrizes no host
	float *hA,*hB,*hC;
	hA = new float[N*N];
	hB = new float[N*N];
	hC = new float[N*N];

	//Iniciando matrizes
	for (int j=0; j<N; j++){
	   for (int i=0; i<N; i++){
		hA[j*N+i] = 2.f*(j+i);
			hB[j*N+i] = 1.f*(j-i);
	   }
	}

	//Criando as matrizes no device
	float *dA,*dB,*dC;
	cudaMalloc((void**)&dA, N*N*sizeof(float));
	cudaMalloc((void**)&dB, N*N*sizeof(float));
	cudaMalloc((void**)&dC, N*N*sizeof(float));


	dim3 threadBlock(tamBloco,tamBloco);//threads = 16x16
	dim3 grid(K,K);//blocos = KxK

	//Copiando as matrizes do host para o device
	cudaMemcpy(dA,hA,size,cudaMemcpyHostToDevice);
	cudaMemcpy(dB,hB,size,cudaMemcpyHostToDevice);
```

```
        //execução do kernel

        matrixMult<<<grid,threadBlock>>>(dA,dB,dC,N);

        //Copia o resultado do device para cpu
        cudaMemcpy(hC,dC, N*N*sizeof(float),cudaMemcpyDeviceToHost);

        cudaDeviceSynchronize();

    cudaDeviceReset();

        return 0;

