UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

LEONARDO DA LUZ DORNELES

Comparando diferentes implementações paralelas de algoritmos genéticos em GPUs com CUDA

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Márcio Dorn
Co-orientador: MSc. Mateus Boiani

Porto Alegre
2021

AGRADECIMENTOS

Agradeço aos meus pais e meus avós que sempre me deram estrutura e suporte para que eu estudasse; aos meus orientadores que através da sua tutoria facilitaram muito o desenvolvimento deste trabalho; e aos pesquisadores e anônimos que distribuem conteúdo de qualidade gratuitamente na internet.

RESUMO

O advento das GPUs para computação de propósito geral ampliou as possibilidades de utilização dessas arquiteturas para além da aceleração de aplicações gráficas. O campo de estudo de algoritmos de otimização tem especial interesse nessas arquiteturas, pois a aceleração desses algoritmos pode impactar significativamente na qualidade das soluções obtidas. Desde o início da última década, uma classe das técnicas de otimização que vem sendo adaptada com sucesso em GPUs é a dos Algoritmos Genéticos. Este trabalho apresenta duas formas distintas de se paralelizar algoritmos genéticos em GPUs com CUDA. Uma utiliza a memória compartilhada da GPU para obter cooperação na computação paralela enquanto a outra utiliza as funções de *Warp-Shuffle* da CUDA. Buscando avaliar cada uma dessas implementações em termos de desempenho, são realizados diversos experimentos variando os parâmetros de execução da aplicação (i.e granularidade do paralelismo, tamanho da população e número de dimensões do problema).

Palavras-chave: Computação evolutiva. algoritmos genéticos. GPU. CUDA. metaheurísticas. processamento massivamente paralelo.

Comparing different parallel implementations of genetic algorithms in GPUs on CUDA

ABSTRACT

The advent of GPUs for general-purpose computing has broadened the possibilities of the use of these architectures beyond the traditional acceleration of graphics-based applications. The research field of optimization algorithms has a special interest in these architectures because the acceleration of these algorithms may improve significantly the quality of the obtained solutions. Since the beginning of the last decade, one of the classes of optimization techniques that have been successfully adapted in the GPUs is that of Genetic Algorithms. This work presents two manners of implementing genetic algorithms in GPU using CUDA that exploit different levels of parallelism. In one of them, the cooperation of the parallel computation is achieved through the shared memory of the GPU, whereas the other one uses the CUDA Warp Shuffle functions. To evaluate the performance of these implementations, different experiments that change the execution parameters of the application (i.e the granularity, the population size, and the number of dimensions of the problem) are made.

Keywords: evolutionary computation, genetic algorithms, CUDA, GPU, metaheuristics, massively parallel computing.

LISTA DE ABREVIATURAS E SIGLAS

AEs	Algoritmos Evolutivos
AGs	Algoritmos Genéticos
API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
HC	Hill-Climbing
HPC	High Performance Computing
OEPs	Otimização por Enxame de Partículas
SA	Simulated Annealing
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor
SPMD	Single Program Multiple Data
PSL	Protein Sequence Length
TS	Pesquisa Tabu

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Muitos problemas práticos de otimização são problemas que se tem pouca ou nenhuma informação sobre as características da função que se quer otimizar. Além disso, esses problemas frequentemente pertencem a classe dos problemas NP-Difícil (TARDOS, 2005, p. 451-452), podendo ter um espaço de busca grande e tornando-se impossível de se utilizar um método exato de otimização em tempo hábil com os recursos computacionais disponíveis. Nesses casos, uma alternativa é encontrar uma solução aproximada que seja boa o suficiente para o problema em questão. O conjunto de procedimentos que definem maneiras eficientes de fazer essa busca é chamado de Meta-heurísticas (TALBI, 2009).

As meta-heurísticas definem métodos de exploração que podem ser aplicados para uma variedade de problemas, desde que seja possível avaliar qualitativamente suas soluções e compará-las. As meta-heurísticas podem ser classificadas em *Meta-heurísticas com Solução Única* ou *Meta-heurísticas Baseadas em População* (TALBI, 2009, p. 24-25). A primeira consiste em explorar o espaço de busca aplicando transformações em uma solução corrente, mantendo apenas essa solução. A segunda mantém um conjunto de soluções e tentam combiná-las para formar soluções melhores.

Uma classe de algoritmos proeminente pertencente às meta-heurísticas baseadas em população são os chamados *Algoritmos Evolutivos* (AEs) (BARTZ-BEIELSTEIN et al., 2014). AEs utilizam técnicas inspiradas em conceitos biológicos, como a teoria da seleção natural de Darwin (DARWIN et al., 1872), reprodução sexuada e mutação genética. O AE mais popular encontrado na literatura são os *Algoritmos Genéticos* (AGs) (HOLLAND, 1992). Eles foram aplicados com sucesso em diferentes classes de problemas oriundos da indústria e academia (TANG et al., 1996).

Os métodos baseado em população, como os AGs, se adaptam naturalmente às técnicas de paralelização de algoritmos, pois exploram diversas regiões do espaço de busca concorrentemente (CALÉGARI, 1999). Com o progresso tecnológico das arquiteturas paralelas e da computação heterogênea nos últimos anos, pesquisas de como adaptar os AGs eficientemente a essas arquiteturas se torna cada vez mais relevante para obtenção de soluções em tempo reduzido de problemas com grande interesse científico e econômico. Diversos modelos foram propostos, desde arquiteturas paralelas com memória distribuída, como *clusters* de HPC (*High Performance Computing*), que envolvem esquemas de troca de mensagem, até arquiteturas de domínio específico capazes de paralelizar a aplicação, como FPGAs (*Field Programmable Gate Array*) e GPUs (*Graphical Proces-*

sing Unit) (TALBI, 2015).

Durante muitos anos as GPUs foram utilizadas exclusivamente para acelerar aplicações gráficas. Porém, os rumos da utilização das placas gráficas na computação mudaram com o lançamento de arquiteturas de GPU completas para computação de propósito geral e da plataforma CUDA pela NVIDIA (NVIDIA, 2021b), que permitiu a programação de GPUs com uma linguagem que estende C/C++, além de outras linguagens de programação populares. Desde então, as GPUs vem tomando a atenção da comunidade científica por possibilitar a computação de grandes volumes de dados eficientemente e, ao mesmo tempo, serem dispositivos acessíveis, estando presentes em laptops, desktops e *clusters*.

Com a emergência das GPUs para computação de propósito geral, diversos tipos de problemas relacionados à computação científica foram adaptados para GPUs de forma eficiente (CHE et al., 2008; GARLAND et al., 2008). Com o sucesso do uso dessas arquiteturas para computação de propósito geral, diferentes estudos sobre a adaptação de algoritmos genéticos em GPUs começaram a surgir ao longo da última década (CHENG; GEN, 2019). Portanto, é um campo de pesquisa relativamente recente e com muito potencial para aceleração desses algoritmos. Além disso, tanto os modelos de programação para GPU quanto as suas arquiteturas estão em constante desenvolvimento.

1.1 Motivação

Algoritmos Genéticos foram aplicados com sucesso em diversos tipos de problemas de otimização com alta dimensionalidade. No entanto, em problemas com alta dimensionalidade os AGs frequentemente demandam alto poder computacional e tendem a apresentar tempo de execução elevado.

A obtenção de soluções de qualidade para um problema de otimização frequentemente demanda um longo tempo de execução. Por esse motivo, existe uma demanda constante da indústria e da pesquisa para que AGs consigam aplicar cada vez mais operações, e consequentemente analisar cada vez mais soluções, em tempo de execução reduzido.

Por ser uma meta-heurística baseada em população, AGs tem uma inclinação a se adaptarem eficientemente em arquiteturas paralelas. Logo, o caminho adotado pela pesquisa para aceleração desses algoritmos é o de paralelização da sua execução. Arquiteturas paralelas de computação distribuída, como *clusters*, são uma alternativa que exigem infra-estrutura adequada e, portanto, é custoso mantê-las. Em contraste, a cres-

cente adesão das GPUs ao longo da última década tornou esses dispositivos *mainstream*, sendo encontrados em qualquer máquina de uso pessoal ou profissional. Devido a essa conveniência e por ser uma arquitetura dedicada à execução paralela com alto poder computacional, se torna natural a tentativa de usá-las para melhorar o desempenho desses algoritmos. No entanto, a programação para essas arquiteturas contam com a possibilidade de execução de milhares de *threads* concorrentes em um único dispositivo, diferenciando-se do modelo de programação para as arquiteturas de CPU, que utilizam poucas *threads*, e de programação distribuída tradicionais, baseado na troca de mensagens entre diferentes CPUs.

A grande diferença que a programação para GPUs traz consigo é que suas características arquiteturais, como o modelo de execução e hierarquia de memória, são expostas no modelo de programação de tal forma que não é possível abstraí-las ao projetar um algoritmo. Logo, o programador deve explorar a arquitetura de maneira eficiente, de modo a utilizar o máximo de paralelismo possível, otimizar os acessos à memória e o fluxo de execução das instruções. Os projetos de algoritmos para essas arquiteturas devem ser pensados em termos de: (1) granularidade do paralelismo, (2) organização dos dados na memória, (3) configuração de como as *threads* serão distribuídas entre os processadores para execução (CHENG; GEN, 2019).

O primeiro surge da arquitetura massivamente paralela das GPUs, contendo um número de *threads* concorrentes na execução superior a qualquer outro modelo de programação paralela anterior. O segundo é devido a maneira como as transações de memória são feitas nessas arquiteturas. Os dados da memória das GPUs são transacionados em grupos de *bytes* para maximizar a largura de banda da memória. Esses grupos pertencem a endereços sequenciais na memória e, portanto, os dados devem estar organizados de forma que se maximize o número de *bytes* acessados por transação de memória. O terceiro é importante porque as diferentes formas de distribuir as *threads* entre as unidades de execução da GPU afetam a granularidade do paralelismo na execução, a eficiência das transações de memória e do fluxo de instruções.

Portanto, é de grande interesse científico e da indústria estudar as diferentes formas de se adaptar os algoritmos genéticos e outras técnicas de otimização bem estabelecidas nessas novas arquiteturas.

1.2 Metodologia

Este trabalho tem o intuito de avaliar uma implementação de AG que utiliza a memória compartilhada da GPU para obter cooperação na computação paralela e uma que utiliza as funções de *Warp-Shuffle* da CUDA para esse fim. Com elas, é possível explorar tanto a granularidade em nível de indivíduo, o que significa que cada thread calcula as operações genéticas de um indivíduo independentemente, e a granularidade em nível de dimensão, o qual as operações genéticas de um indivíduo são calculadas por um conjunto de *threads* cooperativamente.

Uma das implementações utiliza a memória compartilhada da GPU, permitindo explorar o paralelismo da arquitetura em diferentes granularidades. A outra utiliza uma técnica de programação introduzida no CUDA 9, chamada *Cooperative Groups* (NVIDIA, 2021b), para que a cooperação entre *threads* utilize apenas registradores em sua execução a fim de diminuir a latência de acesso nas operações de memória. No entanto, essa técnica limita o número de *threads* que cooperam simultaneamente na computação de um indivíduo.

O foco deste trabalho está em avaliar como diferentes granularidades de paralelismo e configurações da execução das *threads* afetam o desempenho da execução. O desempenho é medido através do *speedup* das implementações em GPU em relação a uma implementação sequencial em CPU. Para avaliar as vantagens e desvantagens de cada implementação foram feitos os seguintes experimentos:

- Análise do ganho em termos de eficiência da implementação com memória compartilhada e a que utiliza apenas operações entre registradores;
- Análise da distribuição de execução das *threads* entre os processadores para analisar a relação entre os diferentes níveis de paralelismo e o desempenho da execução;
- Modificar os parâmetros de execução do AG, como o tamanho da população e número de dimensões do problema de *benchmark*, a fim de observar como o *speedup* de cada implementação escala em função da variação desses parâmetros;

Os resultados obtidos pelos experimentos demonstram que paralelizar o algoritmo de forma que cada thread da GPU compute um gene por vez é mais vantajoso que fazer com que cada thread compute um indivíduo da população. Além disso, é mostrado que, apesar da latência de acesso reduzida, a implementação com registradores não é tão eficiente quanto a implementação com memória compartilhada em problemas de grande di-

mensão. Também é demonstrado que utilizar o número máximo de *threads* para execução de um indivíduo não implica, necessariamente, aumento do desempenho.

Esta monografia está organizada da seguinte forma. Capítulo 2 apresenta a base teórica utilizada no trabalho, introduzindo conceitos fundamentais de meta-heurísticas, assim como técnicas mais comuns para implementação de AGs. O capítulo também apresenta os principais conceitos para entendimento do modelo de programação e arquitetura CUDA. O Capítulo 3 detalha a implementação dos AGs em CUDA abordando os conceitos apresentados no capítulo anterior. Por fim, no Capítulo 4, são descritos os experimentos e são analisados os resultados obtidos.

2 BASE TEÓRICA

Este capítulo descreve, de forma introdutória, os principais conceitos utilizados para a realização desta monografia. A primeira seção do capítulo descreve o que são meta-heurísticas e introduzem os principais conceitos de algoritmos genéticos. A segunda seção apresenta uma introdução sobre GPUs e foca na descrição da plataforma CUDA da NVIDIA.

2.1 Meta-heurísticas

Nesta seção será apresentado brevemente o que são meta-heurísticas e em seguida a classe de meta-heurísticas que será utilizada nesse trabalho: os Algoritmos Genéticos.

2.1.1 Visão Geral

Um problema de otimização pode ser definido pelo par (S, f), onde S representa todas soluções possíveis do problema, ou espaço de busca, e f a função objetivo que se quer otimizar. Essa função atribui um valor qualitativo a cada solução $s \in S$, representado por um número real. A função f permite a comparação de qualquer par de soluções no espaço de busca (TALBI, 2009, p. 3).

A solução dentre todas soluções possíveis que otimiza a função f, ou seja, minimiza ou maximiza o valor de f, é chamada de *ótimo global*. Uma solução que otimiza uma região de f (ou uma solução que é ótima dentro de um conjunto de soluções amostradas do espaço de busca), é chamada de *ótimo local*.

Uma heurística é um procedimento de busca que tenta encontrar soluções aproximadas para problemas de otimização que não se sabe um método exato para calcular a solução ótima em tempo razoável porque o espaço de busca é muito grande (LUKE, 2013, p. 9-10). As meta-heurísticas são definidas como um conjunto de procedimentos de alto-nível que fornecem diretrizes ou estratégias para desenvolver algoritmos de otimização baseados em heurísticas (SÖRENSEN; GLOVER, 2013). Frequentemente esses procedimentos implementam métodos de otimização que fazem uso de componentes estocásticos durante a busca. Os procedimentos definidos pelas meta-heurísticas são independentes do problema, pois eles utilizam apenas os valores da função objetivo f, abstraindo-se, por-

tanto, como os problemas são modelados.

Um exemplo de meta-heurística é o algoritmo *Hill-Climbing* (HC) (TALBI, 2009, p. 121-126). O HC é uma meta-heurística simples que, começando de uma solução aleatória, testa iterativamente novas soluções que estão próximas da região da solução corrente e adota uma nova se ela for melhor que a atual. O algoritmo é chamado de *Hill-Climbing* (em português *Escalada de Colina*), pois espera-se que se encontre soluções melhores com as iterações, eventualmente chegando no máximo global da função (idealmente). Esse máximo seria a *colina* que o algoritmo tenta subir.

As meta-heurísticas podem ser divididas em duas classes: as de *Solução Única* e as *Baseadas em População* (TALBI, 2009, p. 23-25). Meta-heurísticas de solução única são algoritmos que implementam heurísticas que tentam ativamente melhorar uma solução corrente com alguma estratégia para escapar de ótimos locais. Portanto, esses algoritmos tentam sofisticar o método de busca que a solução vai utilizar para percorrer a função sendo otimizada (TALBI, 2009, p. 87). Geralmente, essa classe de meta-heurísticas, baseia a sua busca na diferença entre a aptidão da solução corrente e a de soluções próximas, chamadas de vizinhos. Baseando-se na comparação com os vizinhos, é preciso decidir por qual caminho a busca irá seguir. Após essa decisão, um *movimento* é aplicado à solução corrente. Os diferentes métodos baseados em solução única diferem entre si na escolha e comparação com os vizinhos e nos movimentos aplicados às soluções. Exemplos de meta-heurísticas de solução única incluem o *Simulated Annealing* (SA) (TALBI, 2009, p. 126-133), o *Hill-Climbing* (HC) e a *Pesquisa Tabu* (TS) (TALBI, 2009, p. 140-146).

As meta-heurísticas baseadas em população tem como procedimento padrão a inicialização de um conjunto de soluções distintas, e a exploração do espaço de busca é feita principalmente através da troca de informações entre essas soluções (LUKE, 2013, p. 31-32). Portanto, essa classe de algoritmos não tenta sofisticar a maneira com que cada solução percorre a função, como nos algoritmos de solução única, ela tenta guiar sua busca utilizando uma ótica coletiva que analisa o resultado a partir do conjunto, e não do indivíduo. Dessa forma, o objetivo desses algoritmos é fazer com que o conjunto de soluções troquem informações sobre o espaço de busca de maneira inteligente e tentar restringir o espaço de busca à regiões que tenham potencial de conter soluções ótimas. Exemplos de métodos baseados em população incluem a classe dos *Algoritmos Evolutivos* (AEs) (BARTZ-BEIELSTEIN et al., 2014) e a *Otimização por Enxame de Partículas* (OEPs) (TALBI, 2009, p. 247-252).

Nesta seção é apresentada brevemente a classe dos algoritmos evolutivos, em se-

guida, é apresentado os algoritmos genéticos e suas características de implementação.

2.1.2 Algoritmos Evolutivos

Algoritmos Evolutivos (AEs) são meta-heurísticas baseadas em população inspiradas na seleção natural de Darwin. Neste método os indivíduos mais adaptados tem mais chances de sobreviver e passar seu material genético adiante. Para isso, o algoritmo implementa mecanismos importantes de reprodução e evolução, como a *recombinação de genes*, *mutação* e *seleção* (LUKE, 2013, p. 31-32).

Os AEs utilizam um vocabulário especial análogo a conceitos da biologia. Por exemplo, as soluções do espaço de busca são chamadas de **cromossomo** ou **indivíduos**. Um cromossomo pode ter diversas representações, como uma *string* binária, um vetor ou diversas outras estruturas de dados. No caso de um cromossomo ser representado por um vetor, cada índice desse vetor é chamado de **gene**. Abaixo, uma lista dos principais termos dos AEs utilizados neste trabalho:

- **Indivíduos**: representam as soluções do espaço de busca. Pode ser chamado também de **cromossomo**;
- **População**: conjunto de indivíduos;
- **Gene**: uma das dimensões no vetor que representa o indivíduo/cromossomo;
- **Filhos**: um conjunto de novas soluções geradas após a fase de recombinação de genes;
- **Função de Aptidão**: função objetivo do problema.

