

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

BRUNO BELENDIA MEURER

Implementação paralela do algoritmo Barnes-Hut para simulação do problema N-Corpos
usando um número arbitrário de GPU's

RIO DE JANEIRO
2020

BRUNO BELEND A MEURER

Implementação paralela do algoritmo Barnes-Hut para simulação do problema N-Corpos
usando um número arbitrário de GPU's

Trabalho de conclusão de curso de graduação
apresentado ao Departamento de Ciência da
Computação da Universidade Federal do Rio
de Janeiro como parte dos requisitos para ob-
tenção do grau de Bacharel em Ciência da
Computação.

Orientador: Profa. Silvana Rossetto

Co-orientador:

RIO DE JANEIRO

2020

CIP - Catalogação na Publicação

M598i Meurer, Bruno Belenda
Implementação paralela do algoritmo Barnes-Hut para simulação do problema N-Corpos usando um número arbitrário de GPU's / Bruno Belenda Meurer. -- Rio de Janeiro, 2020.
47 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2020.

1. Barnes-Hut. 2. GPU. 3. CUDA. I. Rossetto, Silvana, orient. II. Título.

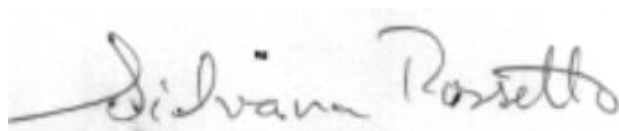
BRUNO BELENDIA MEURER

Implementação paralela do algoritmo Barnes-Hut para simulação do problema N-Corpos usando um número arbitrário de GPU's

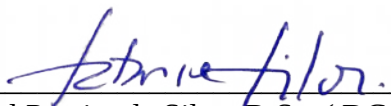
Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 16 de Julho de 2020.

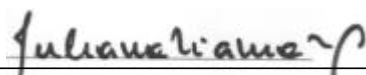
BANCA EXAMINADORA:



Silvana Rossetto, D.Sc.(DCC-UFRJ)



Gabriel Pereira da Silva, D.Sc. (DCC-UFRJ)



Juliana Vianna Valério, D.Sc. (DCG-UFRJ)

“Don't be too proud of this technological terror you've constructed”

Darth Vader

RESUMO

O problema N-corpos é o problema referente à previsão do comportamento de corpos individuais dentro de um sistema dinâmico onde todos os corpos interagem entre si. Ele aparece em diferentes áreas de estudo, desde a simulação das interações entre corpos de gigantesca escala, como corpos celestes (planetas, estrelas, galáxias), até escalas microscópicas como pequenas partículas. Trata-se de um problema que normalmente requer grande esforço computacional para ser resolvido e, devido a isso, diversos algoritmos foram desenvolvidos ao longo dos anos para reduzir o tempo de processamento necessário. Entre eles está o algoritmo de Barnes-Hut, que realiza aproximações dos corpos que estão sendo calculados de modo a agrupá-los e minimizar os cálculos realizados. Nos últimos anos, com a disponibilização das unidades de processamento gráfico (GPUs) para execução paralela de algoritmos de propósito geral, várias soluções de paralelização do problema de N-Corpos foram propostas para essa nova plataforma. Neste trabalho, estendemos uma implementação paralela do algoritmo de Barnes-Hut para GPU, permitindo o uso de um número arbitrário de GPU's. Avaliamos o comportamento do programa a cada GPU que adicionamos, testando com até 4 GPU's simultaneamente. Observamos que os resultados positivos obtidos estão de acordo com nossas expectativas, onde identificamos que a aceleração inicialmente se aproxima do máximo teórico, porém, à medida que se aumenta o número de GPU's, o custo de gerenciamento cresce, reduzindo o ganho de aceleração.

Palavras-chave: Problema N-Corpos. Algoritmo de Barnes-Hut. GPU. CUDA.