}
```

APÊNDICE H - Programa 8: Algoritmo Genético no Modelo das Ilhas implementado em CUDA Versão 1

O código a seguir mostra a versão um do programa apresentado na seção 3.3.3:

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <stdio.h>
#include <curand_kernel.h>

//potência de 2
const int nIndv = 64;
const int nIlhas = 16;

//--------------Griewank-------------//
//constantes griewank's function
__device__ double n = 2;
__device__ double fr = 4000;
//----------------------------//

__device__ curandState s;

struct cromossomo{
   double x[2], y;
};

struct ilha{
   cromossomo crom[nIndv];
};

ilha *Ilha = new ilha[nIlhas]; // nIlhas
cromossomo *PopT = new cromossomo[nIndv*nIlhas];// população total do mundo

//funções
__device__ void BitonicSort(cromossomo * p, int passo, int limite);
__device__ double griewank (double *x);
__global__ void geraPopulacao(ilha *p);
__device__ void ordena(cromossomo *p, int tam);
__global__ void genetico(ilha *p, double aleatorio);
__global__ void classificaIlha(ilha *ilha);
__global__ void classificaMundo(ilha *ilha, cromossomo *p);
//----------------------------//

int main()
{
   ilha *d_Ilha;

   cromossomo P_old;
   cromossomo *d_PopT;

   int loops = 0;
   double alpha = 1.0;

   //----------------- Aloca memória na placa--------------------//
   cudaMalloc((void**)&d_Ilha, nIlhas*sizeof(ilha));
   cudaMalloc((void**)&d_PopT, (nIndv*nIlhas)*sizeof(cromossomo));
   //-----------------------------------------------------------//
```

```
    //-------Cria a primeira população--------//

    geraPopulacao<<<nIlhas, nIndv>>>(d_Ilha);

    //----------------------------------------//
    cudaDeviceSynchronize();

    P_old.y = PopT[0].y - 1;



    while(loops < 100 || ((P_old.y - PopT[0].y) > 0.01))
    {
        P_old.y = PopT[0].y;

        //------------genético aqui--------------//

        genetico<<<nIlhas, nIndv>>>(d_Ilha, alpha);

        //----------------------------------------//

        cudaDeviceSynchronize();

        //------------classificação 1------------//

        classificaIlha<<<1, nIlhas>>>(d_Ilha);

        //----------------------------------------//

        cudaDeviceSynchronize();

        //------------classificação 2 para migração------------//

        classificaMundo<<<1, nIndv*nIlhas>>>(d_Ilha, d_PopT);

        //----------------------------------------//

        cudaDeviceSynchronize();

        cudaMemcpy(PopT, d_PopT, (nIndv*nIlhas)*sizeof(cromossomo),
cudaMemcpyDeviceToHost);//copia para cpu
        cudaMemcpy(Ilha, d_Ilha, nIlhas*sizeof(ilha), cudaMemcpyDeviceToHost);//copia para
cpu

        //migracao
        for (int i = 0; i < nIlhas; ++i)//os 'nIlhas' melhores e vão ser jogadas para cada ilha
        {
            for (int j = 0; j < nIlhas; ++j)
            {
                Ilha[i].crom[j] = PopT[j];
            }
        }
```

```
        cudaMemcpy(d_Ilha, Ilha, nIlhas*sizeof(ilha), cudaMemcpyHostToDevice);//copia para
gpu

        printf("%d: x1= %.9f    x2= %.9f    y=%.9f\n", loops, PopT[0].x[0], PopT[0].x[1],
PopT[0].y);//imprime o melhor da rodada

        alpha = 0.956 * alpha;
        loops++;
    }


    cudaDeviceSynchronize();
    cudaDeviceReset();
    return 0;
}

__global__ void geraPopulacao(ilha *ilha){
    for (int i = 0; i < nIlhas; ++i)
    {
        for (int j = 0; j < nIndv; ++j)
        {
            ilha[i].crom[j].x[0] = curand_uniform(&s);
            ilha[i].crom[j].x[1] = curand_uniform(&s);
            ilha[i].crom[j].y = curand_uniform(&s);
        }
    }

    for (int k = 0; k < nIlhas; ++k)
    {
        ordena(ilha[k].crom, nIndv);
    }
}

__global__ void genetico(ilha *ilha, double alpha){
    int j = threadIdx.x;
    for (int i = blockIdx.x; i < nIlhas; ++i)
    {
        ilha[i].crom[j].x[0] += alpha * (2.0 * curand_uniform(&s) - 1.0);
        ilha[i].crom[j].x[1] += alpha * (2.0 * curand_uniform(&s) - 1.0);
        ilha[i].crom[j].y = griewank(ilha[i].crom[j].x);
        __syncthreads();
    }

    __syncthreads();
}

__global__ void classificaIlha(ilha *ilha){
    int i = threadIdx.x;

    ordena(ilha[i].crom, nIndv);

    __syncthreads();
}

__global__ void classificaMundo(ilha *ilha, cromossomo *p){
```

```
    //---------reunir todas populações em p---------//
    int count = 0;
    for (int i = 0; i < nIlhas; ++i)
    {
        for (int j = 0; j < nIndv; ++j)
        {
            p[count] = ilha[i].crom[j];
            count++;
        }
    }
    //-----------------------------------------------//
    ordena(p, nIlhas*nIndv);
    __syncthreads();
}

__device__ double griewank (double *x){
    double sum = 0;
    double prod = 1;
    for (int j = 0; j < n; ++j)
    {
        sum += x[j]*x[j];
        prod *= cosf(x[j]/sqrtf(j+1));
    }
    return (sum/fr)-prod+1;
}

__device__ void ordena(cromossomo *p, int tam){
    for (int limite = 2; limite <= tam; limite=limite*2)
    {
        for (int passo = limite/2; passo > 0; passo=passo/2)
        {
            BitonicSort(p, passo, limite);
        }
    }
}

//Bitonic Sort
__device__ void BitonicSort(cromossomo * p, int passo, int limite) {
    int indice = threadIdx.x + blockDim.x * blockIdx.x;
    int elem = indice^passo;
    if ((elem)>indice){
        if ((indice&limite)==0 && p[indice].y>p[elem].y) {
            cromossomo temp = p[indice];
            p[indice] = p[elem];
            p[elem] = temp;
        }
        if ((indice&limite)!=0 && p[indice].y<p[elem].y){
            cromossomo temp = p[indice];
            p[indice] = p[elem];
            p[elem] = temp;
        }
    }
    __syncthreads();
}
```

APÊNDICE I - Programa 9: Algoritmo Genético no Modelo das Ilhas implementado em CUDA Versão 2