Os AEs mais populares atualmente são os *Algoritmos Genéticos*, *Estratégias de Evolução* (BEYER; SCHWEFEL, 2002) e *Evolução Diferencial* (TALBI, 2009, p. 225-228). Este trabalho foi desenvolvido utilizando algoritmos genéticos, logo será o foco desta monografia.

2.1.3 Algoritmos Genéticos

O Algoritmo 1 mostra uma descrição em pseudo-código de como os AGs funcionam (LUKE, 2013, p. 37). Todo algoritmo começa inicializando uma população, chamada de primeira geração (Linha 1). Então essa primeira geração passa por um *ciclo evolutivo*

(CHOPARD; TOMASSINI, 2018, p. 117-118). Essa fase é representada pelo laço de repetição que inicia na Linha 4. Esse laço consiste em selecionar um conjunto de pais para reprodução (Linha 6), recombinar seus genes para criação de uma nova população (Linha 7), aplicar um operador de mutação nesses novos indivíduos (Linha 8), e então avaliar a aptidão de cada indivíduo novo gerado (Linha 11). Após isso, uma fase de substituição irá integrar os filhos à população anterior, criando uma nova geração (Linha 12). Esse processo é repetido até que algum critério de parada seja satisfeito.

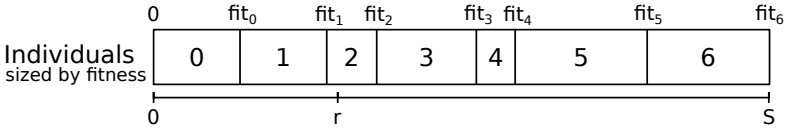
Algorithm 1 Algoritmos Genéticos

```
1:  P ← initialize population
2:  evaluate_fitness(P)
3:  generation ← 1
4:  repeat                                          ▷ início do ciclo evolutivo
5:      for i = 1...pop_size/2 do
6:          P1, P2 ← selection(P)
7:          C1, C2 ← crossover(P1, P2)
8:          C1, C2 ← mutation(C1, C2)
9:          C ← C ∪ {C1, C2}
10:     end for
11:     evaluate_fitness(C)
12:     P ← replacement(P, C)
13:     generation ← generation + 1
14: until stopping criteria met
```

- *Seleção Proporcional à Aptidão*: Também conhecida como *Seleção por Roleta*, é a técnica de seleção originalmente proposta para algoritmos genéticos (LUKE, 2013, p. 44). Esse método tem como objetivo selecionar indivíduos proporcionalmente a sua adaptabilidade ao problema sendo resolvido. Para isso, é feito um dimensionamento da aptidão de cada indivíduo a partir da soma *S* das aptidões de toda população. A Figura 2.1 ilustra o funcionamento do algoritmo. A barra da figura possui tamanho *S*; o número dentro de cada retângulo representam o índice do i-ésimo indivíduo; cada retângulo possui tamanho proporcional ao seu valor de aptidão (fit_i). Um número r dentro do intervalo $[0, S]$ é sorteado sob uma distruibuição uniforme para selecionar um indivíduo. O k-ésimo indivíduo é selecionado quando $\sum_0^{k-1} fit_i \leq r < \sum_0^k fit_i$ (LUKE, 2013, p. 44).
- *Seleção por Torneio*: A seleção por torneio é uma estratégia popular na literatura que também tem como objetivo escolher indivíduos de acordo com a adaptabilidade destes. Todavia, esse método tem uma seleção mais flexível, pois o tamanho do torneio *t* define o quanto a seleção será aleatória. A ideia do algoritmo é selecionar

t indivíduos da população aleatoriamente e, dentre os selecionados, retornar aquele com maior adaptabilidade (LUKE, 2013, p. 45). Portanto, quanto menos indivíduos forem selecionados para o torneio, mais aleatória é a seleção.

Figura 2.1: Seleção Proporcional à Aptidão (Seleção por Roleta)



Fonte: O Autor

2.1.4 Operadores Genéticos

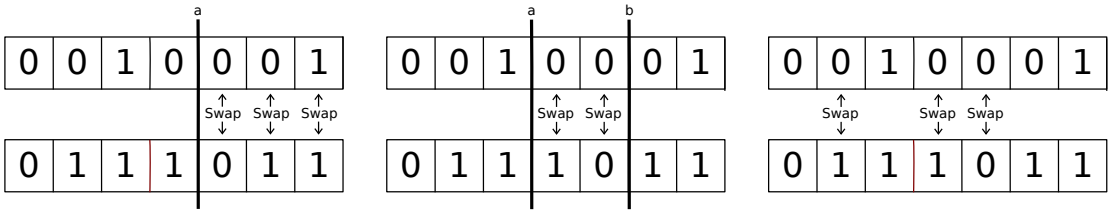
Novos indivíduos são produzidos através da aplicação de operadores genéticos nos pais selecionados, como a recombinação de genes (*crossover*) e a mutação. O operador de *crossover* recombina os genes de dois cromossomos distintos para geração de outros dois novos, enquanto o operador de mutação aplica transformações nos genes de um cromossomo. Geralmente ambos fazem parte da geração de novos indivíduos, sendo o *crossover* o primeiro a ser aplicado, seguido da mutação.

O *crossover* é inspirado pela reprodução sexual de seres vivos, o qual dois indivíduos trocam e recombinam seus genes para produzir um novo. Nessa técnica, cada casal de pais irá gerar dois novos indivíduos. A recombinação do material genético ocorre com uma certa probabilidade, chamada de *probabilidade de crossover*. A Figura 2.2 mostra as três maneiras clássicas de aplicar o *crossover* entre dois vetores (LUKE, 2013, p. 37-39). São elas:

- *Recombinação de um ponto*: sorteia um índice *a* entre 0 e o tamanho dos vetores *n*. Na sequência são trocados todos os genes de *a* até *n*.
- *Recombinação de dois pontos*: sorteia dois índices *a* e *b* dentro do intervalo [0, n] e troca todos os genes entre *a* e *b*.
- *Recombinação Uniforme*: aplica o operador de crossover em todos os genes dos vetores e os troca com uma probabilidade *p*.

A troca de genes é usada quando os cromossomos são vetores de valores binários. Mas quando os vetores são formados por valores contínuos, abordagens diferentes podem ser usadas. Um algoritmo famoso para lidar com valores contínuos é o chamado

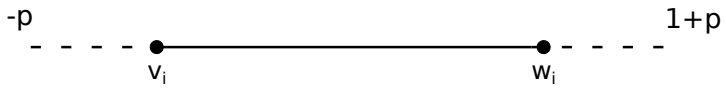
Figura 2.2: Métodos Clássicos de Recombinação



Fonte: O Autor

Recombinação de Linha (MUHLENBEIN; SCHLIERKAMP-VOOSEN, 1993). Nele, a recombinação de dois genes v_i e w_i funciona da seguinte forma: uma linha é traçada entre v_i e w_i, então dois pontos ao longo dessa linha são sorteados e atribuídos às soluções filhas (Figura 2.3). A linha pode ser estendida de forma que seja possível atribuir valores além dos pontos limitados pelos genes dos pais. Um valor positivo p, definido a priori, estabelece os valores limites para o sorteio dos pontos, que podem cair em qualquer ponto da linha entre $[-p, 1 + p]$.

Figura 2.3: Recombinação de Linha



Fonte: O Autor

2.2 Unidades de Processamento Gráfico

A primeira Unidade de Processamento Gráfico (GPU, do inglês *Graphics Processing Unit*) programável foi lançada em 2001 pela NVIDIA. A GPU se chama GeForce 3 e foi um marco da tecnologia dos processadores gráficos. Anteriormente, as placas gráficas atuavam como aceleradores de *hardware* com objetivo exclusivo de acelerar a computação de aplicações gráficas e de vídeo. Com o lançamento da GeForce 3, os programadores conseguiram pela primeira vez ter controle exato da computação feita na GPU, sendo possível programar vértices para renderização e sombreadores de *pixels*.

Desde então, muitos pesquisadores iniciaram a busca por alternativas que tornassem possível a utilização de GPUs para computação geral através de uma Interface de Programação de Aplicações (API, do inglês *Application Programming Interface*), interpretando os valores que representavam as cores dos *pixels* como um número qualquer no

domínio da aplicação (SANDERS; KANDROT, 2010, p. 5-6).

Em 2006, a NVIDIA lançou a placa de vídeo GeForce 8800 GTX, a primeira placa de vídeo feita com suporte a arquitetura CUDA (*Compute Unified Device Architecture*). E assim iniciava, formalmente, o caminho das GPUs em direção a computação de propósito geral.

Este trabalho foi desenvolvido utilizando a plataforma CUDA, portanto ela será a única plataforma apresentada. Porém, atualmente, as arquiteturas de placa de vídeo da AMD estão tão eficientes quanto as da NVIDIA. Além disso, a framework de código aberto *Open-CL* para programação de GPUs está adquirindo cada vez mais relevância acadêmica (Diaz; Muñoz-Caro; Niño, 2012).

2.2.1 CUDA

CUDA (do inglês *Compute Unified Device Architecture*) é uma plataforma de computação paralela e modelo de programação criada e mantida pela NVIDIA. Com CUDA, desenvolvedores podem programar GPUs *CUDA-enabled* para computações de propósito geral utilizando linguagens de programação populares como C, C++, Fortran, Python, entre outras. Devido ao seu modelo escalável e eficiente de programação paralela e a grande capacidade de computação das GPUs, CUDA tem atraído a atenção de diversas comunidades científicas, abrangendo várias áreas de pesquisa que precisam de computação de alto desempenho.

2.2.2 Modelo de Programação

CUDA é uma plataforma para computação heterogênea. Seu modelo de programação assume um sistema constituído por diferentes tipos de processadores: *hosts* (ex: CPU) e *dispositivos* (ex: GPUs), cada um com sua própria memória. Por padrão, o *host* não pode acessar nenhum endereço de memória do dispositivo e vice-versa. Todos os dados são transferidos entre eles através do barramento de comunicação usando chamadas explícitas de cópia de memória. No entanto, a NVIDIA introduziu a Memória Unificada a partir da CUDA 6.0, permitindo que a CPU e as GPUs compartilhem os mesmos endereços virtualmente, desonerando os programadores do uso dessas chamadas explícitas de memória (NVIDIA, 2021b).

O código compilado para o *host* é diferente do compilado para o dispositivo e, portanto, é preciso definir explicitamente quais funções ou variáveis são feitas para GPU e quais são feitas para CPU de modo que o compilador CUDA possa gerar o código de máquina para o alvo correto. O modelo de programação divide o código em duas partes: (1) código do *host* e (2) código do dispositivo. Essa divisão é feita através de marcadores especiais (Tabela 2.1) que são adicionados antes da declaração de funções e variáveis. Por exemplo, o qualificador `__global__` é utilizado para as funções que serão compiladas para a arquitetura da GPU, mas podem ser chamadas pela CPU. Diferentemente das funções com o qualificador `__device__` que são compiladas para GPU, mas podem ser chamadas apenas em funções que executam na GPU. O papel do *host* é controlar o contexto e o fluxo de execução principal da aplicação, enquanto a GPU atua como um co-processador, cujas tarefas são enviadas pela CPU através da chamada de *kernels*. Um *kernel* é o código (ou uma função) que executa na GPU. Diferentemente das chamadas de funções convencionais em C, a chamada de um *kernel* é assíncrona por padrão e quando ele é lançado, o controle retorna imediatamente para o *host* (CHENG MAX GROSSMAN, 2014, p. 36-39). Além disso, eles são chamados com uma síntaxe especial que utilizam *brackets* `<<<...>>>`, cujo papel é de comunicar à GPU sobre como o processamento das *threads* deve ser mapeado dentro do dispositivo.

Tabela 2.1: Qualificadores de função no CUDA

Qualificadores	*Execução*	*Chamamento*
`__global__`	Executa no dispositivo	Pode ser chamado pelo host e pelos dispositivos
`__device__`	Executa nos dispositivos	Pode ser chamado pelos dispositivos apenas
`__host__`	Executa no host	Pode ser chamado pelo host apenas

Em CUDA, uma thread é uma entidade abstrata que representa a execução de um *kernel*. Quando um *kernel* é lançado para execução, são criadas diferentes *threads* que executam o código desse *kernel*, sendo atribuída a cada thread uma indexação única (SANDERS; KANDROT, 2010, p. 44). As *threads* são agrupadas em *Blocos de Threads* e os blocos são agrupados em um *Grid*. A arquitetura de GPU é projetada para lidar com milhares de *threads* concorrentes e, com essa abstração na programação, CUDA fornece um modelo hierárquico e escalável que ajuda o programador a definir os níveis de paralelismo e cooperação entre elas.

Todas as *threads* criadas pelo lançamento de um *kernel* executam o mesmo có-

digo e compartilham o mesmo espaço de *Memória Global*. As *threads* dentro de um mesmo bloco compartilham também um espaço comum de *Memória Compartilhada*, o qual é frequentemente utilizado para obter cooperação entre essas *threads*. Já que elas executam concorrentemente, é preciso um mecanismo de sincronização para coordenar o acesso a essa memória. Esse mecanismo é a função `__syncthreads()`. Ela atua como uma barreira de execução que bloqueia todas as *threads* **dentro de um mesmo bloco** até que todas elas cheguem na mesma instrução. Não há mecanismo equivalente para que *threads* de diferentes blocos consigam sincronizar sua execução, mas eles podem se coordenar usando operações atômicas na memória global (HENNESSY; PATTERSON, 2017, p. 312).

A execução de um *kernel* é distribuída entre os processadores da GPU de acordo com o mapeamento do grid definido no momento da chamada. Após isso, todas as *threads* e todos os blocos são indexados de acordo. Tanto os blocos como os grids podem ter uma estrutura unidimensional, bidimensional ou tridimensional. O índice das *threads* e dos blocos são acessados pelo identificador `threadIdx` e `blockIdx`, respectivamente, seguidos pela dimensão acessada. Por exemplo, dado um bloco com 2 dimensões, para obter o índice de uma thread ao longo do eixo x e y, basta acessar `threadIdx.x` e `threadIdx.y`. Com a mesma lógica, para acessar o tamanho do bloco e do grid, os identificadores `blockDim` e `gridDim` são utilizados (NVIDIA, 2021b).

Os blocos são processados por processadores chamados *Streaming Multiprocessors* (SM). As GPUs contemporâneas podem conter dezenas de SMs, como a placa GeForce RTX 3090, lançada em 2020, que contém 82 SMs. Esses processadores tem execuções independentes, e podem funcionar todos ao mesmo tempo. Logo, uma GPU atual tem capacidade de processar milhares de *threads* simultaneamente. Um bloco, quando despachado para execução em um SM, aloca os recursos do processador até que ele termine sua execução. Além disso, o número de blocos que podem ser executados por um SM, é inversamente proporcional à quantidade de recursos que os blocos utilizam. Por exemplo, quanto mais memória cada bloco utilizar, menos blocos poderão ser lançados.

2.2.3 Arquitetura

A arquitetura de GPU tem um *design* e modelo de execução escaláveis, permitindo que diversas unidades de execução chamadas *Streaming Multiprocessors* (SM) realizem grandes tarefas computacionais simultaneamente. Uma GPU pode ter de uma até dezenas

de SMs, então, para prover escalabilidade, eles contam com um escalonador de blocos de *threads* que atribui os blocos do grid a cada processador. Dessa forma, não importa quantas SMs uma GPU contém, pois o escalonador garante que todo bloco será executado. As execuções nas SMs são independentes entre si e uma única SM pode processar até 1024 *threads* concorrentemente. Como um *chip* pode ter dezenas desses processadores, é possível executar milhares de *threads* concorrentemente em uma GPU.

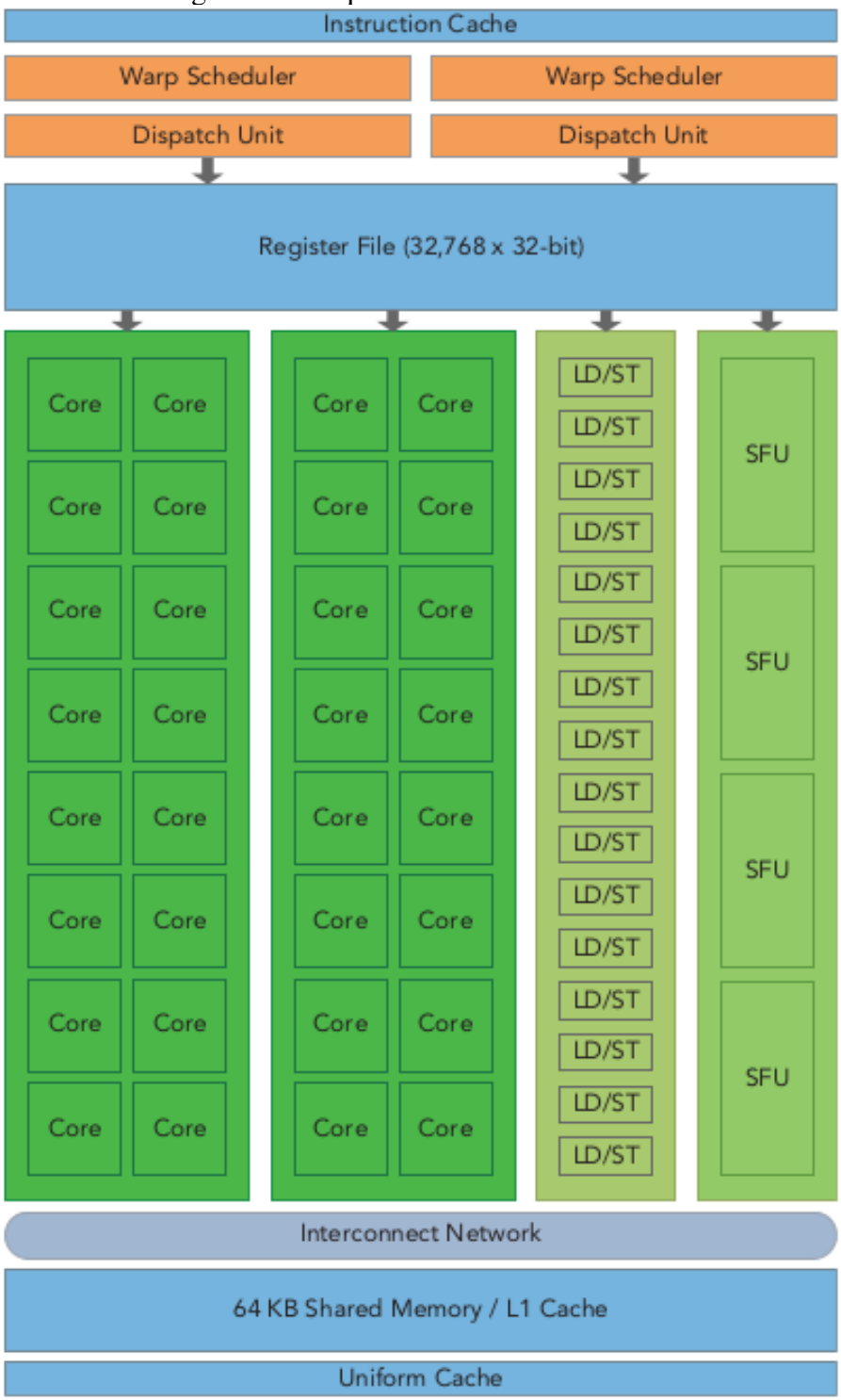
A Figura 2.4 apresenta a organização de um SM da arquitetura Fermi, lançada em 2010. Em um SM há quatro componentes principais que devem ser destacados. Os ***CUDA Cores***, representados pelos quadrados verdes, contém as unidades de execução dos SMs. A **memória compartilhada/*Cache* L1** é uma memória acessível por todas unidades de execução dentro de um mesmo SM. Outro componente de memória dentro dos *Streaming Multiprocessors* é o **Arquivo de Registradores**, cujo os recursos são repartidos entre as *threads* pertencentes ao bloco. Por último, os **escalonadores de *warp*** que mantém uma fila de *warps* que serão despachados e executados.