ABSTRACT

The N-bodies problem is the problem regarding the prediction of the behavior of individual bodies within a dynamic system where all bodies interact with each other. It appears in different areas of study, from the simulation of interactions between bodies of gigantic scale, like celestial bodies (planets, stars, galaxies), up to microscopic scales like small particles. It is a problem that usually requires a great deal of computational effort to be solved and, due to this, several algorithms have been developed over the years to reduce the necessary processing time. Among them is the Barnes-Hut algorithm, which performs approximations of the bodies being calculated in order to group them and avoid performing unnecessary calculations. In recent years, with the availability of graphic processing units (GPUs) for parallel execution of general purpose algorithms, several solutions to parallelize the N-body problem have been proposed for this new platform. In this work, we extend a parallel implementation of the Barnes-Hut algorithm for GPU, allowing the use of an arbitrary number of GPU's and then we evaluate the program's behavior for each GPU we add, up to 4 simultaneously. We observed that the positive results obtained are in line with our expectations, where we identified that the speedup initially reaches the theoretical maximum, however, as the number of GPU's increases, the management cost increases, reducing the gain of speedup.

Keywords: N-bodies problem. Barnes-Hut algorithm. GPU. CUDA.

LISTA DE ILUSTRAÇÕES

Figura 1 – Representação básica do problema N-corpos.	14
Figura 2 – Interações com o método de Euler.	16
Figura 3 – Aproximação de corpos distantes em um único corpo.	17
Figura 4 – Nó raiz que representa todo o espaço da simulação.	18
Figura 5 – Primeira subdivisão do espaço.	18
Figura 6 – Segunda e última subdivisão do espaço.	19
Figura 7 – Interação entre o corpo e uma outra folha da árvore.	20
Figura 8 – Interação entre o corpo e um nó interno da árvore.	21
Figura 9 – Identificando que os corpos do nó devem ser aproximados em um só corpo.	21
Figura 10 – Diferenças estruturais entre CPU e GPU.	24
Figura 11 – Hierarquia de grade, blocos e <i>threads</i>	26
Figura 12 – Hierarquia de memória	28
Figura 13 – Fluxo de execução original.	32
Figura 14 – Medidas de desempenho do código original.	33
Figura 15 – Fluxo de execução original \times Fluxo de execução alterado.	34
Figura 16 – Representação da ordenação em memória (o vetor 1 guarda as infor- mações dos corpos na ordenação inicial, enquanto o vetor 2 armazena a ordenação dos corpos após o <i>kernel 4</i>).	35
Figura 17 – Alterações feitas pelas GPU's (após cada GPU calcular as interações, os resultados não se encontram sequencialmente em memória).	35
Figura 18 – Fluxo de execução alterado.	36
Figura 19 – Topologia do servidor	38
Figura 20 – Velocidade de transferência de memória	38
Figura 21 – Resultados de aceleração	41
Figura 22 – Resultados de tempo de execução	42

LISTA DE CÓDIGOS

Código 1	Kernel 7	47
----------	--------------------	----

LISTA DE TABELAS

Tabela 1 – Métricas obtidas (execução completa)	40
Tabela 2 – Métricas obtidas (apenas do <i>kernel</i> alterado)	40

LISTA DE ABREVIATURAS E SIGLAS

CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
CUDA	Compute Unified Device Architecture
ALU	Arithmetic Logic Unit
SIMT	Single Instruction Multiple Thread