O código a seguir mostra a versão dois do programa apresentado na seção 3.3.3:

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <stdio.h>
#include <curand_kernel.h>

//Cada arquipélago possui várias ilhas, cada ilha possui vários indivíduos
//Cada bloco possui várias threads, cada thread controla vários indivíduos
//O conjunto de arquipélago = mundo

const int nArquipelago = 16; //blocos = arquipélagos
const int nIlhas = 16; //threads = ilhas por arquipélago
const int nIndv = 32; //número de indivíduo de cada ilha
const int populacaoTotal = nArquipelago*nIlhas*nIndv;

//--------------Griewank-------------//
//constantes griewank's function
__device__ double n = 2;
__device__ double fr = 4000;
//----------------------------//

__device__ curandState s;

struct cromossomo{
   double x[2], y;
};

struct ilha{
   cromossomo crom[nIndv];
};


ilha *World = new ilha[nArquipelago * nIlhas];
cromossomo *PopT = new cromossomo[populacaoTotal];

//funções
__device__ void BitonicSort(cromossomo *p, int passo, int limite);
__device__ void Sort(cromossomo *p, int ehTotal);
__device__ double griewank (double *x);
__global__ void geraPopulacao(ilha *ilha);
__device__ void ordena(cromossomo *p, int tam);
__global__ void genetico(ilha *p, double aleatorio);
__global__ void classificaIlha(ilha *ilha);
__global__ void classificaMundo(ilha *ilha, cromossomo *p);
//--------------------------//


int main()
{
   ilha *d_World;

   cromossomo P_old;
   cromossomo *d_PopT;
```

```
int loops = 0;
double alpha = 1.0;

//----------------- Aloca memória na placa--------------------//
cudaMalloc((void**)&d_World, nArquipelago*nIlhas*sizeof(ilha));
cudaMalloc((void**)&d_PopT, populacaoTotal*sizeof(cromossomo));
//-----------------------------------------------------------//

//-------Cria a primeira população--------//

geraPopulacao<<<nArquipelago, nIlhas>>>(d_World);

//---------------------------------------//

cudaDeviceSynchronize();

P_old.y = PopT[0].y - 1;


while(loops < 100 || (P_old.y - PopT[0].y > 0.01))
{
    P_old.y = PopT[0].y;

    //------------genético aqui--------------//

    genetico<<<nArquipelago, nIlhas>>>(d_World, alpha);

    //---------------------------------------//

    cudaDeviceSynchronize();

    //----------classificação 1--------------//

    classificaIlha<<<1, nIlhas>>>(d_World);

    //---------------------------------------//

    cudaDeviceSynchronize();

    //---------classificação 2 para migração--------------//

    classificaMundo<<<1, nIndv*nIlhas>>>(d_World, d_PopT);

    //-------------------------------------//

    cudaDeviceSynchronize();


    cudaMemcpy(World, d_World, nArquipelago*nIlhas*sizeof(ilha),
cudaMemcpyDeviceToHost);//copia para cpu
    cudaMemcpy(PopT, d_PopT, populacaoTotal*sizeof(cromossomo),
cudaMemcpyDeviceToHost);//copia para cpu

    //micração
```

```
        for (int i = 0; i < nArquipelago*nIlhas; ++i)
        {
            for (int j = 0; j < nIlhas; ++j)//os 'nIlhas' melhores e vão ser jogadas para cada ilha
            {
                World[i].crom[j] = PopT[j];
            }
        }

        cudaMemcpy(d_World, World, nArquipelago*nIlhas*sizeof(ilha),
cudaMemcpyHostToDevice);//copia para gpu


        printf("%d: x1= %.9f    x2= %.9f    y=%.9f\n", loops, PopT[0].x[0], PopT[0].x[1],
PopT[0].y);//imprime o melhor da rodada


        alpha = 0.95 * alpha;
        loops++;
    }


    cudaDeviceSynchronize();

    cudaDeviceReset();

    return 0;
}

__global__ void geraPopulacao(ilha *ilha){
    for (int i = 0; i < nIlhas*nArquipelago; ++i)
    {
        for (int j = 0; j < nIndv; ++j)
        {
            ilha[i].crom[j].x[0] = curand_uniform(&s);
            ilha[i].crom[j].x[1] = curand_uniform(&s);
            ilha[i].crom[j].y = curand_uniform(&s);
        }
    }

    for (int k = 0; k < nIlhas*nArquipelago; ++k)
    {
        ordena(ilha[k].crom, 0);
        //Sort(ilha[k].crom, 0);
    }
}