A natureza hierárquica do modelo de programação descrito na última seção vem diretamente da organização das GPUs. A Figura 2.5 mostra como as abstrações do modelo são mapeadas para os diferentes níveis de unidades de execução. É possível observar a correspondência entre as *threads* que são despachadas para a execução nos CUDA Cores, os blocos que são agendados em um SM e os grids que são lançados para execução no dispositivo quando um *kernel* é invocado. É importante ressaltar que essa hierarquia não está presente apenas nas unidades de execução, mas também na organização da memória.

Em CUDA, o modelo de memória expõe uma hierarquia de 3 níveis. Assim como todo processador moderno, essa hierarquia é composta por memórias de progressivamente menor latência, porém menor capacidade. A memória que todo componente da GPU pode acessar é chamada de Memória Global. Ela é também a memória com maior espaço e latência no *hardware*. Devido a alta latência de acesso, o *design* de memória da NVIDIA busca melhorar o desempenho otimizando o fluxo das transferências de memória. As transações de memória são feitas em grandes fatias de dados, com cada requisição lendo e tranferindo multiplos endereços distintos. Portanto, para obter o melhor desempenho possível, o programador deve se preocupar em organizar os acessos à memória global de forma que as requisições tenham os endereços alinhados e unidos (CHENG MAX GROSSMAN, 2014, p. 159). Além disso, o conteúdo armazenado na memória global persiste durante todo o tempo de vida da aplicação.

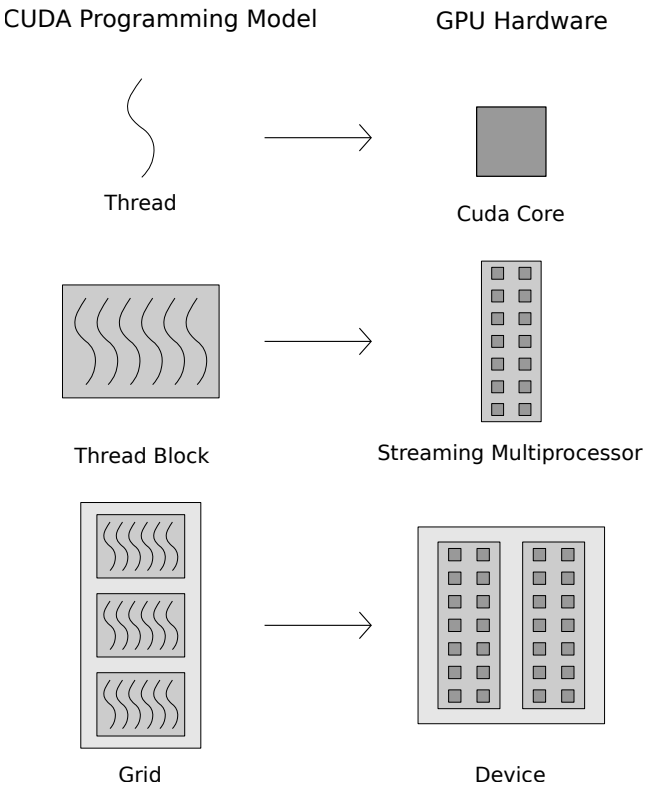
Descendo mais um nível na hierarquia, temos a Memória Compartilhada/*Cache*

Figura 2.4: Arquitetura Fermi da NVIDIA



Fonte: Retirado do livro *Professional CUDA C Programming*

Figura 2.5: Mapeamento das abstrações do modelo de programação para as unidades de execução da GPU



Fonte: O Autor

L1, presente dentro de cada *Streaming Multiprocessor* (SM). Nas arquiteturas mais recentes, a memória compartilhada e a *cache* L1 compartilham o mesmo espaço físico no *chip* (a exceção é a arquitetura Pascal) (NVIDIA, 2021b). Apesar de ocuparem o mesmo espaço físico, apresentam diferenças significativas. A principal é que o programador tem controle total sobre o que é armazenado na memória compartilhada, enquanto que na *cache* L1, o despejo dos dados é feito de forma autônoma pela GPU (CHENG MAX GROSSMAN, 2014, p. 213).

O conteúdo da memória compartilhada é restrito a todas *threads* de um mesmo bloco e ele persiste durante todo o tempo de vida de um bloco de *threads*. Além disso, seu espaço de endereços é compartilhado por todas *threads* e blocos residentes no SM. Portanto, existe um *trade-off* entre o número de blocos residentes de um SM e a quantidade de memória compartilhada alocada pelo *kernel* (CHENG MAX GROSSMAN, 2014, p. 140). Por exemplo, se o tamanho da memória compartilhada de um SM for *48KB*, e um bloco de um *kernel* alocar *20KB* de espaço, é possível ter apenas dois blocos residentes nesse SM.

O acesso à memória compartilhada é feito através de bancos de memória. Bancos de memória distintos podem ser acessados simultaneamente. Em CUDA, a memória

compartilhada contam com 32 bancos distintos devido às 32 *threads* em um *warp*. Dessa forma, se não houver conflito de acesso a um banco, é possível retornar todos os dados de uma solicitação de 32 *threads* em uma única transação de memória (NVIDIA, 2021b).

Os registradores são a menor e mais rápida unidade de memória das GPUs. Além disso, é o nível mais baixo na hierarquia de memória, sendo utilizados para armazenar as variáveis declaradas localmente por um *kernel*. Essas variáveis são privadas para cada thread. Se o número de registradores disponíveis for o suficiente, é possível que o armazenamento de vetores locais seja delegado aos registradores (CHENG MAX GROSSMAN, 2014, p. 139).

Registradores são recursos escassos divididos entre todas as *warps* ativas em um SM através de um Arquivo de Registradores. O número de registradores disponíveis por bloco de *threads* varia de acordo com a arquitetura, porém este número fica entre 32K e 64K. CUDA limita a 255 registradores de 32-bit o número de registradores que podem ser utilizados por cada *thread* no bloco. Quando não há mais espaço para o armazenamento de novas variáveis nos registradores, elas serão armazenadas na *Memória Local*. A memória local ocupa o mesmo espaço físico da memória global. Portanto, um uso excessivo de *threads* ativas pode prejudicar o desempenho geral da aplicação por fazer com que variáveis locais sejam guardadas na memória global (CHENG MAX GROSSMAN, 2014, p. 139-140).

A Tabela 2.2 apresenta uma síntese dos tamanhos e velocidade dos tipos de memórias da GPU descritos até aqui.

Tabela 2.2: Tipos de Memória

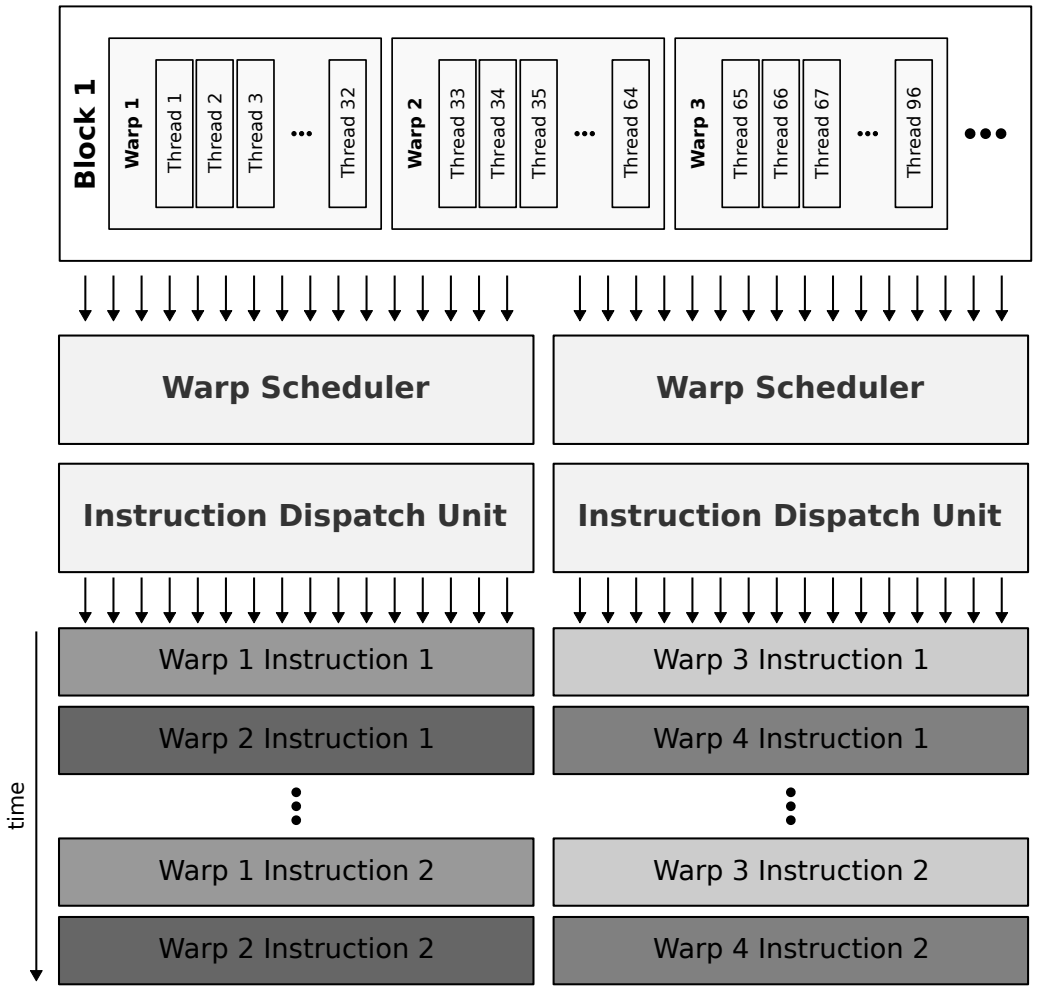
Tipo de memória	*Latência de Acesso*	*Tamanho*	*Tamanho Exato (Titan V)*
Global	Média	Grande	12GB
Shared	Rápida	Pequeno	96KB (max. por SM)
Registradores	Muito Rápida	Muito Pequeno	1KB (max. por *Thread*)
Local	Média	Médio	512 MB (max. por SM)

Fonte: Adaptado de (LUONG, 2011)

A CUDA estende a Taxonomia de Flynn (FLYNN, 1966) introduzindo uma arquitetura *Single Instruction Multiple Threads* (SIMT) ao seu modelo de execução. Esse modelo organiza as *threads* em grupos de 32 chamado *warps*. Cada *warp* tem seu próprio contexto. Os SM agendam a execução dos *warps* enfileirando-os de forma que aquelas que estão prontas para execução fiquem primeiro. Em seguida, as *warps* são enviadas em sequência para a unidade de despacho, que, por sua vez, manda uma única instrução das *threads* para ser executada por todas as *threads* do *warp* (Figura 2.6). Porém, nem todas

elas serão executadas e essa é a principal diferença entre a arquitetura SIMD e a SIMT (NVIDIA, 2021b). Apesar de todas as *threads* em um *warp* serem despachadas em conjunto, iniciando na mesma instrução, elas podem ter comportamentos diferentes e divergir em múltiplos caminhos de execução (HENNESSY; PATTERSON, 2017, p. 323-326).

Figura 2.6: Despacho e execução das *warps* de um bloco



Fonte: O Autor

A divergência entre as execuções de *threads* é um fator extremamente prejudicial ao desempenho da aplicação. Para lidar com condicionais, a GPU utiliza uma máscara para executar todas as *threads* que satisfazem uma dada condição. As *threads* de um *warp* executam em diferentes *lanes* que utilizam um registrador com função de ativar ou desativar a execução de sua *lane*. Dessa forma, uma máscara de execução é aplicada nesses registradores para ativar a execução de todas as *lanes* que satisfazem uma determinada condição e desativar todas aquelas que não a satisfazem. Portanto, em uma instrução IF-THEN-ELSE, primeiro serão executadas as *threads* que satisfazem o condicional e depois

as que não fazem. Isso faz com que, em caminhos de tamanho igual, a execução ocorra com uma eficiência de 50% ou menos (HENNESSY; PATTERSON, 2017, p. 323-326).

2.2.4 Cooperação entre *Threads*

O *hardware* da GPU suporta dois níveis distintos de cooperação entre *threads*: em nível de bloco e em nível de *warp*. O primeiro é o método mais comum na literatura e também é o método mais antigo de cooperação. Ele é feito através do uso da memória compartilhada em combinação com a função de barreira `__synchthreads()`. Ela é necessária para garantir o resultado correto da execução quando há concorrência pelos recursos de memória, já que, apesar das *threads* em um bloco executarem em paralelo à nivel de programação, elas não executam todas ao mesmo tempo no *hardware*. Para utilizar a memória compartilhada, o identificador de tipo `shared` é utilizado antes da declaração de variáveis dentro de um *kernel*.

CUDA suporta cooperação e sincronização com uma granularidade mais fina utilizando as funções de *Warp Shuffle* (em dispositivos com CC[1] 3.x ou maior). Elas são um conjunto de funções que permitem a troca de dados entre *threads* de um mesmo *warp* sem a necessidade do uso de memória compartilhada. Recentemente, no CUDA 9, foi introduzida a funcionalidade de *Grupos Cooperativos*. Essa funcionalidade permite separar as *threads* em diferentes grupos a fim de facilitar o uso de operações coletivas, como as funções de *Warp Shuffle*.

Há diferentes formas de criar um grupo usando a API do CUDA. Uma maneira, por exemplo, é a função `tiled_partition<group_size>(thread_block)`. Dessa forma, o tamanho dos grupos é definido estaticamente, permitindo que o compilador faça otimizações. Definindo um grupo, é possível utilizar os seus métodos com a sintaxe tradicional de programação orientada a objetos. Por exemplo, tendo declarado um grupo `group`, é possível chamar uma função escrevendo `group.function()`. Todas as *threads* pertencentes a um grupo são indexadas em relação a ele. Para acessar esse índice, basta chamar o método `group.thread_rank()`.

Neste trabalho serão utilizadas apenas duas funções de *Warp-Shuffle*: `shfl()` e `shfl_down()`. Grupos cooperativos podem utilizar essas funções como métodos. Eles são invocados, portanto, com a sintaxe de `group.shfl()`, por exemplo. A função `shfl` permite uma thread acessar uma variável de outra thread especificando um

[1] *Compute Capability* (Capacidade Computacional)

índice referente a essa thread (desde que ela esteja no mesmo *warp*). `shfl_down` tem a mesma funcionalidade, porém o acesso é feito por deslocamento. Por exemplo, se uma thread de um grupo invocar o método `group.shfl(var, i)`, ela estará acessando a variável *var* referente a i-ésima thread do grupo. Por outro lado, se uma thread chamar `group.shfl_down(var, i)`, ela estará acessando a thread cujo índice é calculado através do índice da thread que chamou a função somado ao parâmetro `i`.

3 TRABALHOS RELACIONADOS

A partir do início da última década diversos trabalhos foram propostos: desde análises de desempenho de AGs em GPUs até *frameworks* para implementação desses algoritmos para essas arquiteturas. Esta seção apresenta um breve resumo de alguns trabalhos selecionados que se relecionam aos temas do trabalho.

Em Mussi (2009) é apresentado um algoritmo genético em GPU para resolver o problema One-Max. Todas as etapas do AG (seleção, *crossover*, mutação e avaliação) foram implementadas em GPU para evitar transferência de dados entre CPU e GPU. Os indivíduos são representados por *strings* de 1s e 0s, e cada gene é representado por um char. A seleção é feita por torneio; no *kernel* de *crossover*, cada bloco computa dois indivíduos para reprodução e cada thread lida com 4 bits cada; O de mutação utiliza como operador de mutação operações de XOR, com cada thread computando 4 bits e cada bloco lida com um indivíduo; O *kernel* de avaliação implementa uma redução paralela, com uma configuração de grid igual ao *kernel* de mutação.

Em Arora, Tulshyan and Deb (2010) é apresentado uma implementação de AG em que todas as etapas do algoritmo também é feita em GPU. O objetivo do estudo é investigar como a variação de alguns parâmetros afetam o *Speedup* da implementação (e.g tamanho da população, número de dimensões do problema, complexidade do problema e número de *threads* na execução). A variação da complexidade do problema é feita através de diferentes funções de *benchmark*. As funções de *benchmark* testadas são: Função de One-Max, Função Elíptica e Função de Rosenbrock. Os autores apresentam uma descrição de como os *kernels* de seleção *crossover* e mutação foram desenvolvidos, ambos explorando o paralelismo por dimensão. Os experimentos realizados mostram que com problemas mais complexos se obtém menos *Speedup* em relação aos problemas mais simples, mas a medida que o tamanho da população aumenta, os Speedups tendem a ficar parecidos. Além disso, se verifica que para aplicações que há um grande intervalo no qual o aumento do tamanho da população implica aumento no *Speedup*.

Em Luong, Melab and Talbi (2010) é proposto um esquema para implementação de AGs em GPU utilizando modelos de ilha. O esquema propõe que cada bloco seja uma ilha e cada thread esteja associada a um indivíduo, portanto foi utilizado paralelismo a nível de indivíduo. O artigo compara 3 formas distintas implementação. A primeira é um esquema Mestre-Escravo, em que todo o algoritmo genético é executado na CPU, exceto a parte de avaliação das soluções, que é processada pela GPU; cada thread processa a

avaliação de um indivíduo da ilha. O segundo e o terceiro esquema implementam todo o algoritmo na GPU. Nesses esquemas, cada thread representa um indivíduo e cada bloco uma ilha. Os autores limitam uma ilha a um bloco devido a facilidade de cooperação entre os indivíduos de um mesmo bloco. Além disso, eles separam o algoritmo em 6 *kernels* distintos para que tenha uma sincronização implícita entre os blocos ao finalizar a execução de um *kernel*. Esses *kernels* são: seleção, *crossover*, mutação, avaliação, substituição e migração. A diferença entre o segundo e o terceiro esquema é que o terceiro utiliza a memória compartilhada como uma *cache* que auxilia na computação de todos os *kernels*, exceto o de migração. Os resultados do experimento apontam que o esquema com memória compartilhada e que minimiza o número de transferências entre host e dispositivo diminui consideravelmente o tempo de execução da aplicação.