SUMÁRIO

1	INTRODUÇÃO	12
1.1	OBJETIVO	13
1.2	CONTRIBUIÇÕES	13
1.3	ORGANIZAÇÃO DO TEXTO	13
2	O PROBLEMA N-CORPOS	14
2.1	DEFINIÇÃO DO PROBLEMA	14
2.2	A MATEMÁTICA POR TRÁS DO PROBLEMA	14
2.3	VISÃO GERAL DOS ALGORITMOS	15
2.3.1	Método de Euler	15
2.3.2	Algoritmo de Barnes-Hut	16
2.3.2.1	Divisão do espaço utilizando a estrutura de dados árvore	17
2.3.2.2	Utilizando a estrutura de dados árvore para calcular interações	19
2.3.3	Implementações sequenciais do algoritmo de Barnes-Hut	20
2.3.4	Implementações paralelas do algoritmo de Barnes-Hut em CPU	21
3	PROGRAMAÇÃO PARALELA EM GPU	23
3.1	HISTÓRICO E PRINCÍPIOS ARQUITETURAIS DAS GPU'S	23
3.2	PLATAFORMA DE PROGRAMAÇÃO CUDA	24
3.2.1	Modelo de programação de CUDA	24
3.2.2	Hierarquia de <i>threads</i> , blocos e grade	25
3.2.3	Modelo de execução de CUDA	26
3.2.4	Hierarquia de memória	27
3.2.5	Estratégias básicas de otimização	29
4	IMPLEMENTAÇÃO PARALELA DO ALGORITMO DE BARNES-HUT COM UM NÚMERO ARBITRÁRIO DE GPU'S	31
4.1	ESTRUTURA DO CÓDIGO ORIGINAL	31
4.2	EXTENSÃO DO CÓDIGO ORIGINAL PARA EXECUÇÃO EM UM NÚMERO ARBITRÁRIO DE GPU'S	33
4.3	PROJETO DA EXPANSÃO DA SOLUÇÃO ORIGINAL	33
5	AVALIAÇÃO	37
5.1	PLATAFORMA DE TESTES	37
5.2	MÉTRICAS DE AVALIAÇÃO	39
5.3	RESULTADOS OBTIDOS	39

6	CONCLUSÃO	43
	REFERÊNCIAS	44
	APÊNDICE A – CÓDIGO DO KERNEL 7 CRIADO PARA TRANS- FERÊNCIA DE MEMÓRIA.	47

1 INTRODUÇÃO

Compreender a dinâmica de corpos celestes de modo a prever seu comportamento ao longo do tempo é fundamental para a astrofísica moderna. Observações diretas podem ser realizadas através da utilização de telescópios massivos distribuídos em localizações remotas ao redor do globo ou até mesmo do espaço. Essas observações são capazes de revelar muito a respeito das características dos corpos que estamos observando hoje em dia, porém não revelam como suas trajetórias atuais irão se comportar daqui a centenas ou milhares de anos.

Para realizar esse tipo de análise, muitas vezes é necessário fazer uso de simulações computacionais, utilizando algoritmos partícula-partícula para determinar o comportamento futuro dos corpos. A solução mais trivial para esse caso é o algoritmo de força bruta onde todas as interações entre todas as partículas são calculadas a cada passo da simulação. Apesar de bastante útil para pequenas simulações por ser o mais preciso, esse método demanda um poder computacional que não temos disponível hoje, até mesmo nos mais poderosos supercomputadores, quando começamos a nos aproximar da quantidade de corpos envolvidos em simulações de colisões de galáxias, por exemplo.

Devido a essa complexidade, ao longo dos anos foram realizados esforços para elaborar algoritmos que permitissem diminuir a quantidade de processamento necessário para realizar essas simulações. Um desses algoritmos é o algoritmo de Barnes-Hut (BARNES; HUT, 1986) em que, por meio de algumas aproximações, corpos muito distantes são aproximados como se fossem apenas um único corpo. Dessa forma, cálculos de interações com corpos que teriam mínima influência na força aplicada sobre o corpo são minimizados, diminuindo assim a complexidade de execução do algoritmo de $O(N^2)$ para $O(N \log(N))$. Esse algoritmo, aliado ao poder computacional das GPU's modernas, nos permite realizar simulações que anos atrás seriam inimagináveis.

Burtscher e Pingali (BURTSCHER; PINGALI, 2011a) propuseram uma implementação paralela do algoritmo de Barnes-Hut para GPUs na qual todos os procedimentos necessários para executar o algoritmo de Barnes-Hut foram implementados para serem totalmente executados na GPU, sem necessidade de realizar transferências de dados entre a memória do sistema e a memória da GPU, exceto pela transferência inicial de todas as informações da simulação e pela transferência do resultado final. A implementação feita por eles ((BURTSCHER; PINGALI, 2011b)) serviu de base para o nosso trabalho.