__global__ void genetico(ilha *ilha, double alpha){
    int numeroIlha = threadIdx.x + blockIdx.x*blockDim.x;
    for (int i = 0; i < nIndv; ++i)
    {
        ilha[numeroIlha].crom[i].x[0] += alpha * (2.0 * curand_uniform(&s) - 1.0);
        ilha[numeroIlha].crom[i].x[1] += alpha * (2.0 * curand_uniform(&s) - 1.0);
        ilha[numeroIlha].crom[i].y = griewank(ilha[numeroIlha].crom[i].x);
    }
    __syncthreads();
```

```
}

__global__ void classificaIlha(ilha *ilha){
    int i = threadIdx.x;

    ordena(ilha[i].crom, nIndv);
    //Sort(ilha[i].crom, nIndv);

    __syncthreads();
}

__global__ void classificaMundo(ilha *ilha, cromossomo *p){
    //---------reunir todas populações no p---------//
    int count = 0;
    for (int i = 0; i < nArquipelago*nIlhas; ++i)
    {
        for (int j = 0; j < nIndv; ++j)
        {
            p[count] = ilha[i].crom[j];
            count++;
        }
    }
    //-----------------------------------------------------//

    ordena(p, nArquipelago*nIlhas*nIndv);

    __syncthreads();
}

__device__ double griewank(double *x){
    double sum = 0;
    double prod = 1;
    for (int j = 0; j < n; ++j)
    {
        sum += x[j]*x[j];
        prod *= cosf(x[j]/sqrtf(j+1));
    }
    return (sum/fr)-prod+1;

    //cod matlab
    /*for j = 1:n; s = s+x(j)^2; end
    for j = 1:n; p = p*cos(x/sqrt(j)); end
    y = s/fr-p+1;*/
}

__device__ void ordena(cromossomo *p, int tam){

    for (int limite = 2; limite <= tam; limite=limite*2)
    {
        for (int passo = limite/2; passo > 0; passo=passo/2)
        {
            BitonicSort(p, passo, limite);
        }
    }
}
```

```
//Bitonic Sort
__device__ void BitonicSort(cromossomo *p, int passo, int limite) {
    int indice = threadIdx.x + blockDim.x * blockIdx.x;
    int elem = indice^passo;
    if ((elem)>indice){
        if ((indice&limite)==0 && p[indice].y>p[elem].y) {
            cromossomo temp = p[indice];
            p[indice] = p[elem];
            p[elem] = temp;
        }
        if ((indice&limite)!=0 && p[indice].y<p[elem].y){
            cromossomo temp = p[indice];
            p[indice] = p[elem];
            p[elem] = temp;
        }
    }
    __syncthreads();
}


__device__ void Sort(cromossomo *p, int ehTotal){
    double aux;
    if (ehTotal == 0) //ordena uma ilha
    {
        for (int i = 0; i < nIndv; ++i)
        {
            for (int j = 0; j < nIndv-1; ++j)
            {
                if(p[j].y > p[j+1].y)
                {
                    aux = p[j].y;
                    p[j].y = p[j+1].y;
                    p[j+1].y = aux;
                }
            }
        }
    }else{
        for (int i = 0; i < populacaoTotal; ++i)
        {
            for (int j = 0; j < populacaoTotal-1; ++j)
            {
                if(p[j].y > p[j+1].y)
                {
                    aux = p[j].y;
                    p[j].y = p[j+1].y;
                    p[j+1].y = aux;
                }
            }
        }
    }
}
```