Em Jaros (2012), um algoritmo genético foi proposto para resolver o problema da mochila 0/1 (RASHID; NOVOA; QASEM, 2010) em um *cluster* com múltiplas GPUs. É um artigo detalhado quanto a implementação, apresentando como os dados são codificados, como eles foram posicionados na memória e descrevendo todos os *kernels* desenvolvidos (inclusive fornecendo o código da implementação). A implementação explora o paralelismo por dimensão com granularidade a nível de *warp*, colocando cada indivíduo para ser computado por uma *warp*. Além disso, os autores propõem a codificação dos genes como bits, onde um inteiro representa 32 genes (por ter 32-bits). Todos os blocos lançados aos *kernels* são bidimensionais. O eixo x desses blocos corresponde aos genes, enquanto o eixo y corresponde aos cromossomos. O número de cromossomos em um bloco é fixo em 8, pois 256 *threads* por bloco é considerada uma granularidade boa (SANDERS; KANDROT, 2010, p. 279).

Em Boiani and Parpinelli (2020) são propostos três estruturas diferentes para os blocos de *threads* que são utilizados na execução do algoritmo de Evolução Diferencial em GPU aplicado para predição de estruturas proteicas. Apesar do algoritmo utilizado ser o de Evolução Diferencial, essas estruturas podem ser estendidas em geral para todas meta-heurísticas baseadas em população. A primeira estrutura, chamada de S1, fixa a execução em 32 *threads* por bloco e o número de blocos lançados é calculado pelo tamanho da população dividido por 32. Essa estrutura é utilizada para os *kernels* que realizam tarefas que não dependem do tamanho do problema. A segunda estrutura, S2, utiliza um bloco inteiro para computar as operações relacionadas a um indivíduo da população, explorando a granularidade em nível de dimensão. Nessa estrutura, o número de *threads* utilizadas por um bloco dependem do tamanho do problema a ser otimizado. O número

de threads em um bloco de S2 corresponde ao menor múltiplo de 32 que seja maior ou igual ao número de dimensões do problema. Esse cálculo é feito para que seja lançado o número exato de *warps* para realizar de forma totalmente concorrente os cálculos relacionados a um indivíduo. A terceira estrutura, S3, é relacionada com o domínio do problema de otimização. A S3 é uma extensão da S2, mas ao invés de utilizar o número de dimensões do problema, ela utiliza o PSL (*Protein Sequence Length*) para o cálculo do número de *threads* em um bloco. Os resultados apresentam *Speedups* em relação a implementação feita em CPU de $14,016\times$ para a menor proteína do *dataset* e de $277,536\times$ para a maior.

Por último, um artigo que não aborda algoritmos genéticos, mas faz uma implementação do algoritmo de *Dynamic Time Warping* em GPU que utiliza as diretivas de *warp* para computar os *scores* desse algoritmo (SCHMIDT; HUNDT, 2020). Os autores exploram ao máximo as funções de *Warp-Shuffle*, utilizando-as para diminuir a latência da comunicação entre *threads* e possibilitar computações que utilizem apenas registradores. Os autores conseguem bons resultados em comparação a implementação estado da arte do algoritmo, chegando a um *Speedup* de três ordens de magnitude em relação a implementação estado da arte em CPU e de uma ordem de magnitude em relação a implementação mais rápida do algoritmo em GPU até então.

4 MODELOS PROPOSTOS

O objetivo deste trabalho é avaliar como a granularidade da paralelização e a configuração do grid lançado pelos *kernels* afetam o desempenho da execução. Para isso, a métrica desempenho é avaliada a partir da variação nos parâmetros de execução dos AGs (número de dimensões do problema avaliado e tamanho da população).

A fim de verificar as relações de desempenho entre granularidade e os parâmetros variados, foi desenvolvida uma implementação que permite a execução dos *kernels* em qualquer nível de paralelismo através da configuração do grid lançado. Apenas essa implementação seria necessária para cumprir o objetivo deste trabalho, porém há outras maneiras de se paralelizar AGs utilizando CUDA e seria interessante trazê-las a este trabalho a fim de comparações de desempenho. Portanto, uma outra implementação foi desenvolvida utilizando as funções de *Warp-Shuffle* com os Grupos Cooperativos da CUDA (NVIDIA, 2021b).

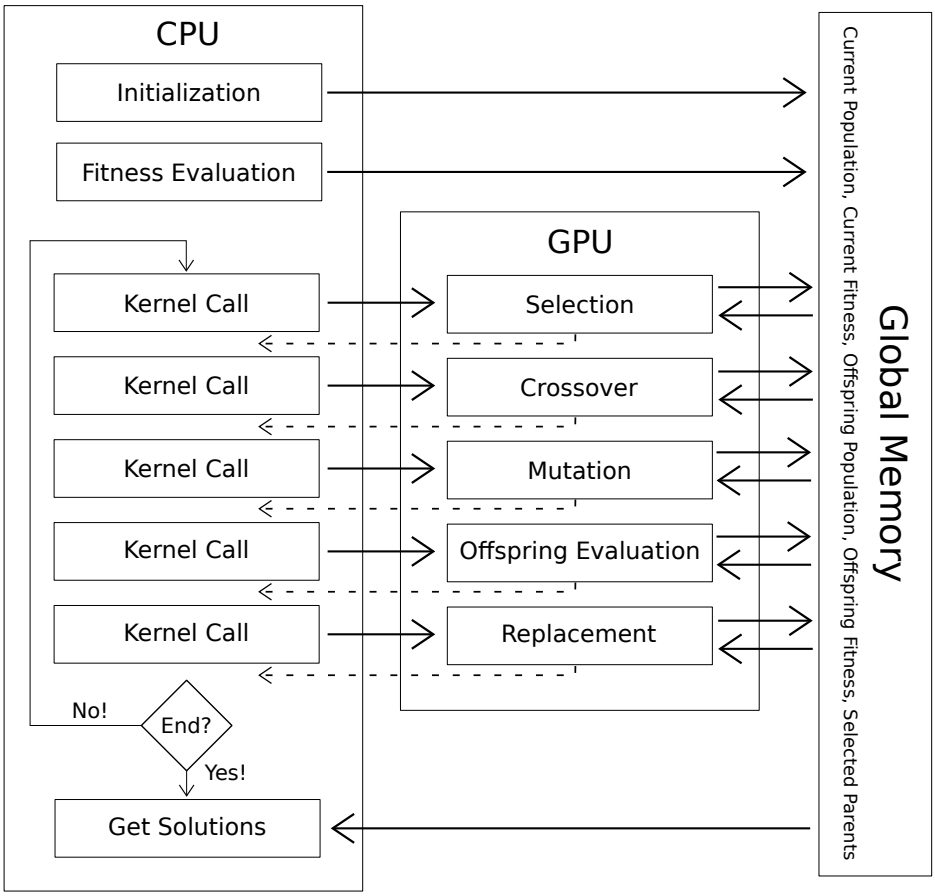
Uma proposta de implementação de AGs baseada em Grupos Cooperativos é apresentada neste capítulo. Essa implementação é também baseada no paralelismo em nível de gene, porém o número de *threads* que cooperam para computar as operações genéticas de um indivíduo tem tamanho fixo de 32 *threads* (um *warp*).

Ambas propostas de implementação seguem o fluxograma da Figura 4.1. Todas rotinas relacionadas ao algoritmo genético são executadas na GPU. Portanto, neste modelo não há cooperação entre a CPU e a GPU. As únicas tarefas da CPU são de controle da aplicação principal, inicialização da população e cálculo da aptidão da população inicial. O uso da memória compartilhada não é mostrado no fluxo. É possível ver o uso da memória global à direita da imagem, no qual são armazenados os dados correspondentes à:

- População corrente
- Aptidões da população corrente
- População de novos indivíduos (gerada após o *crossover*)
- Aptidões dos novos indivíduos
- Pais selecionados (gerados pelo *kernel* de seleção)

A vantagem de implementar um *kernel* para cada rotina do algoritmo genético e chamá-los em sequência é que simplifica o projeto e há uma sincronização implícita dos blocos ao fim de cada *kernel*, garantindo que os dados estejam prontos no começo de cada

Figura 4.1: Fluxo de execução do algoritmo



Fonte: O Autor

tarefa (LUONG; MELAB; TALBI, 2010).

Cada indivíduo da população é representado por um vetor de genes. Portanto, o i-ésimo gene de um indivíduo é acessado com $individuo[i]$. Dessa forma, diversos genes podem ser acessados simultaneamente por diferentes *threads*.

Os *kernels* são estruturados para aproveitar os princípios de *design* descritos no manual de boas práticas da CUDA (NVIDIA, 2021a):

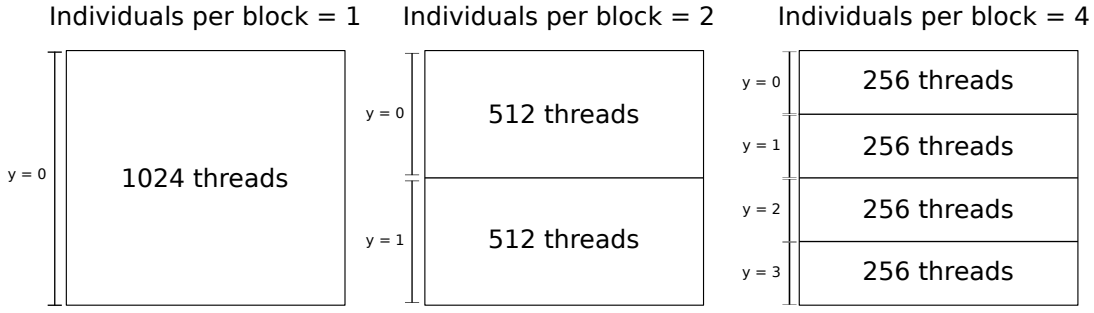
- Minimizar as transferências de dados entre o *host* e o dispositivo;
- Acessar a memória global de maneira alinhada e unida;
- Aproveitar de maneira eficiente a arquitetura massivamente paralela das GPUs através do paralelismo por dimensão.

Neste Capítulo, é apresentada uma descrição detalhada dos kernels implementados para cada rotina dos algoritmos genéticos: seleção, *crossover*, mutação, avaliação de aptidão e substituição. Primeiro, são introduzidos os *kernels* que utilizam um esquema

genérico de paralelismo que permite a execução em qualquer granularidade, podendo-se utilizar n-*threads* por indivíduo. Após isso, é apresentado o esquema baseado em paralelismo por dimensão com granularidade em nível de *warp*, no qual cada indivíduo pode usar apenas 32 *threads*. Essa implementação utiliza, sempre que possível, os Grupos Cooperativos e as funções de *Warp-Shuffle*.

Com exceção dos *kernels* de *seleção*, tanto a implementação n-*threads* como a de *warps* utilizam a ideia de indivíduos por bloco. Nesse modelo, os *kernels* são implementados baseados numa configuração de execução que utiliza blocos bidimensionais, mapeando os índices `threadIdx.y` para índices de indivíduos e os índices de `threadIdx.x` para índices de genes. A Figura 4.2 ilustra como é organizado um bloco bidimensional com 1, 2 e 4 indivíduos por bloco na implementação de n-*threads*. Os índices y na imagem representam os valores de `threadIdx.y`. Cada conjunto de *threads* fica indexado para um único y, por isso esses conjuntos serão chamados de *sub-blocos* ao longo deste texto. A implementação com paralelismo em nível de *warp* funciona da mesma forma, porém cada sub-bloco contém a quantidade fixa de 32 *threads*. O número de blocos lançados por esses *kernels* é calculado pelo tamanho da população dividido pelo número de indivíduos por bloco.

Figura 4.2: Divisão dos blocos pelos indivíduos



Fonte: O Autor

4.1 Granularidade de n-*threads*

Kernels com granularidade n-*threads* representam uma implementação flexível das rotinas de algoritmos genéticos utilizando paralelismo por dimensão. Este modelo permite computar de 1 a 1024 indivíduos em um mesmo bloco, utilizando o máximo de *threads* disponíveis para cada um. Portanto, se um bloco contém apenas um indivíduo, ele

poderá usar até 1024 *threads* do SM. Conseguinte, se um bloco conter 4 indivíduos, cada um terá 256 *threads* disponíveis para uso. Esta implementação sempre utiliza a memória compartilhada para cooperação entre *threads* de um mesmo bloco. Não há cooperação entre blocos distintos.

Seleção

O *kernel* implementa a *Seleção por Roleta*. Ele sempre é chamado com um grid contendo apenas um bloco unidimensional com *population_size threads*. Essa estrutura, com apenas um bloco por grid, foi escolhida porque permite a paralelização da soma dos valores de aptidão e do cálculo de aptidão mínima e máxima da geração avaliada. Os valores são necessários para normalizar as qualidades de cada cromossomo em relação à população corrente. Se a normalização não fosse feita, a seleção não funcionaria para casos em que se somariam valores negativos ou todos valores iguais a zero. Tanto a soma, como o máximo e mínimo são obtidos através de uma redução utilizando a memória compartilhada.

O Algoritmo 2 mostra como a seleção por roleta foi implementada no *kernel*. Cada uma das primeiras *population_size/2 threads* irão rodar esse algoritmo e armazenarão os pais selecionados na memória global. Os números aleatórios r_1 e r_2 são sorteados dentro do intervalo $[0, S]$, onde S é a soma de todas as aptidões da geração considerada.

A variável *acc* acumula os valores das aptidões dos indivíduos da população. Portanto, os laços de repetição das linhas 5-8 e 11-14 vão terminar quando o número aleatório r_i estiver no intervalo correspondente ao indivíduo i cuja aptidão é $acc{-}normalized\_fit(i)$.

Crossover

O *kernel* de *crossover* implementa a técnica de *crossover* de dois pontos com operações de *swap* entre os genes, quando utilizado em problemas de domínio binário e com Recombinação de Linha, quando utilizado em problemas com domínio contínuo. Ele trabalha com um grid de blocos bidimensionais, onde cada sub-bloco ($x = 1...blockDim.x, y = i$) recombina dois pais, gerando dois indivíduos.

A geração de novos indivíduos é feita através da recombinação dos genes dos dois pais selecionados no *kernel* de seleção. Esses pais são armazenados na memória

Algorithm 2 Kernel de Seleção por Roleta

1: $parents[0:1] \leftarrow 0$
2: $r_1 \leftarrow random\_float(0, S)$
3: $r_2 \leftarrow random\_float(0, S)$
4: $acc, acc2 \leftarrow 0$
5: **repeat**
6: $\quad acc \leftarrow acc + normalized\_fit(individual)$
7: $\quad individual \leftarrow individual + 1$
8: **until** $r_1 > acc$
9: $parents[0] \leftarrow individual$
10: $individual \leftarrow 0$
11: **repeat**
12: $\quad acc2 \leftarrow acc2 + normalized\_fit(individual)$
13: $\quad individual \leftarrow individual + 1$
14: **until** $r_2 > acc2$
15: $parents[1] \leftarrow individual$
16: **if** $parents[0] = parents[1]$ **then**
17: $\quad parents[1] \leftarrow new\_parent()$
18: **end if**

compartilhada, pois serão acessados frequentemente por todas as *threads* no sub-bloco durante a etapa de recombinação dos genes. Cada sub-bloco utiliza dois pontos aleatórios (relativos ao *crossover* de dois pontos) e, portanto, é preciso armazená-los na memória compartilhada também.

O *kernel* é dividido em duas etapas: a de inicialização das variáveis na memória compartilhada e a de recombinação dos genes. A primeira etapa utiliza as primeiras *threads* de cada bloco, ignorando a separação por sub-bloco. O código do Fragmento de Código 4.1, mostra como é implementada essa etapa. Em resumo, as operações efetuadas são a de armazenar na memória compartilhada os pais calculados pelo *kernel* de seleção (linha 3) e sortear os dois pontos utilizados pelo *crossover*, também armazenando na memória compartilhada (linha 4 e 5). A variável *tid_block* guarda o identificador da thread dentro do bloco. *parents_smem* é um vetor na memória compartilhada de uma estrutura que contém dois inteiros, o qual cada um representa um pai. O vetor *selected* armazena os pais selecionados pelo *kernel* de seleção e é acessado por diferentes blocos simultaneamente. Cada bloco acessa a memória global de maneira alinhada, pelas primeiras *individuals_p_block threads*. O vetor de estruturas *crossover_points_smem* também está armazenado na memória compartilhada. O acesso ao vetor *state*, que é o gerador de números aleatórios na memória global, também é feito de forma alinhada em cada bloco. Na última linha as *threads* são sincronizadas para o início da recombinação dos genes.

Fragmento de Código 4.1 – Crossover com Memória Compartilhada

```
1 int tid_block = threadIdx.x + blockDim.x*threadIdx.y;
2 if(tid_block < individuals_p_block){
3    parents_smem[tid_block] =
        selected[blockIdx.x*individuals_p_block + tid_block];
4    crossover_points_smem[tid_block].p1 = rand_int(0,
        dimensions, &state[tid_block]);
5    crossover_points_smem[tid_block].p2 = rand_int(0,
        dimensions, &state[tid_block]);
6 }
7 __syncthreads();
```

Na segunda etapa, cada thread é responsável por fazer a recombinação de um gene dos dois pais selecionados. Cada sub-bloco é responsável por recombinar todos os genes de um par de indivíduos de indivíduos. Então, todas as *threads* desse sub-bloco acessam os dois pais e os dois filhos durante a execução. O Algoritmo 3 é uma reprodução simplificada, em pseudo-código, do *kernel* que cada thread executa. Nesse algoritmo, é ignorada a presença de outros blocos, focando na execução de um bloco apenas. Pode-se ver, já na linha 2, que a thread executa um laço de repetição. Esse laço é necessário para assegurar que todos os genes paternos façam parte da reprodução dos filhos, já que o número de dimensões do problema pode exceder o número de *threads* de um sub-bloco. O número de *threads* disponíveis para a recombinação de um par de indivíduos é dada por `blockDim.x`. Logo, quando cada thread executa uma iteração, a operação de *crossover* é aplicada em `blockDim.x` genes. A variável *offset* representa um deslocamento à direita equivalente ao número de *threads* do sub-bloco no acesso ao material genético dos indivíduos. Por exemplo, se há 32 *threads* em cada sub-bloco e um problema de 64 dimensões, na primeira iteração são acessados os genes $[gene_1, gene_{32}]$, então é incrementado 32 ao *offset* e na segunda iteração os genes $[gene_{33}, gene_{64}]$ são acessados. Na linha 4 o índice da dimensão acessada é calculado a partir do identificador da thread mais o $offset$; na linha 5-6, com a ajuda da memória compartilhada, os genes dos dois pais são armazenados em variáveis privadas. Em seguida, é verificado se o gene está no intervalo dos dois pontos sorteados (que estão armazenados na memória compartilhada) e se os genes vão ser recombinados em função da probabilidade de *crossover* (linha 8). Caso ocorra a recombinação nessa posição do material genético dos pais, é aplicada a função de recombinação, gerando dois novos genes. Essa função pode ser um *swap* ou

uma recombinação de linha, dependendo do problema. Por fim, os genes são colocados no indíce que corresponde aos novos indivíduo no vetor $offspring$ (linha 12-13). O índice do sub-bloco $threadIdx.y$ é utilizado para calcular o índice dos novos indivíduos, de forma que o *sub-bloco 0* é responsável por gerar os indivíduos 0 e 1 da nova população, o sub-bloco 1 é responsável por gerar o 2 e 3; assim sucessivamente.