Em (MEURER et al., 2018) apresentamos uma extensão dessa implementação para execução simultânea em duas GPUs. Os resultados mostraram que, mesmo com o *overhead* adicionado ao incluir mais uma GPU, devido às transferências de memória extras que eram necessárias, a aceleração obtida aproximou-se do máximo teórico possível.

1.1 OBJETIVO

Neste trabalho, propomos generalizar a implementação anterior (MEURER et al., 2018) de modo que seja possível executar as simulações do problema N-Corpos com uma quantidade arbitrária de GPU's.

Os principais desafios a serem tratados nesse trabalho são a criação da lógica de gerenciamento das transferências de memória entre todas as GPU's envolvidas, além da criação de um módulo extra que organiza os resultados calculados das outras GPU's antes que a simulação possa continuar.

Avaliamos a nova implementação com simulações de 40 milhões de corpos (a maior quantidade possível que conseguimos simular em nosso ambiente de teste). Os resultados obtidos estão de acordo com nossas expectativas, onde identificamos que a aceleração inicialmente se aproxima do máximo teórico, porém, à medida que se aumenta o número de GPU's, o custo de gerenciamento aumenta, reduzindo o ganho de aceleração.

1.2 CONTRIBUIÇÕES

A principal contribuição deste trabalho é a disponibilização de uma nova implementação paralela em GPU do algoritmo de Barnes-Hut, baseada no trabalho original de Burtscher e Pingali. Essa implementação está disponível em <https://github.com/bmeurer1/TCC>

1.3 ORGANIZAÇÃO DO TEXTO

O restante deste texto está organizado da seguinte forma: No capítulo 2 são apresentados alguns conceitos importantes a respeito do problema N-Corpos e de como são realizadas as simulações. Em seguida, no capítulo 3 são descritas as características principais das GPU's e de seu modelo de execução. No capítulo 4 a proposta deste trabalho é apresentada e detalhada. No capítulo 5 essa proposta é avaliada e os resultados obtidos são comparados com os resultados do trabalho original. Por fim, no capítulo 6, são apresentadas as conclusões deste trabalho.

2 O PROBLEMA N-CORPOS

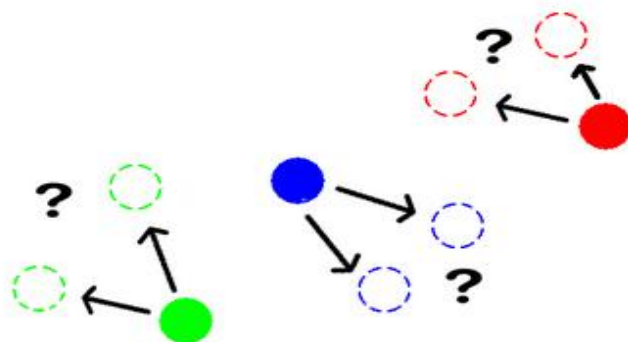
Nesse capítulo iremos detalhar o que é o problema de N-Corpos e mostrar alguns algoritmos básicos para solucioná-lo. Por fim iremos explorar algumas das implementações existentes, tanto sequenciais, quanto paralelas desses algoritmos.

2.1 DEFINIÇÃO DO PROBLEMA

O problema de N-corpos trata da seguinte questão: dado um conjunto de corpos (por exemplo, partículas, planetas ou galáxias) e a configuração inicial das forças e interações agindo sobre eles, como será o comportamento desse sistema no futuro, dado que todos esses corpos interagem entre si?

A Figura 1 ilustra o problema de maneira simplificada. São representados 3 corpos (círculos preenchidos) que estão interagindo entre si em um sistema e as possíveis posições futuras dos mesmos (círculos pontilhados).

Figura 1 – Representação básica do problema N-corpos.