Algorithm 3 Recombinação de genes

```
1:  offset ← 0
2:  repeat
3:      if threadIdx.x + offset < dimensions then
4:          gene_id ← threadIdx.x + offset
5:          gene_p1 ← parent1[gene_id]
6:          gene_p2 ← parent2[gene_id]
7:          if gene_id ∈ [point1, point2] then                ▷ Two-Point Crossover
8:              if rand_float(0, 1) < crossover_probability then
9:                  gene_p1, gene_p2 ← recombination(gene_p1, gene_p2)
10:             end if
11:         end if
12:         offspring[2 * threadIdx.y][gene_id] ← gene_p1
13:         offspring[2 * threadIdx.y + 1][gene_id] ← gene_p2
14:         offset ← offset + blockDim.x
15:     end if
16: until offset = n_dimensions
```

Mutação

O *kernel* de mutação aplica, com uma probabilidade p, um *bit-flip* em todos os genes do cromossomo, quando o problema é de domínio discreto, e aplica uma soma ou subtração de passo k definido a priori, quando as variáveis são contínuas. Ele é configurado para utilizar o máximo de blocos e *threads* possíveis considerando o tamanho do problema e da população. Um bloco pode ser lançado com um ou mais indivíduos de qualquer tamanho, como demonstrado na Figura 4.2. Consequentemente, pode acontecer que um indivíduo tenha menos *threads* disponíveis para fazer uma redução que o número de dimensões do problema.

O Algoritmo 4 descreve o código que cada thread executa. O cálculo de índices dos blocos e do cromossomo foi abstraído para deixar o código mais simples. O vetor $chromosome$ (linha 5) representa um ponteiro para os genes de um indivíduo na memória global. O índice $threadIdx.x + offset$ corresponde a um gene do cromossomo.

Se p for menor que uma probabilidade de mutação pré-definida, o gene do indivíduo $threadIdx.x + offset$ é negado. O laço de repetição garante que todos os genes do indivíduo sejam acessados, mesmo que o número de *threads* disponíveis seja menor que o número de dimensões do problema.

Algorithm 4 Mutação

```
1: offset ← 0
2: repeat
3:     p ← rand_float(0, 1)
4:     if p < Mutation Probability then
5:         chromosome[threadIdx.x + offset] ← ¬chromosome[threadIdx.x + offset]
6:     end if
7:     offset ← offset + blockDim.x
8: until offset ≥ n_dimensions
```

Avaliação de Aptidão

O *kernel* de avaliação aplica uma redução paralela em todos os indivíduos da população. A configuração do grid é a mesma do *kernel* de mutação.

O *kernel* foi projetado para problemas cujo a qualidade da solução é calculada através de um somatório $\sum_{i=0}^{n} f(x_i)$, onde f é qualquer função aplicável em x_i e n é o número de dimensões do problema. Caso esse número exceda o número de *threads* disponíveis para o cromossomo, primeiro é feita uma redução de n dimensões para `blockDim.x` *threads*, acumulando todos os valores correspondentes ao peso de cada gene do indivíduo no cálculo de aptidão. Após essa primeira redução, é feita uma redução final de `blockDim.x` valores para 1. A Figura 4.3 ilustra, à esquerda, como é feita a acumulação desses pesos na memória compartilhada. O cromossomo carrega seus primeiros `blockDim.x` genes, aplica a função de aptidão e guarda esse valor na memória compartilhada, em seguida pega os próximos `blockDim.x` genes, aplica a função e soma nos mesmos endereços da memória compartilhada utilizados anteriormente. O processo se repete até todos os genes do cromossomo forem utilizados. Após isso, restarão apenas `blockDim.x` valores que serão reduzidos (à direita na Figura 4.3) a um único valor, que representa a aptidão da solução.

O Algoritmo 5 descreve como a aptidão de cada solução é calculada. A fim de facilitar o entendimento do pseudo-código, ele conta com abstrações que serão explica-

das a seguir. Este pseudo-código é um recorte da execução para um único indivíduo e não há diferenciação dos blocos de *threads*. Então, como o *kernel* é chamado com 2 dimensões, `threadIdx.y` mapeia exatamente para o índice do cromossomo na população. Enquanto que `threadIdx.x` mapeia para as *threads* disponíveis para o indivíduo utilizar na manipulação dos genes. Na linha 1, a memória compartilhada é declarada com tamanho `blockDim.x`, isto é, cada indivíduo aloca o espaço relativo a quantidade disponível de *threads* que ele terá. Portanto, se há um indivíduo por bloco, a memória compartilhada terá tamanho 1024[1]; se há 32 indivíduos por bloco, cada indivíduo aloca 32 endereços da memória compartilhada. Nas linha 4-7, o laço de repetição faz a acumulação descrita na Figura 4.3. Os endereços $[0, blockDim.x)$ do indivíduo na memória global são acumulados na memória compartilhada, em seguida é a vez dos endereços $[blockDim.x, 2 * blockDim.x)$, e assim sucessivamente até que todos os genes tenham sido devidamente acumulados. Por último é feita a redução paralela no restante dos valores e armazenado no vetor de aptidões da nova população fit_temp.

Algorithm 5 Avaliação de Aptidão de um Cromossomo

1: $shared[blockDim.x]$
2: $offset = 0$
3: $tId \leftarrow threadIdx.x$
4: **repeat**
5: $\quad shared[tId] \leftarrow shared[tId] + f(global[tId + offset])$
6: $\quad offset \leftarrow offset + blockDim.x$
7: **until** $offset = n\_dimensions$
8: $fit\_temp[threadIdx.y] \leftarrow parallel\_reduction(shared, blockDim.x)$

Substituição

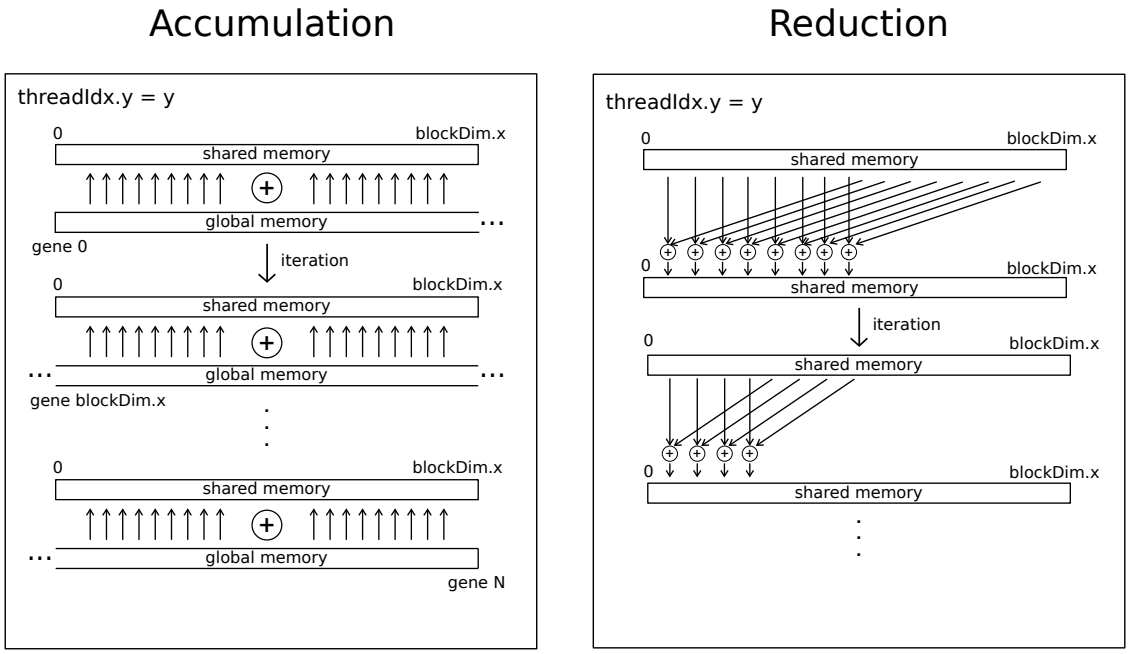
Esse *kernel* implementa uma substituição simples que substitui toda população corrente pelos novos indivíduos gerados. Não foi feita a implementação de torneio ou qualquer método de seleção para escolher os indivíduos que vão para a próxima geração.

A configuração do grid na execução é de *population_size/individuals_p_block* blocos e $n\_dimensions \times individuals\_p\_block$ ou 1024 *threads*. O grid contém blocos unidimensionais.

O Algoritmo 6 apresenta o pseudo-código da implementação do *kernel*. Dessa vez, a presença de outros blocos não foram abstraídos. A variável tid, da linha 1, é o

[1] máximo número de *threads* em um bloco

Figura 4.3: Visualização do algoritmo de acumulação (à esquerda) e algoritmo de redução (à direita)



Fonte: O Autor

identificador global da thread, o que considera todos os blocos no grid. Primeiro, os valores do vetor de aptidão da população original são substituídos pelos valores calculados pelo *kernel* de avaliação. As primeiras $population\_size$ *threads* do grid fazem a substituição desse valores, como mostrado nas linhas 2-4. Em seguida, é feito um laço de repetição para substituir todos os valores do material genético da população. O laço garante que todos os valores do vetor sejam substituídos caso o número de *threads* do grid seja menor que o tamanho do vetor dos dados. O tid de cada thread é incrementado com o tamanho do grid enquanto todos os dados não forem substituídos. Cada thread substitui um gene, porém se o tamanho do grid for menor que o tamanho dos dados, as *threads* podem vir a substituir mais genes.

Algorithm 6 Substituição

1: $tid \leftarrow$ unique thread id ▷ threadIdx.x + blockDim.x*blockIdx.x
2: **if** $tid < population\_size$ **then**
3: $\quad fit[tid] \leftarrow fit\_temp[tid]$
4: **end if**
5: **repeat**
6: $\quad population\_data[tid] \leftarrow population\_data\_temp[tid]$
7: $\quad tid \leftarrow tid + blockDim.x * gridDim.x$
8: **until** $tid \geq length(population\_data)$

4.2 Granularidade em tamanho de *warp*

Os *kernels* baseados em paralelismo em nível de *warp* implementam os mesmos algoritmos da Seção 3.2.1, mas utilizam as diretivas de *warp* ao invés da memória compartilhada sempre que possível nas operações que exigem cooperação. Para que seja possível utilizar apenas os registradores para as operações, é preciso que todos os cromossomos utilizem apenas 32 *threads*. Porém, limitando o número de *threads* para 32 por indivíduo, é esperado que o acesso à memória global fique menos unido e o *kernel* tenha uma execução mais sequencial do que com n-*threads* por indivíduo.

Seleção

O *kernel* implementa o mesmo algoritmo de seleção descrito anteriormente. A diferença é que ele utiliza apenas um bloco de 32 *threads*, permitindo assim a utilização das diretivas de *warp* em toda população. As diretivas são usadas para calcular o máximo, o mínimo e a soma das aptidões de todos os indivíduos da população. Os cálculos são necessários para a normalização dos valores.

A lógica de acumulação da Figura 4.3 foi utilizada nos cálculos de mínimo e máximo para reduzir pop_size valores a 32 para depois fazer uma redução 32 para 1. Pode-se pensar na redução pop_size a 32 como valores organizados em uma matriz$pop_size/32 \times 32$ e os menores/maiores valores de cada coluna são escolhidos em paralelo através de $pop\_size/32$ iterações. O Fragmento de Código 4.2 mostra a implementação do cálculo de mínimo por coluna, nas linhas 1-9, e do mínimo entre os mínimos de cada coluna, nas linhas 11-22.

O cálculo de mínimo começa com a *acumulação* feita através do laço de repetição das linhas 1-9. O de máximo é feito da mesma forma, mas muda a expressão dos condicionais. O laço de repetição incrementa 32 à variável $warp$, que exerce um papel de *offset* no acesso aos dados de aptidão. Logo, o valor de $fit[id\_thread]$ será comparado com $fit[id\_thread + 32]$, $fit[id\_thread + 64]$ e assim sucessivamente, até que todo vetor seja percorrido. Ao terminar o laço, a variável fit_r de cada thread armazena o menor valor da coluna. Como restam 32 valores e todos estão numa thread distinta, é feito uma redução paralela para encontrar o menor valor. A diretiva `shfl_down(var, i)` acessa a variável min da thread $id\_thread + i$. Logo, na linha 13, as primeiras 16 *threads* do grupo vão comparar o seu valor armazenado localmente em min com o min da 16º

thread à sua direita e escolher o menor. A mesma lógica se aplica até o fim da redução, restando o menor valor na variável min da thread de índice 0 do grupo. Por último, cada thread utiliza o método `shfl(var, i)` para acessar o resultado da redução na thread 0 e armazenar seu valor na variável local min_shared.

A soma dos valores de aptidão é feita utilizando a mesma redução descrita no Fragmento de Código 4.4, na sub-seção seguinte, que descreve o *kernel* de avaliação. Para que todas as *threads* recebam o resultado da soma, foi feito um *broadcast* do valor da mesma forma que foi feito na linha 24 do Fragmento de Código 4.2.

Fragmento de Código 4.2 – Aptidão Mínima da População

```
1  for(warp = WARP_SIZE; warp < population; warp += WARP_SIZE){
2     if(group.thread_rank()+warp < population){
3        fit_n = fit[group.thread_rank() + warp];
4
5        if(fit_n < fit_r){
6           fit_r = fit_n;
7        }
8     }
9  }
10
11 min = fit_r;
12
13 fit_n = group.shfl_down(min, 16);
14 min = min < fit_n ? min : fit_n;
15 fit_n = group.shfl_down(min, 8);
16 min = min < fit_n ? min : fit_n;
17 fit_n = group.shfl_down(min, 4);
18 min = min < fit_n ? min : fit_n;
19 fit_n = group.shfl_down(min, 2);
20 min = min < fit_n ? min : fit_n;
21 fit_n = group.shfl_down(min, 1);
22 min = min < fit_n ? min : fit_n;
23
24 min_shared = group.shfl(min, 0);
```

Crossover

O *kernel* utiliza um grid de blocos de *threads* bidimensionais, onde cada sub-bloco ($x = 1...32; y = i$) gera dois indivíduos novos. As 32 *threads* do sub-bloco são agrupadas usando a função `tiled_partition()` da biblioteca de grupos cooperativos.

O Fragmento de Código 4.3 mostra o código implementado no *kernel*. Notar que, diferentemente do *kernel* de n-*threads*, os pais e os pontos utilizados na recombinação são salvos em registradores. Nas linhas 2-5, a primeira thread de cada grupo sorteia os dois pontos para serem usados no cruzamento de dois pontos. Na linha 7 e 8, é possível observar a diretiva `shfl()` sendo usada para fazer um *broadcast* desses valores sorteados para as outras *threads* do grupo. Dessa forma, todas as informações necessárias para fazer o cruzamento ficam salvas nos registradores de cada thread.

Fragmento de Código 4.3 – Crossover com Grupos Cooperativos

```
1 parents_t parents = selected[blockIdx.x*parents_p_block/2 +
     threadIdx.y];
2 if(group.thread_rank() == 0){
3   point1 = rand_integer(0, dimensions, &state[tid_block]);
4   point2 = rand_integer(0, dimensions, &state[tid_block]);
5 }
6
7 point1 = group.shfl(point1, 0);
8 point2 = group.shfl(point2, 0);
```

O algoritmo de recombinação é feito de maneira idêntica a descrita no Algoritmo 3. A diferença é que o tamanho `blockDim.x` é restrito a 32 *threads*. Essa restrição limita o grau de paralelismo do *kernel*, mas permite a cooperação sem o uso da memória compartilhada.

Mutação

O *kernel* de mutação é o mesmo que o descrito anteriormente, mas utilizando 32 *threads* para cada indivíduo.

Avaliação de Aptidão

O *kernel* de avaliação com paralelismo em nível de *warp* realiza as mesmas operações descritas no *kernel* de avaliação com n-*threads*. Porém, ele é projetado esperando blocos bidimensionais de tamanho $x = 32$ e $y = individuals\_p\_block$. Portanto, cada indivíduo de um bloco tem apenas 32 *threads* disponíveis para a computação e são agrupados pelo método de partição `tiled_partition()` da biblioteca de grupos cooperativos.

A computação realizada pelo *kernel* é uma redução (à direita na Figura 4.3). Devido ao número de dimensões do problema poder ser superior a 32, é preciso um mecanismo que permita fazer mais de uma redução. O Fragmento de Código 4.4 mostra como isso é feito sem o uso de memória compartilhada. Um registrador sum é usado como acumulador; apenas a primeira thread do grupo acumula valores nessa variável porque a redução paralela sempre armazena o resultado na primeira thread. Na linha 3, a variável fit_warp recebe o valor $f(x_i)$ para o cálculo do somatório $\sum_{i=0}^{n} f(x_i)$, sendo x_i um gene de algum indivíduo. Nas linhas seguintes é feita a redução utilizando a diretiva `shfl_down()`. No total serão feitas $dimensions/32$ reduções, todas acumuladas na variável sum da primeira thread do grupo.

Fragmento de Código 4.4 – Avaliação de Aptidão

```
1  float sum = 0;
2  for(int warp = 0; warp < dimensions; warp += WARP_SIZE){
3     fit_warp = f(data[dimensions*get_individual_2d(ind_p_block)
          + warp + group.thread_rank()]);
4     fit_warp += group.shfl_down(fit_warp, 16);
5     fit_warp += group.shfl_down(fit_warp, 8);
6     fit_warp += group.shfl_down(fit_warp, 4);
7     fit_warp += group.shfl_down(fit_warp, 2);
8     fit_warp += group.shfl_down(fit_warp, 1);
9
10    if(group.thread_rank() == 0){
11       sum += fit_warp;
12    }
13 }
```

Substituição

Não foi feito um *kernel* de substituição com paralelismo em nível de *warp*.

5 EXPERIMENTOS E RESULTADOS

Para validar os diferentes métodos de programação e paralelismo propostos neste trabalho, o problema de One-Max foi abordado, descrito na Equação 5.1:

$$\text{Minimize: } f(x_1, x_2, ..., x_n) = \sum_{i=1}^{n}(x_i) \tag{5.1}$$

O domínio do problema foi restrito a $x_i \in \{0, 1\}$ para torná-lo um problema binário. O n do somatório é o tamanho da *string* de 1s e 0s. A solução do problema é $f(x_1, x_2, ..., x_n) = 0$, portanto basta maximizar o número de 0s na string. É um problema aparentemente simples para humanos, já que basta que minimize a soma colocando zeros. Porém, um computador não sabe de ante-mão isso e precisaria testar 2^n soluções em um algoritmo de força-bruta para ter certeza de que encontrou a solução com menor soma.

Como a intenção do trabalho é verificar desempenho de algoritmos genéticos em GPU para diferentes formas de paralelismo, os experimentos consistem na variação nos parâmetros de execução: dimensão do problema, população do algoritmo genético e número de indivíduos por bloco.

O algoritmo genético executa com 80% de probabilidade de *crossover* e 0,01% de probabilidade mutação nos genes (mutação de 1 gene a cada 10000), definidos empiricamente. O número de gerações foi fixado em 2000. As implementações propostas foram testadas para o OneMax com 32, 128, 256, 512 e 1024 dimensões (tamanho de *string*). Para cada dimensão utilizada, uma população de tamanho 32, 64 e 128. Todas essas variações nos parâmetros foram executadas utilizando 1, 2, 4, 8, 16 e 32 indivíduos por bloco a fim de verificar a relação entre desempenho, número de blocos despachados e volume de computação por bloco.

Este trabalho verifica as implementações já apresentadas anteriormente, contemplando 2 níveis de paralelismo em GPU: paralelismo por indivíduo e paralelismo por dimensão. O primeiro foi feito transformando a implementação de n-*threads* em *1-thread*, utilizando blocos bidimensionais de tamanho 1 na dimensão x e $individuals\_per\_block$ na dimensão y. Essa modificação faz com que cada thread seja um indivíduo que executa um algoritmo sequencial. Já o paralelismo por dimensão aparece em diferentes granularidades com os *kernels* de n-*threads* e de *Warp*. Portanto, são 3 implementações distintas avaliadas: 1-thread, n-*threads* e *Warp*.

O desempenho é avaliado com o cálculo de *speedup* em relação a uma implemen-

tação *baseline* em CPU utilizando a Equação 5.2 (HENNESSY; PATTERSON, 2017, p. 49-50), onde t é o tempo de execução médio da aplicação. Essa implementação utiliza a versão sequencial do mesmo algoritmo genético implementado em GPU. Todas medidas foram computadas, para cada configuração e implementação, com base em 30 execuções do programa. A média e o desvio padrão do tempo de execução para cada implementação são encontrados nas tabelas do Apêndice B.

$$Speedup = \frac{t_{cpu}}{t_{gpu}} \tag{5.2}$$

5.1 Equipamentos Utilizados

Em todos os testes foram utilizados a GPU Titan V da NVIDIA. A CPU utilizada como *baseline* é a Intel Xeon E5-2407. Abaixo, na Tabela 5.1, são listadas as especificações da placa de vídeo utilizada.

Tabela 5.1: GPUs

GPU	*Titan V*
Architecture	Volta
Compute Capability	7.0
SM Count	80
FP32 Cores / SM	64
FP32 Cores / GPU	5120
FP64 Cores / SM	32
FP64 Cores / GPU	2560
Memory Size	12GB
Memory Clock	850 MHz
Memory Clock (effective)	1700 MHz
Total Memory Bandwidth	652.8 GB/s
Maximum amount of Shared Memory / SM	96KB
Maximum amount of Shared Memory / Thread Block	96KB
Register File Size / SM	256 KB
Register File Size / GPU	20480KB
Ratio of SM registers to FP32 Cores	1024
L2 Cache Size	6144KB
L1 Cache Size / SM	Up to 96 KB
Base Clock	1200 MHz

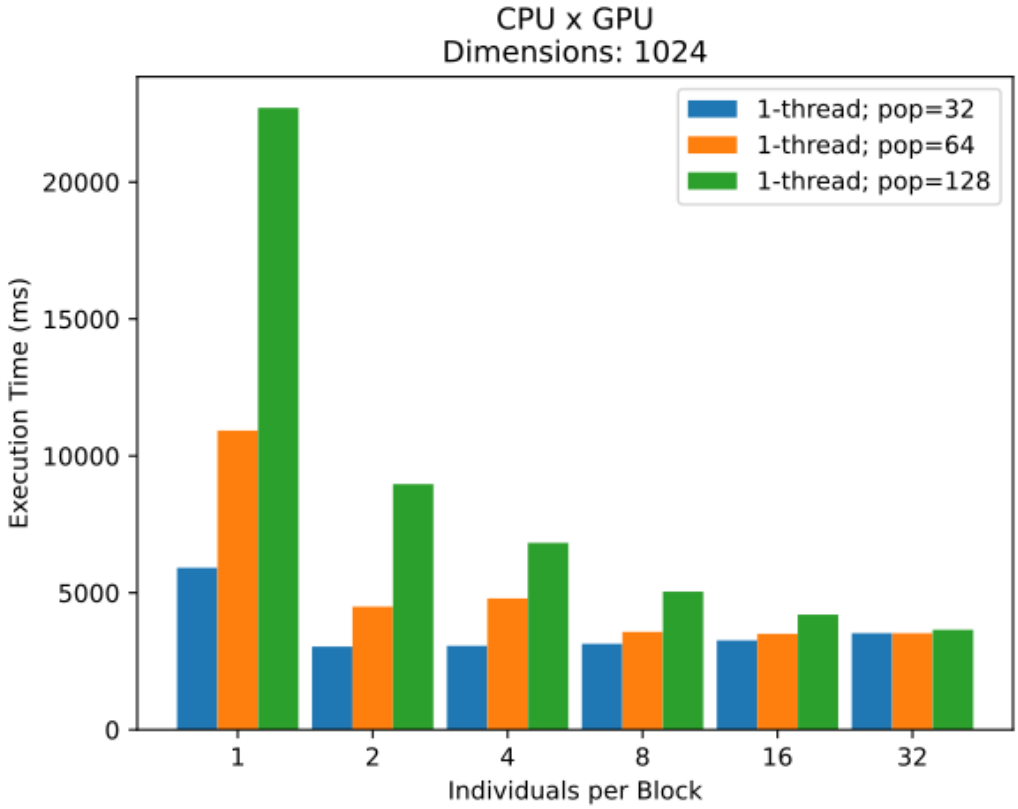
5.2 Resultados Paralelismo por Indivíduo

Nesta Seção, os resultados dos experimentos baseados em paralelismo por indivíduo são apresentados e analisados. Essa metodologia de implementação de algoritmos em GPU é comumente encontrado em trabalhos similares da literatura, principalmente nos anos iniciais da pesquisa de implementação de AGs em GPUs por ser uma adaptação direta de algoritmos feitos para execução sequencial em CPU. Portanto, é uma análise interessante para se entender melhor quando esse tipo de implementação é ou não válida.

O número de indivíduos computados em um bloco afeta diretamente o desempenho. A Figura 5.1 mostra o tempo de execução da implementação 1-thread para resolver um problema de tamanho fixo de 1024 dimensões. Cada barra representa o mesmo algoritmo efetuado com populações distintas: 32, 64 e 128. O número de indivíduos por bloco é deixado livre para se fazer a comparação em termos dessa variável. O número de blocos do grid é calculado por $population/individuals\_p\_block$, então, para o caso de um indivíduo por bloco e 128 soluções avaliadas, se tem um grid com 128 blocos. Esse é o único caso que há uma fila de execução para os blocos, já que a Titan V conta com 80 SMs e nenhuma outra configuração lança mais blocos que esse número. Com relação aos resultados obtidos, é possível ver que aumentar o número de *threads* por bloco não melhora, necessariamente, o desempenho para esse algoritmo.

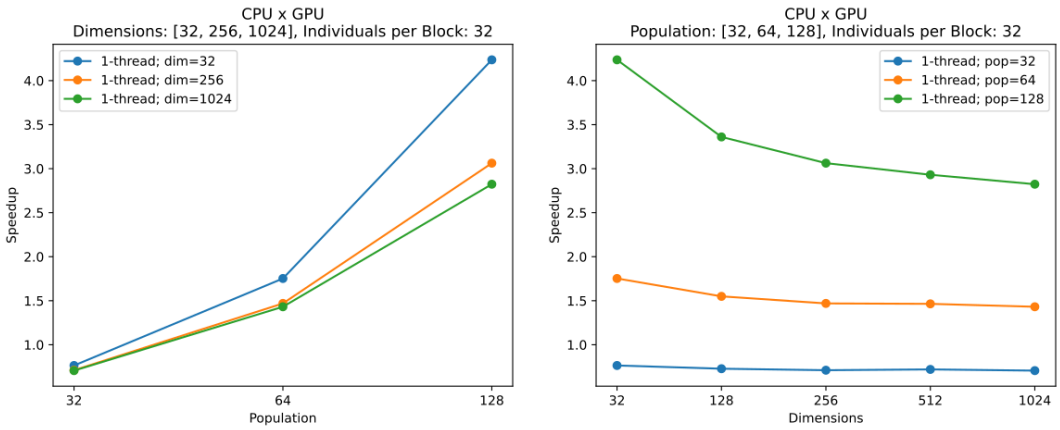
A implementação utilizando paralelismo por indivíduo tem a característica de escalar bem com o aumento da população utilizada, mas escalar mal com o aumento do tamanho do problema, pois aumenta o trabalho computacional executado por cada thread. A Figura 5.2 mostra a relação entre o *speedup* e variação da população e tamanho do problema. O número de indivíduos por bloco é 32 em todas execuções porque obteve bons resultados para todas configurações (como mostrado na Figura 5.1), mas esse parâmetro não é relevante nessa comparação. No gráfico à esquerda, as dimensões do problema são fixas em 32, 256 e 1024, enquanto a população varia. Claramente o *speedup* é proporcional ao número de indivíduos presentes na execução, tendo um maior aumento quando se tem problemas menores. À direita, o inverso ocorre: a população é fixada em 32, 64 e 128 enquanto o tamanho do problema varia. Nesse caso, o tamanho do problema prejudica o *Speedup* porque as unidades de execução e de controle das CPUs são muito mais complexas que as presentes em GPU. Portanto, aumentar a carga de processamento em cada thread, fazendo com que elas executem mais código sequencial, tende a reduzir o *Speedup* em relação a CPU.

Figura 5.1: Comparação do tempo de execução da implementação *1-thread* para populações de tamanhos distintos variando o número de indivíduos por bloco - 30 execuções



Fonte: Próprio Autor

Figura 5.2: Comparação de Speedup da implementação *1-thread* em relação ao algoritmo em CPU em função da população (à esquerda) e do número de dimensões do problema (à direita)- 30 execuções



Fonte: O Autor

5.3 Resultados Paralelismo por Dimensão

Já foi verificado por outros trabalhos que o tipo de implementação que aproveita melhor a arquitetura das GPUs são as que exploram o paralelismo por dimensão. Isso se deve tanto à capacidade de distribuir melhor a computação entre as *threads*, como a possibilidade de projetar algoritmos de redução que reduzem a complexidade de laços de repetição através do paralelismo. Os resultados apresentados nesta seção comparam as diferentes implementações desenvolvidas ao longo da monografia, além de analisar como o *Speedup* se comporta com a variação dos parâmetros analisados.

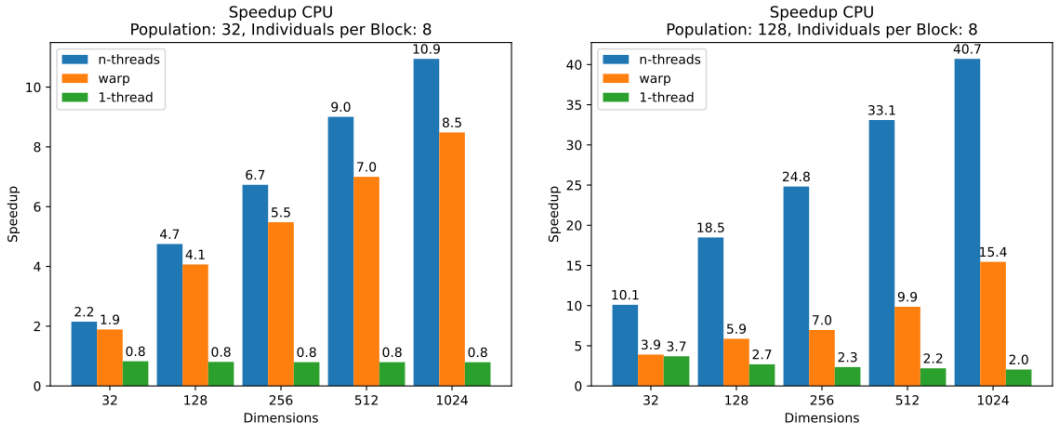
5.3.1 Influência do Tamanho do Problema e da População no *Speedup*

A ideia principal do paralelismo por dimensão é fazer com que cada thread seja responsável, idealmente, por um gene do indivíduo. Devido a isso, o aumento no tamanho do problema faz com que o tempo de execução do algoritmo em GPU cresça menos do que a implementação em CPU, o que resulta na melhora da métrica de *Speedup*.

Na Figura 5.3 é possível observar como o *Speedup* escala proporcionalmente ao tamanho do problema. Todos experimentos utilizaram 8 indivíduos por bloco. O gráfico da esquerda fixa a população em 32 indivíduos enquanto o da direita fixa em 128. É possível visualizar que a métrica escala também com a população. Por exemplo, a versão n*-threads* obteve um *Speedup* de 10,9x com uma população de 32 e 1024 dimensões; quadruplicando a população e mantendo o número de dimensões, o *Speedup* foi de 40,7x. Uma diferença de quase 4x para um número de cromossomos na população 4x maior. Essa proporção se mantém mais ou menos a mesma para todas as dimensões testadas. Já na versão de Grupos Cooperativos, limitando o paralelismo ao tamanho de um *warp*, a proporção dessa métrica em relação ao aumento do número de indivíduos se mostrou irregular, mas ainda assim escalando bem. Como já foi mostrado anteriormente, a implementação paralela por indivíduo também escala com a população, mas se torna menos eficiente com o aumento do número de dimensões. De outra sorte, os resultados para as implementações paralelas por dimensão confirmam que elas se tornam mais eficientes, em relação a CPU, a medida que cresce o tamanho do problema. Os resultados mostram um aumento de 4x no número de dimensões faz com que o *Speedup* aumente aproximadamente 2x para a versão n*-threads* com 8 indivíduos por bloco.

O número de indivíduos por bloco foi fixado em 8 nos experimentos apresentados

Figura 5.3: Comparação do Speedup das diferentes implementações em relação à CPU para uma execução com tamanho de população 32 (à esquerda) e 128 (à direita) variando o número de dimensões - 30 execuções



Fonte: O Autor

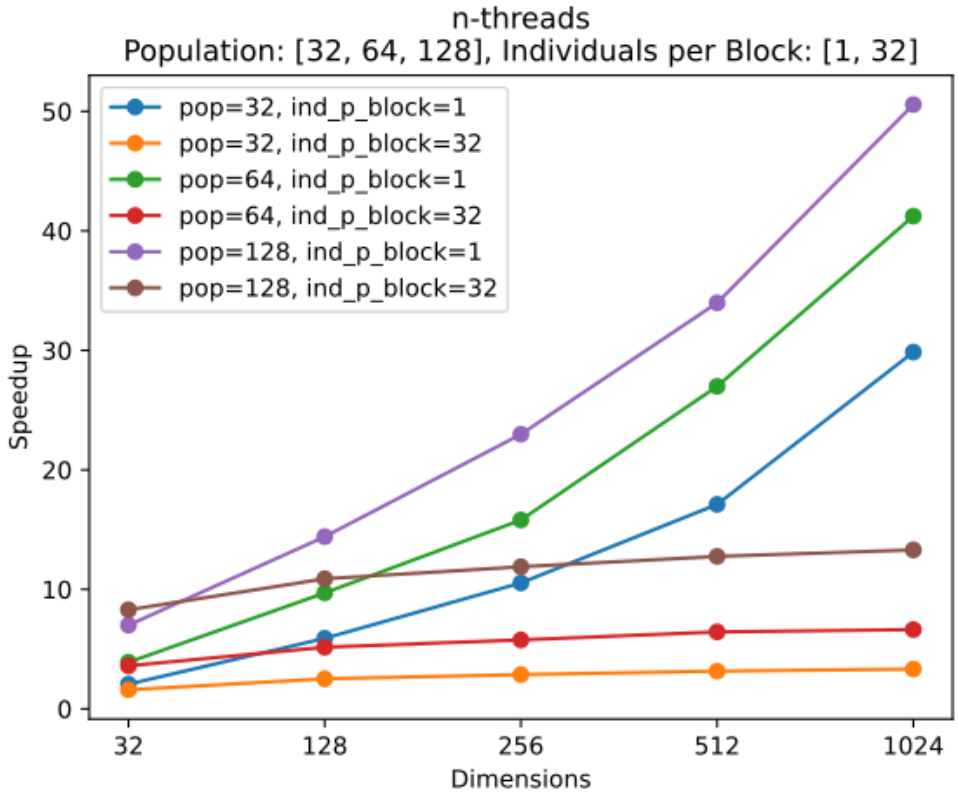
na Figura 5.3, mas para a implementação n-*threads* esse parâmetro influencia diretamente como o *Speedup* escala em função dos outros parâmetros, pois afeta a granularidade do paralelismo da execução.

5.3.2 Influência do Número de Indivíduos por Bloco no *Speedup*

A configuração de lançamento dos *kernels*, através da modificação da estrutura do grid, afeta diretamente a granularidade do paralelismo da execução. Nas implementações apresentadas, a modificação da estrutura do grid é modificada através do número de indivíduos computados em um bloco. Esse parâmetro afeta, portanto, diretamente o número de blocos utilizados na execução e o número de *threads* utilizadas nas operações genéticas de um indivíduos, sendo um parâmetro de fundamental importância para a eficiência da computação.

Na Figura 5.4 é apresentada uma comparação da relação *Speedup* e dimensões do problema entre configurações de execução distintas. São 6 configurações que abrangem o número máximo (32) e mínimo (1) de indivíduos por bloco e todos tamanhos de população testados nessa monografia. É possível ver que o *Speedup* obtido na versão de 32 indivíduos chegou em um limite e melhora muito pouco com o aumento do número de dimensões. Nessa versão são utilizadas apenas 32 *threads* para as operações genéticas de um indivíduo, logo um aumento no número de dimensões aumenta o trabalho computacional de cada thread. Em contraste, a versão com apenas um indivíduo por bloco obtém melhores Speedups a medida que o número de dimensões do problema aumenta.