2.2 A MATEMÁTICA POR TRÁS DO PROBLEMA

Apesar de a ideia de simular milhões de corpos interagindo entre si ao mesmo tempo parecer extremamente complexa, os princípios físicos que regem o comportamento são relativamente simples. Para o cálculo das forças relativas às interações entre os corpos, utilizamos duas equações: A segunda lei de Newton (2.1) e a lei da Gravitação Universal (2.2). A primeira calcula a força vetorial exercida no corpo \vec{F} em função da massa m_1 e

da aceleração \vec{a} , e a segunda calcula o vetor da força exercida no corpo \vec{F} em função da constante gravitacional G , das massas dos dois corpos envolvidos na interação, m_1 e m_2 , e da distância vetorial entre os corpos \vec{r} .

$$\vec{F} = m_1 \cdot \vec{a} \quad (2.1)$$

$$\vec{F} = G \frac{m_1 m_2}{\|\vec{r}\|^2} \cdot \frac{\vec{r}}{\|\vec{r}\|} \quad (2.2)$$

A partir das equações acima, podemos começar a fazer manipulações que nos permitam simular cada passo de tempo. Ao manipularmos as equações 2.1 e 2.2, podemos obter o valor da aceleração:

$$m_1 \cdot \vec{a} = G \frac{m_1 m_2}{\|\vec{r}\|^2} \cdot \frac{\vec{r}}{\|\vec{r}\|} \quad (2.3)$$

$$\vec{a} = G \frac{m_2}{\|\vec{r}\|^2} \cdot \frac{\vec{r}}{\|\vec{r}\|} \quad (2.4)$$

Tendo obtido a aceleração na equação 2.4, temos apenas que atualizar os valores da velocidade e da posição de cada corpo, utilizando as equações da velocidade (2.5) e da posição (2.6) e o valor do passo de tempo desejado. A primeira equação calcula a velocidade \vec{V} em forma vetorial ao final do passo de tempo atual, em função da aceleração calculada na equação 2.4, da velocidade inicial \vec{V}_0 e do intervalo de tempo representado por um passo de tempo simulado Δt . A segunda equação utiliza o mesmo procedimento da equação 2.5 para realizar o cálculo da posição após o passo de tempo \vec{P} em função da posição inicial \vec{P}_0 , da velocidade atual \vec{V} e do intervalo de tempo Δt .

$$\vec{V} = \vec{V}_0 + \vec{a} \cdot \Delta t \quad (2.5)$$

$$\vec{P} = \vec{P}_0 + \vec{V} \cdot \Delta t \quad (2.6)$$

2.3 VISÃO GERAL DOS ALGORITMOS

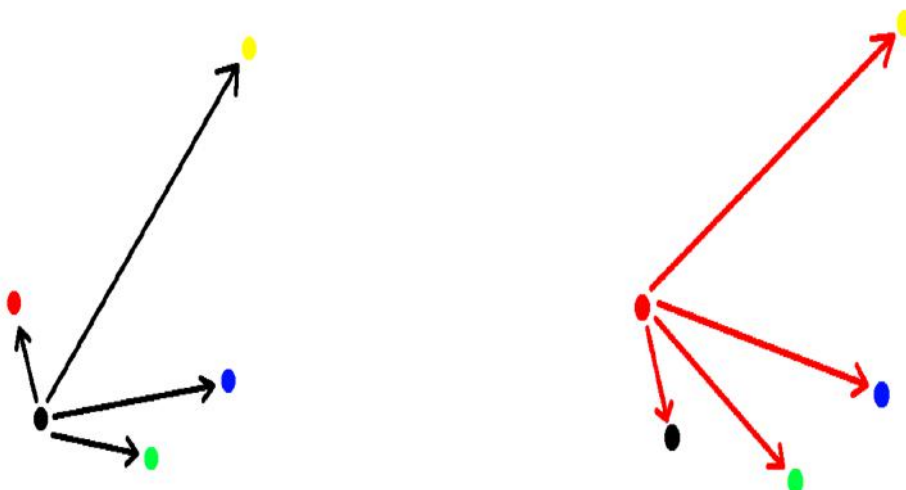
Para o problema de N-Corpos foram desenvolvidos alguns algoritmos ao longo dos anos. Nessa seção iremos explorar dois deles: o método de Euler, que é bastante preciso mas peca em termos de desempenho, e o algoritmo de Barnes-Hut, que realiza algumas aproximações em troca de um desempenho significativamente melhor.