Figura 5.4: Comparação do *Speedup* da implementação n-*threads* em relação à CPU para diferentes tamanhos de população e indivíduos por bloco variando o número de dimensões - 30 execuções

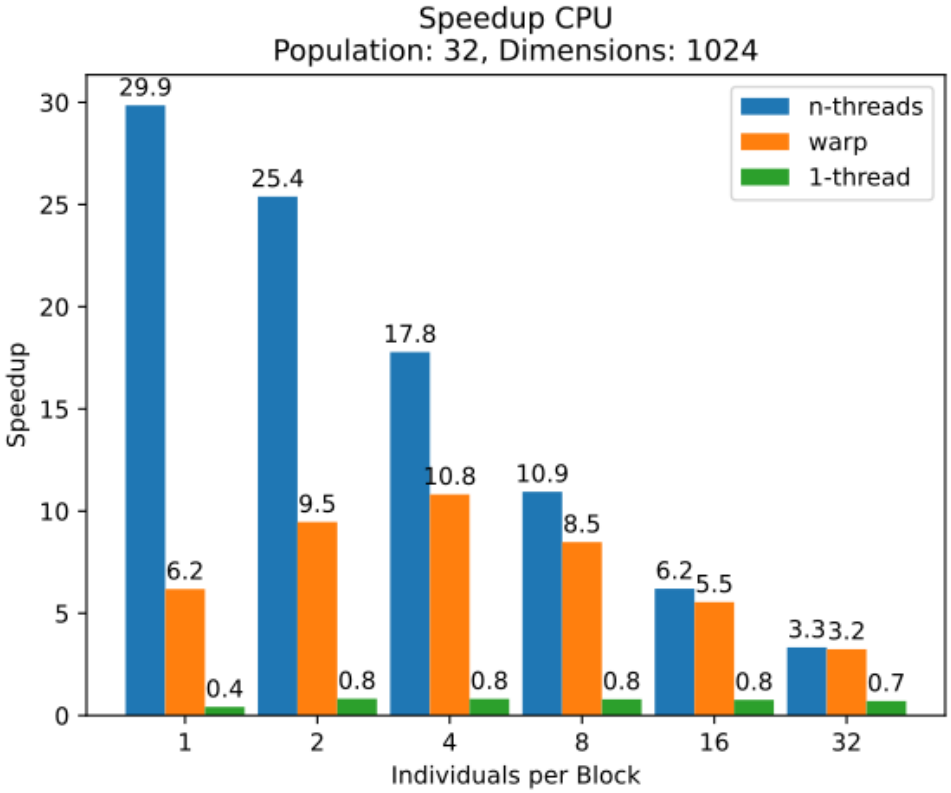


Fonte: O Autor

Na Figura 5.4, é possível analisar como as diferentes configurações de execução escalam com o tamanho da população através da distância entre as curvas de mesmo número de soluções por bloco. A versão de 32 indivíduos escalou melhor com o tamanho da população que a versão de um indivíduo por bloco. No caso, observa-se que há uma distância linear entre as linhas referentes à execução de um indivíduo por bloco, enquanto a distância entre as linhas da execução com 32 indivíduos por bloco aumenta progressivamente com o aumento do tamanho da população. Portanto, há a evidência de um *trade-off* de desempenho entre aumentar a carga computacional por bloco a fim de diminuir o número de blocos despachados e aumentar o número de blocos despachados para diminuir a carga computacional em cada um.

Ao fixar o tamanho da população em 32 e dimensões em 1024, então variar o número de indivíduos por bloco, as 1024 *threads* no bloco serão utilizadas em todas configurações. Além disso, o grid inteiro sendo executado de uma vez só, já que há 80 SMs disponíves na Titan V e o máximo de blocos despachados é 32 (colocando um indivíduo por bloco). Então, a medida que se põe mais soluções computadas em um bloco, ocorre

Figura 5.5: Comparação do *Speedup* das diferentes implementações em relação à CPU executando um algoritmo com tamanho de população 32 e 1024 dimensões no problema; variando o número de indivíduos por bloco - 30 execuções

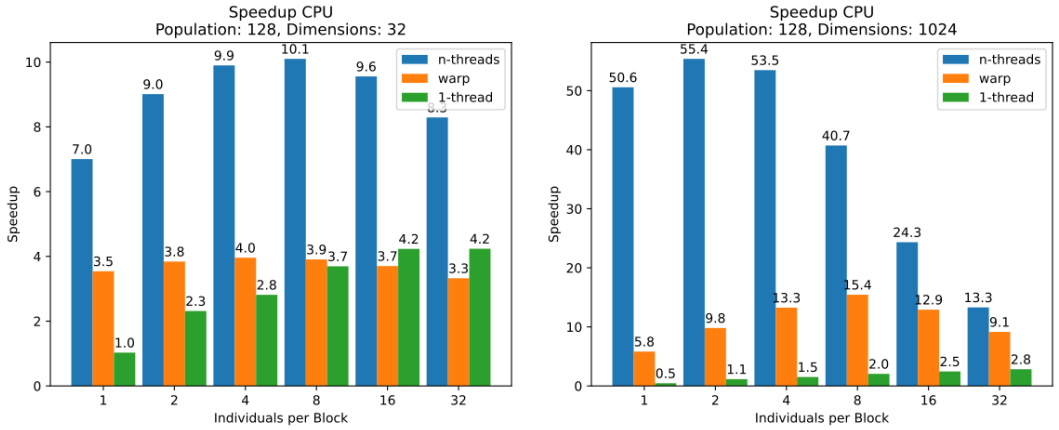


Fonte: O Autor

um acréscimo de computação sequencial por thread sem alguma melhora na ocupação da SM e no número de blocos executados simultaneamente. Dessa forma, o *Speedup* se torna inversamente proporcional ao número de indivíduos por bloco utilizado, como mostra a figura 5.5. Nota-se a diferença de 9x entre o *Speedup* utilizando 32 indivíduos e 1 indivíduo por bloco. Na implementação baseada em *warps*, a melhor eficiência foi obtida utilizando 4 indivíduos por bloco, ou 4 *warps* (128 *threads*). Acima dessa quantidade, uma maior ocupação do bloco não melhora a eficiência da execução.

Na Figura 5.6 há também uma comparação das diferentes implementações variando o número de indivíduos por bloco e mantendo fixa a população. Todavia, há dois gráficos distintos que mostram os resultados da execução para problemas de 32 e 1024 dimensões. É interessante perceber que, diferentemente da comparação anterior, o aumento no número de soluções em um bloco não piora, necessariamente, o *Speedup* da implementação n-*threads*. Não só isso, como há uma pequena melhora de desempenho quando se diminui o número de blocos utilizados e se aumenta o trabalho realizado em cada bloco para alguns casos. Por exemplo, no gráfico da esquerda temos um problema pequeno que

Figura 5.6: Comparação do *Speedup* das diferentes implementações em relação à CPU variando o número de indivíduos por bloco em uma execução com população de tamanho 128 e número de dimensões 32 (à esquerda) e 1024 (à direita) - 30 execuções



Fonte: O Autor

utiliza apenas 32 *threads* por indivíduo, ocupando todas as 1024 *threads* apenas quando se coloca 32 indivíduos por bloco. Portanto, um aumento no número de soluções computadas em um bloco, tem com consequência uma melhor ocupação do SM. Nesse mesmo gráfico, é possível ver que o pico de *Speedup* é quando se coloca 8 indivíduos por bloco, ou se utiliza 256 *threads* por bloco. Considerando que cada *Streaming Multiprocessor* da Titan V é dividido em 4 unidades de execução, tendo um escalonador de *warp* em cada uma e, portanto, cada SM sendo capaz de despachar 4 *warps* por ciclo de clock, é esperado que os melhores Speedups estejam por essa ordem de indivíduos por bloco. Conforme pode ser observado na Figura 5.6, para a implementação baseada em *Warps*, variar esse parâmetro não altera de maneira significativa o *Speedup* para o One-Max de 32 dimensões.

5.4 Resultados da Granularidade de Tamanho de *Warp*

Os resultados da implementação do algoritmo genético com granularidade em nível de *warp* foram significativamente piores que os da implementação n-*threads*. Então, foi utilizada a ferramenta `nvprof` (NVIDIA, 2021c), disponibilizada pela NVIDIA, para fazer uma análise mais detalhada dos resultados. Com essa ferramenta é possível coletar informações de atividade da aplicação CUDA, como tempo execução de cada *kernel*, quantas vezes cada *kernel* foi chamado e informações referentes transferência de memória.

Os resultados de desempenho de cada *kernel* das implementações de *warp* e n-

threads são apresentados na Tabela 5.2. Os parâmetros de execução da aplicação analisada pelo `nvprof` foram uma população de tamanho 128, problema de 1024 dimensões e 8 indivíduos por bloco.

Tabela 5.2: Medidas de tempo de execução pelo *nv-profile* para uma execução com população de tamanho 128, problema de 1024 dimensões e 8 indivíduos por bloco

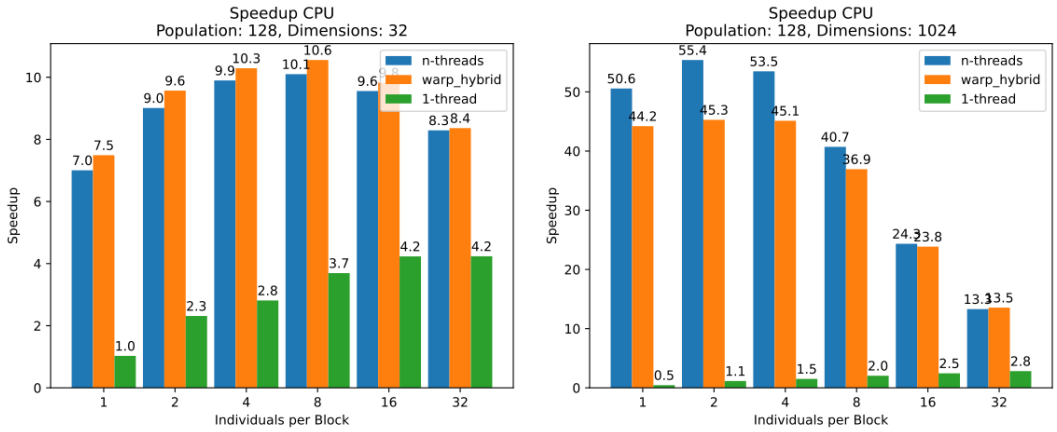
Kernel	Tempo de Execução Médio (60000 chamadas)	% Tempo de Execução da Aplicação
`selection_warp`	197,49us	60,74%
`mutation_warp`	69,13us	21.26%
`crossover_warp`	43,22us	13,29%
`evaluation_warp`	9,23us	2,84%
`selection_nthreads`	17,19us	14,67%
`crossover_nthreads`	33,09us	28,24%
`mutation_nthreads`	54,83us	46,79%
`evaluation_nthreads`	5.99us	5,12%
`replacement_nthreads`	3.16us	2,69%

O *kernel* de seleção da implementação de *warp* `selection_warp` é um gargalo na execução, representando 60,74% do tempo da aplicação. Em comparação com o *kernel* da implementação de n*-threads* `selection_dim`, percebe-se que o `selection_warp` é uma implementação ineficiente. Então, foi analisada uma nova implementação que substitui o *kernel* `selection_warp` pelo `selection_nthreads` na implementação de *warp*. Nessa nova implementação, o *kernel* de mutação também foi substituído por não utilizar as diretivas de *warp* e não ter apresentado nenhum ganho em relação ao *kernel* de mutação n*-threads*. Portanto, a nova implementação híbrida utiliza os *kernels*:

- `selection_nthreads`
- `crossover_warp`
- `mutation_nthreads`
- `evaluation_warp`
- `replacement_nthreads`

A Figura 5.7 apresenta os resultados para os mesmos testes mostrados na Figura 5.6. O *Speedup* adquirido pela implementação híbrida é comparável à versão n*-threads*, obtendo vantagem com problemas menores e blocos com mais indivíduos. O gráfico à esquerda executa com 32 dimensões; os resultados são semelhantes (mas melhores) na implementação híbrida para qualquer número de indivíduos por bloco. É um resultado esperado visto que o problema de 32 dimensões consegue ser computado em paralelo por

Figura 5.7: Comparação do *Speedup* entre a implementação *warp* híbrida e as outras implementações em relação à CPU variando o número de indivíduos por bloco em uma execução com população de tamanho 128 e número de dimensões 32 (à esquerda) e 1024 (à direita) - 30 execuções



Fonte: O Autor

um único *warp*. À direita, com 1024 dimensões, o resultado da implementação *warp* híbrida é pior que o da n-*threads* quando há menos indivíduos por blocos, pois há mais *threads* disponíveis por indivíduo para os *kernels* n-*threads* usufruirem.

Por fim, a Tabela 5.3 apresenta os resultados de *Speedup* para todos os experimentos realizados, com exceção dos experimentos relacionados com a execução de 1-thread. Nessa Tabela, é possível observar o *Speedup* máximo observado para todas as implementações: $55,4\times$ para n-*threads*, $45,3\times$ para a implementação *warp* modificada e apenas $15,4\times$ para a implementação *warp* original. Além disso, os resultados mostram que, nas implementações n-*threads* e *warp* modificada, há uma tendência de crescimento em *Speedup* de aproximadamente $2\times$ para um aumento no tamanho da população de $2\times$ com todas as dimensões testadas para uma execução com 8, 16 ou 32 indivíduos por bloco. Para o One-Max de 32 dimensões, esse crescimento em *Speedup* ocorre também para 2 e 4 indivíduos por bloco. Portanto, à medida que o tamanho do problema cresce, em termos de dimensões, é observada uma queda na taxa de crescimento de *Speedup* em função do tamanho da população. Menores números de soluções computadas por bloco obtém melhor desempenho em problemas de dimensões maiores, porém crescem menos com o aumento no tamanho da população em relação à maiores soluções computadas em um bloco.

Na Tabela 5.3 estão destacadas as implementações com melhores *Speedups* para cada número de dimensões do problema e tamanho da população. Por exemplo, para um tamanho de população 128 e número de dimensões 32, o melhor *Speedup* obtido foi o de $10,6$ da implementação warp modificada com 8 indivíduos por bloco.

Como os resultados demonstram que há espaço para que mais indivíduos por bloco tenham *Speedup* melhores com o aumento do tamanho da população, foi testado para as implementações de n-*threads* e *warp* modificada a execução do problema utilizando 256 e 512 como tamanho da população. Os resultados estão na Tabela 6.5 encontrada no Apêndice A.

Tabela 5.3: Resultados de *Speedup* para todas implementações testadas

Parâmetros		n-threads			Warp Modificada			Warp		
Dim.	Ind. p/ Bloco	População			População			População		
		32	64	128	32	64	128	32	64	128
32	1	2,1	3,9	7,0	2,1	4,1	7,5	1,8	3,1	3,5
32	2	**2,3**	4,5	9,0	**2,3**	4,7	9,6	2,0	3,4	3,8
32	4	2,2	4,6	9,9	**2,3**	**4,9**	10,3	2,0	3,5	4,0
32	8	2,2	4,6	10,1	2,2	4,8	**10,6**	1,9	3,4	3,9
32	16	1,9	4,2	9,6	2,0	4,4	9,8	1,7	3,2	3,7
32	32	1,6	3,6	8,3	1,7	3,6	8,4	1,4	2,7	3,3
128	1	5,9	9,7	14,4	5,6	9,5	15,3	3,6	4,9	4,8
128	2	**6,2**	**11,0**	17,7	5,9	10,7	17,8	4,4	6,3	5,2
128	4	5,7	10,7	18,5	5,6	10,7	**18,9**	4,6	6,9	5,7
128	8	4,7	9,4	18,5	4,8	9,4	18,4	4,1	6,7	5,9
128	16	3,7	7,3	14,9	3,7	7,5	15,2	3,2	5,6	5,8
128	32	2,5	5,2	10,9	2,6	5,3	11,1	2,3	4,1	4,8
256	1	**10,5**	15,8	23,0	8,8	14,7	23,4	4,6	5,5	5,0
256	2	10,2	**16,8**	26,7	9,0	15,6	25,4	6,0	7,8	5,8
256	4	8,7	16,3	**28,4**	8,1	15,3	27,1	6,4	8,8	6,6
256	8	6,7	13,1	24,8	6,5	12,6	24,3	5,5	8,6	7,0
256	16	4,7	9,3	18,4	4,7	9,3	18,5	4,1	7,0	6,9
256	32	2,9	5,8	11,9	2,9	5,9	12,1	2,6	4,8	5,7
512	1	**17,1**	27,0	34,0	13,3	23,0	32,6	5,5	6,3	5,0
512	2	16,8	**29,0**	**40,2**	13,2	23,6	35,8	7,8	9,5	7,4
512	4	13,0	24,6	39,9	11,1	21,7	35,9	8,5	11,7	9,0
512	8	9,0	17,8	33,1	8,4	16,7	31,1	7,0	11,5	9,9
512	16	5,6	11,2	21,8	5,5	11,1	21,6	4,9	8,6	8,9
512	32	3,2	6,4	12,8	3,2	6,6	13,0	3,0	5,6	7,1
1024	1	**29,9**	**41,2**	50,6	19,1	31,0	44,2	6,2	6,9	5,8
1024	2	25,4	**41,2**	**55,4**	17,5	29,8	45,3	9,5	10,8	9,8
1024	4	17,8	33,4	53,5	14,0	26,7	45,1	10,8	14,9	13,3
1024	8	10,9	21,4	40,7	9,9	19,3	36,9	8,5	14,6	15,4
1024	16	6,2	12,3	24,3	6,1	12,1	23,8	5,5	10,1	12,9
1024	32	3,3	6,6	13,3	3,4	6,8	13,5	3,2	6,1	9,1

6 CONCLUSÃO

Neste trabalho foram apresentadas implementações de AGs em CUDA que exploram diferentes paralelismo em diferentes granularidades. Os resultados obtidos demonstram que a implementação com paralelismo por indivíduo não explora devidamente o potencial computacional das GPUs, já que a implementação com paralelismo por dimensão escala melhor tanto com o aumento da população, tanto com o aumento do tamanho do problema.

Foi analisado também como o número de soluções computadas em um bloco afeta a eficiência da computação. Os resultados mostram que é preferível, em termos de desempenho, utilizar mais blocos para a computação mesmo que isso tenha como consequência um menor número de *threads* por bloco. Ainda assim, é possível obter melhores resultados ao aumentar o número de indivíduos computados em um bloco a fim de diminuir quantos são despachados durante a execução. Nas implementações apresentadas, o ganho de desempenho relacionado ao número de blocos despachados está atrelado ao tamanho da população utilizada pelo algoritmo genético. Em populações maiores e problemas com dimensões menores, foi observado ganhos de desempenho melhores quando se põe mais indivíduos em um bloco. Porém, um aumento excessivo nesse parâmetro torna a computação menos eficiente por aumentar o trabalho realizado por cada thread, e as execuções com menos indivíduos por bloco se sobressaem como as de melhores desempenho.

Além disso, foi proposto um método de explorar o paralelismo por dimensão sem recorrer à memória compartilhada através do uso das funções de *Warp-Shuffle*. Essa implementação é mais vantajosa para problemas menores, mas escala pior com o tamanho do problema do que a implementação que utiliza a memória compartilhada para cooperação.

A implementação proposta para o *kernel* de seleção por roleta utilizando as funções de *Warp-Shuffle* se mostrou ineficiente em relação ao modelo com memória compartilhada. É preciso uma análise mais detalhada da execução para se entender o motivo do má desempenho obtida.

Uma direção para trabalhos futuros pode ser a criação de *kernels* híbridos que exploram as vantagens de ambas implementações. Como as implementações que utilizam as funções de *Warp-Shuffle* são mais rápidas para problemas com menos dimensões, mas limitam a computação de uma solução a um *warp*, elas podem ser utilizadas em conjunto com a memória compartilhada para que múltiplos *warps* computem uma única solução.

REFERÊNCIAS

ARORA, R.; TULSHYAN, R.; DEB, K. Parallelization of binary and real-coded genetic algorithms on gpu using cuda. In: IEEE CONGRESS ON EVOLUTIONARY COMPUTATION, 2010, Barcelona, Spain. **IEEE Congress on Evolutionary Computation**. Barcelona, Spain: IEEE, 2010. p. 1–8.

BARTZ-BEIELSTEIN, T. et al. Evolutionary algorithms. **WIREs Data Mining and Knowledge Discovery**, v. 4, n. 3, p. 178–195, 2014.

BEYER, H.-G.; SCHWEFEL, H.-P. Evolution strategies –a comprehensive introduction. **Natural Computing: An International Journal**, Kluwer Academic Publishers, USA, v. 1, n. 1, p. 3–52, may 2002.

BOIANI, M.; PARPINELLI, R. S. A gpu-based hybrid jde algorithm applied to the 3d-ab protein structure prediction. **Swarm and Evolutionary Computation**, v. 58, p. 100711, 2020. ISSN 2210-6502.

CALÉGARI, P. R. **Parallelization of Population-based Evolutionary Algorithms for Combinatorial Optimization Problems**. Thesis (PhD), 1999. Lausanne, EPFL.

CHE, S. et al. A performance study of general-purpose applications on graphics processors using cuda. **Journal of Parallel and Distributed Computing**, v. 68, n. 10, p. 1370–1380, 2008. General-Purpose Processing using Graphics Processing Units.

CHENG, J. R.; GEN, M. Accelerating genetic algorithms with gpu computing: A selective overview. **Computers Industrial Engineering**, v. 128, p. 514–525, 2019.

CHENG MAX GROSSMAN, T. M. J. **Professional CUDA C Programming**. 1st. ed. GBR: Wrox Press Ltd., 2014.

CHOPARD, B.; TOMASSINI, M. **An Introduction to Metaheuristics for Optimization**. 1st. ed. Switzerland: Springer Publishing Company, Incorporated, 2018.

DARWIN, C. et al. **On the origin of species by means of natural selection, or, The preservation of favoured races in the struggle for life**. Albemarle Street, London: John Murray, 1872.

Diaz, J.; Muñoz-Caro, C.; Niño, A. A survey of parallel programming models and tools in the multi and many-core era. **IEEE Transactions on Parallel and Distributed Systems**, v. 23, n. 8, p. 1369–1386, 2012.

FLYNN, M. Very high-speed computing systems. **Proceedings of the IEEE**, IEEE, v. 54, n. 12, p. 1901–1909, 1966.

GARLAND, M. et al. Parallel computing experiences with cuda. **IEEE Micro**, IEEE, v. 28, n. 4, p. 13–27, 2008.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Sixth Edition: A Quantitative Approach**. 6th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

HOLLAND, J. H. Genetic Algorithms. **Scientific american**, JSTOR, v. 267, n. 1, p. 66 – 73, 1992.

JAROS, J. Multi-gpu island-based genetic algorithm for solving the knapsack problem. In: IEEE CONGRESS ON EVOLUTIONARY COMPUTATION. Brisbane, QLD, Australia: IEEE, 2012. p. 1–8.

LUKE, S. **Essentials of Metaheuristics**. second. USA: Lulu, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

LUONG, T. V. **Métaheuristiques parallèles sur GPU**. Thesis (PhD), 2011. Thèse de doctorat dirigée par Melab, Nouredine et Talbi, El-Ghazali Informatique Lille 1 2011.

LUONG, T. V.; MELAB, N.; TALBI, E.-G. GPU-based Island Model for Evolutionary Algorithms. In: **Genetic and Evolutionary Computation Conference (GECCO)**. Portland, United States: HAL, 2010.

MUHLENBEIN, H.; SCHLIERKAMP-VOOSEN, D. Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. 1993.

MUSSI, L. Implementation of a simple genetic algorithm within the cuda architecture. In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE (GECCO). Montreal, Canada, 2009.

NVIDIA. **Best Practices Guide**. 2021. Available from Internet: <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>.

NVIDIA. **CUDA C Programming Guide**. 2021. Available from Internet: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>.

NVIDIA. **Profiler User Guide**. 2021. Available from Internet: <https://docs.nvidia.com/cuda/profiler-users-guide/index.html>.

RASHID, H.; NOVOA, C.; QASEM, A. An evaluation of parallel knapsack algorithms on multicore architectures. In: ARABNIA, H. R. et al. (Ed.). **Proceedings of the 2010 International Conference on Scientific Computing, CSC 2010, July 12-15, 2010**. Las Vegas, Nevada, USA: CSREA Press, 2010. p. 230–235.

SANDERS, J.; KANDROT, E. **CUDA by Example: An Introduction to General-Purpose GPU Programming**. 1st. ed. USA: Addison-Wesley Professional, 2010.

SCHMIDT, B.; HUNDT, C. cudtw++: Ultra-fast dynamic time warping on cuda-enabled gpus. In: EUROPEAN CONFERENCE ON PARALLEL PROCESSING. **Euro-Par 2020: Parallel Processing**. Warsaw, Poland: Springer, 2020. p. 597–612.

SÖRENSEN, K.; GLOVER, F. W. Metaheuristics. In: GASS, S. I.; FU, M. C. (Ed.). **Encyclopedia of Operations Research and Management Science**. Boston, MA: Springer US, 2013. p. 960–970.

TALBI, E.-G. **Metaheuristics: From Design to Implementation**. Hoboken, New Jersey, US: Wiley Publishing, 2009.

TALBI, E.-G. Parallel evolutionary combinatorial optimization. In: KACPRZYK, J.; PEDRYCZ, W. (Ed.). **Springer Handbook of Computational Intelligence**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, (Springer Handbooks). p. 1107–1125.

TANG, K. et al. Genetic algorithms and their applications. **IEEE Signal Processing Magazine**, v. 13, n. 6, p. 22–37, 1996.

TARDOS, J. K. E. **Algorithm Design**. USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

APÊNDICE A - MATERIAL SUPLEMENTAR

Tabela 6.1: Resultados de *Speedup* com experimentos realizados também para tamanhos de população de 256 e 512.

Parâmetros		n-threads					Warp Modificada				
Prob. Size	Ind. p/ Blk.	Population					Population				
		32	64	128	256	512	32	64	128	256	512
32	1	2,1	3,9	7,0	12,7	25,3	2,1	4,1	7,5	13,7	27,7
32	2	2,3	4,5	9,0	17,0	34,8	2,3	4,7	9,6	17,8	35,2
32	4	2,2	4,6	9,9	20,6	41,4	2,3	4,9	10,3	21,6	44,2
32	8	2,2	4,6	10,1	22,4	46,7	2,2	4,8	10,6	22,7	48,4
32	16	1,9	4,2	9,6	22,1	47,5	2,0	4,4	9,8	22,6	49,1
32	32	1,6	3,6	8,3	20,1	47,3	1,7	3,6	8,4	20,3	48,0
128	1	5,9	9,7	14,4	22,2	36,1	5,6	9,5	15,3	24,3	41,0
128	2	6,2	11,0	17,7	29,2	48,4	5,9	10,7	17,8	29,2	50,6
128	4	5,7	10,7	18,5	32,2	51,7	5,6	10,7	18,9	32,8	55,0
128	8	4,7	9,4	18,5	33,5	57,3	4,8	9,4	18,4	33,5	59,5
128	16	3,7	7,3	14,9	30,3	57,0	3,7	7,5	15,2	30,8	58,4
128	32	2,5	5,2	10,9	23,6	50,9	2,6	5,3	11,1	23,8	51,6
256	1	10,5	15,8	23,0	32,6	45,9	8,8	14,7	23,4	34,9	50,4
256	2	10,2	16,8	26,7	37,7	55,0	9,0	15,6	25,4	37,4	56,2
256	4	8,7	16,3	28,4	42,4	58,9	8,1	15,3	27,1	42,8	61,9
256	8	6,7	13,1	24,8	42,2	62,8	6,5	12,6	24,3	41,6	64,6
256	16	4,7	9,3	18,4	36,0	61,5	4,7	9,3	18,5	36,3	62,6
256	32	2,9	5,8	11,9	25,0	50,9	2,9	5,9	12,1	25,4	51,5
512	1	17,1	27,0	34,0	45,0	63,9	13,3	23,0	32,6	45,3	63,2
512	2	16,8	29,0	40,2	51,2	71,7	13,2	23,6	35,8	48,9	70,2
512	4	13,0	24,6	39,9	54,6	70,8	11,1	21,7	35,9	53,2	72,1
512	8	9,0	17,8	33,1	53,3	74,3	8,4	16,7	31,1	51,2	76,0
512	16	5,6	11,2	21,8	41,7	72,0	5,5	11,1	21,6	41,5	73,3
512	32	3,2	6,4	12,8	26,0	53,0	3,2	6,6	13,0	26,4	54,0
1024	1	29,9	41,2	50,6	65,3	83,6	19,1	31,0	44,2	61,5	80,3
1024	2	25,4	41,2	55,4	66,4	83,8	17,5	29,8	45,3	60,1	79,6
1024	4	17,8	33,4	53,5	70,4	83,2	14,0	26,7	45,1	66,1	81,1
1024	8	10,9	21,4	40,7	68,3	84,4	9,9	19,3	36,9	63,6	84,0
1024	16	6,2	12,3	24,3	47,3	81,7	6,1	12,1	23,8	46,6	81,8
1024	32	3,3	6,6	13,3	26,8	54,1	3,4	6,8	13,5	27,2	55,0

APÊNDICE B - MATERIAL SUPLEMENTAR

Tabela 6.2: Média e desvio padrão do tempo de execução em milissegundos dos algoritmos em GPU para implementação n-threads (30 execuções)

Parâmetros		n-threads		
Prob. Size	Ind. p/ Blk.	Population		
		32	64	128
32	1	57.8 ± 1.7	74.2 ± 1.3	112.2 ± 0.8
32	2	53.0 ± 0.7	64.2 ± 0.9	87.3 ± 0.7
32	4	53.3 ± 2.0	63.5 ± 2.6	79.4 ± 1.3
32	8	55.5 ± 0.8	62.5 ± 1.0	77.8 ± 1.1
32	16	61.7 ± 1.0	68.2 ± 0.9	82.3 ± 1.3
32	32	74.8 ± 1.6	80.3 ± 0.9	94.8 ± 0.5
128	1	59.8 ± 0.8	78.6 ± 0.6	121.4 ± 0.9
128	2	57.3 ± 0.9	69.5 ± 1.5	98.6 ± 1.1
128	4	62.0 ± 1.2	71.5 ± 1.1	94.6 ± 1.2
128	8	74.3 ± 1.1	81.3 ± 1.1	94.6 ± 0.9
128	16	96.2 ± 0.8	104.2 ± 1.1	117.3 ± 1.0
128	32	140.2 ± 1.4	147.9 ± 1.1	160.7 ± 0.8
256	1	61.7 ± 0.7	86.0 ± 0.8	128.8 ± 1.2
256	2	63.5 ± 1.5	80.7 ± 1.0	111.0 ± 0.9
256	4	75.0 ± 1.3	83.6 ± 0.9	104.1 ± 1.0
256	8	96.7 ± 0.7	103.9 ± 0.7	119.2 ± 1.5
256	16	139.6 ± 1.0	146.8 ± 0.6	160.9 ± 0.7
256	32	227.0 ± 1.2	235.6 ± 0.7	248.9 ± 1.0
512	1	73.8 ± 0.7	97.9 ± 0.7	159.6 ± 1.1
512	2	75.3 ± 1.0	91.2 ± 1.0	134.7 ± 0.9
512	4	97.3 ± 1.1	107.4 ± 2.4	135.8 ± 0.8
512	8	140.2 ± 0.8	148.0 ± 0.8	163.8 ± 1.0
512	16	227.0 ± 0.7	234.9 ± 1.1	248.8 ± 0.7
512	32	400.2 ± 1.5	410.4 ± 0.8	425.0 ± 0.8
1024	1	83.4 ± 0.6	122.3 ± 1.0	203.8 ± 1.0
1024	2	98.0 ± 1.1	122.5 ± 0.7	186.1 ± 0.8
1024	4	140.0 ± 1.1	151.0 ± 1.1	192.7 ± 1.5
1024	8	227.4 ± 1.3	235.3 ± 0.9	253.1 ± 0.8
1024	16	401.3 ± 1.0	408.9 ± 0.7	423.5 ± 0.8
1024	32	747.0 ± 0.7	760.3 ± 0.8	774.9 ± 0.8

Tabela 6.3: Média e desvio padrão do tempo de execução em milissegundos dos algoritmos em GPU para implementação warp modificada (30 execuções)

Parâmetros		Warp Modificada		
Prob. Size	Ind. p/ Blk.	Population		
		32	64	128
32	1	56.7 ± 2.7	71.1 ± 1.3	101.6 ± 0.8
32	2	52.3 ± 2.5	63.3 ± 1.1	83.2 ± 1.1
32	4	54.0 ± 2.3	59.9 ± 1.1	77.0 ± 3.5
32	8	55.2 ± 1.2	63.4 ± 2.1	75.7 ± 0.9
32	16	61.3 ± 1.7	67.9 ± 0.9	81.5 ± 1.0
32	32	73.3 ± 1.5	80.7 ± 0.9	93.3 ± 0.7
128	1	86.9 ± 1.2	124.6 ± 0.7	197.0 ± 1.2
128	2	69.6 ± 1.0	93.6 ± 0.9	139.7 ± 0.9
128	4	66.1 ± 0.5	80.8 ± 1.4	108.4 ± 1.1
128	8	75.7 ± 1.4	82.5 ± 0.9	100.4 ± 0.9
128	16	95.9 ± 1.7	102.2 ± 0.9	115.8 ± 0.9
128	32	138.1 ± 0.9	146.3 ± 1.0	159.1 ± 0.8
256	1	126.7 ± 1.2	199.9 ± 1.0	329.7 ± 0.9
256	2	94.6 ± 1.2	133.7 ± 1.3	218.6 ± 0.9
256	4	85.8 ± 1.4	108.0 ± 1.4	153.4 ± 0.9
256	8	101.8 ± 0.7	109.9 ± 0.8	128.6 ± 0.9
256	16	140.7 ± 0.8	149.1 ± 1.1	163.3 ± 1.1
256	32	223.5 ± 1.0	232.6 ± 1.2	246.0 ± 0.8
512	1	207.4 ± 1.0	350.2 ± 1.2	725.5 ± 1.2
512	2	142.9 ± 1.2	216.1 ± 0.8	376.9 ± 0.8
512	4	122.9 ± 0.8	159.5 ± 1.1	242.9 ± 0.9
512	8	154.1 ± 0.8	163.2 ± 1.0	193.4 ± 1.7
512	16	234.0 ± 1.9	241.1 ± 0.7	256.1 ± 1.1
512	32	392.0 ± 0.8	404.2 ± 0.6	418.6 ± 0.9
1024	1	368.5 ± 0.7	655.5 ± 1.0	1411.2 ± 0.9
1024	2	235.4 ± 1.0	387.3 ± 1.2	690.7 ± 0.8
1024	4	197.7 ± 1.6	266.2 ± 1.0	420.4 ± 0.8
1024	8	260.9 ± 0.8	270.0 ± 0.7	309.2 ± 1.3
1024	16	416.3 ± 0.9	425.1 ± 0.7	438.6 ± 0.9
1024	32	733.7 ± 0.9	747.3 ± 0.9	762.5 ± 1.7

Tabela 6.4: Média e desvio padrão do tempo de execução em milissegundos dos algoritmos em GPU para implementação warp original (30 execuções)

Parâmetros		Warp		
Prob. Size	Ind. p/ Blk.	Population		
		32	64	128
32	1	65.3 ± 1.8	93.3 ± 1.7	222.0 ± 0.8
32	2	60.7 ± 0.2	85.6 ± 0.3	204.7 ± 1.0
32	4	60.9 ± 0.7	83.2 ± 1.3	198.6 ± 0.8
32	8	63.1 ± 0.5	84.0 ± 0.3	201.3 ± 1.2
32	16	69.0 ± 0.4	91.0 ± 0.4	212.4 ± 1.4
32	32	83.6 ± 0.6	106.1 ± 0.5	236.4 ± 1.5
128	1	98.5 ± 1.1	157.3 ± 1.0	367.5 ± 1.9
128	2	80.5 ± 0.2	120.8 ± 0.7	333.5 ± 2.5
128	4	77.3 ± 0.4	111.3 ± 0.7	305.2 ± 2.7
128	8	86.8 ± 0.9	113.8 ± 0.4	297.9 ± 3.1
128	16	109.9 ± 0.5	136.3 ± 0.6	299.4 ± 2.6
128	32	155.4 ± 0.8	185.9 ± 0.7	364.8 ± 2.3
256	1	141.9 ± 1.1	248.2 ± 1.0	595.6 ± 3.1
256	2	107.8 ± 0.4	173.5 ± 0.8	510.8 ± 2.2
256	4	101.9 ± 0.4	155.1 ± 0.5	448.0 ± 2.5
256	8	118.8 ± 0.5	157.9 ± 0.7	425.0 ± 2.7
256	16	159.6 ± 0.6	195.1 ± 0.8	430.5 ± 2.2
256	32	246.2 ± 0.4	285.0 ± 0.8	517.6 ± 3.2
512	1	230.7 ± 3.9	417.5 ± 0.5	1081.0 ± 1.2
512	2	160.8 ± 0.5	277.4 ± 0.7	734.4 ± 0.8
512	4	149.2 ± 0.7	226.1 ± 0.6	601.9 ± 1.1
512	8	180.4 ± 0.4	230.3 ± 0.5	550.1 ± 1.0
512	16	258.6 ± 0.4	306.3 ± 0.7	609.2 ± 1.2
512	32	422.8 ± 0.4	471.6 ± 0.5	760.9 ± 2.1
1024	1	402.4 ± 0.7	729.5 ± 0.4	1770.3 ± 0.7
1024	2	262.9 ± 0.7	465.0 ± 0.5	1050.4 ± 0.7
1024	4	230.1 ± 0.6	339.5 ± 0.4	777.6 ± 0.7
1024	8	293.5 ± 0.5	345.3 ± 0.2	667.2 ± 0.6
1024	16	448.6 ± 0.4	499.3 ± 0.5	797.9 ± 0.7
1024	32	767.5 ± 0.6	822.6 ± 0.4	1126.5 ± 0.8

APÊNDICE C - MATERIAL SUPLEMENTAR

Tabela 6.5: Média e desvio padrão do tempo de execução em milissegundos dos algoritmos em CPU (30 execuções)

Parâmetros	CPU		
Prob. Size	Population		
	32	64	128
32	119.3 ± 4.2	289.4 ± 10.3	786.1 ± 22.3
128	352.7 ± 16.2	762.7 ± 24.3	1748.9 ± 3.2
256	650.7 ± 0.9	1359.0 ± 7.0	2959.7 ± 1.9
512	1262.4 ± 0.8	2642.0 ± 11.0	5420.9 ± 0.8
1024	2488.8 ± 0.9	5043.4 ± 14.6	10304.9 ± 19.8