2.3.1 Método de Euler

O método de Euler (SINKAROV et al., 2014) é um algoritmo de partícula-partícula que realiza todas as interações possíveis entre os corpos que estamos tentando simular.

Esse é o algoritmo mais simples e preciso pois não realiza qualquer tipo de aproximação das forças e interações e é bastante recomendado para pequenas simulações onde a acurácia dos resultados é de grande importância. A Figura 2 apresenta uma visualização do método. Primeiro são calculadas todas as interações entre o corpo preto e o resto dos corpos, em seguida são calculadas todas as interações entre o corpo vermelho e o resto dos corpos, até que todos os pares tenham sido calculados.

Figura 2 – Interações com o método de Euler.



Apesar de ser um algoritmo bastante recomendado em termos de precisão, a partir do momento em que as simulações começam a crescer em termos de número absoluto de corpos, ele se torna impraticável devido ao tempo computacional necessário. Isso se dá pelo fato de que, por calcular todas as interações entre todos os corpos, sua complexidade é da ordem de $O(N^2)$, sendo N o número de corpos envolvidos na simulação.

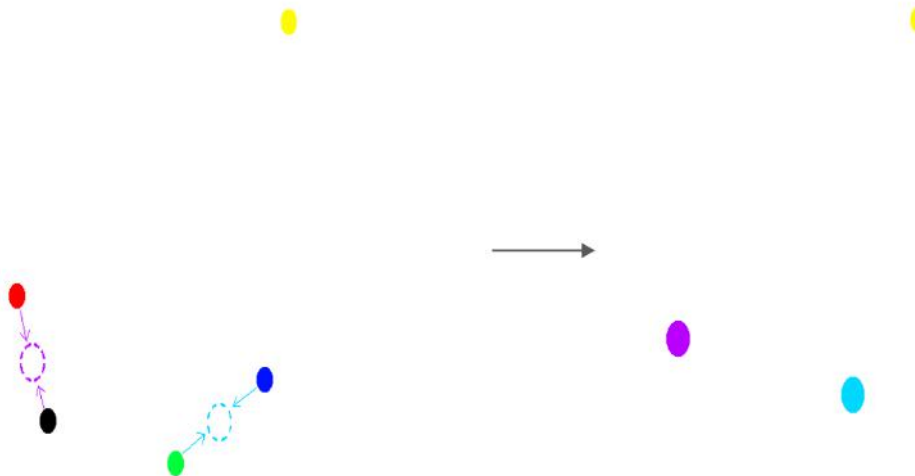
2.3.2 Algoritmo de Barnes-Hut

Em casos de simulações com uma quantidade expressiva de corpos, mas que não necessariamente necessitam precisão absoluta, como simulações de colisões de galáxias, por exemplo, onde o algoritmo de Euler não seria factível, o algoritmo de Barnes-Hut se torna uma opção viável. Esse algoritmo faz uso de algumas aproximações para diminuir significativamente a quantidade de interações calculadas, permitindo assim que a complexidade computacional caia para $O(N \log(N))$.

A principal característica do algoritmo de Barnes-Hut é considerar um grupo de corpos que estejam muito distantes do corpo cujas interações estamos calculando como se fosse um único corpo, cuja posição se encontra no centro de massa de todos os corpos que estamos aproximando e cuja massa corresponde à soma das massas de todos esses corpos, como

pode ser visto na Figura 9. Nesse cenário, os corpos vermelho e preto são aproximados pelo corpo roxo e os corpos azul escuro e verde são aproximados pelo corpo azul claro. Quando as interações forem calculadas, o corpo amarelo só calculará as interações com os corpos roxo e azul claro, economizando metade dos cálculos que seriam realizados.

Figura 3 – Aproximação de corpos distantes em um único corpo.



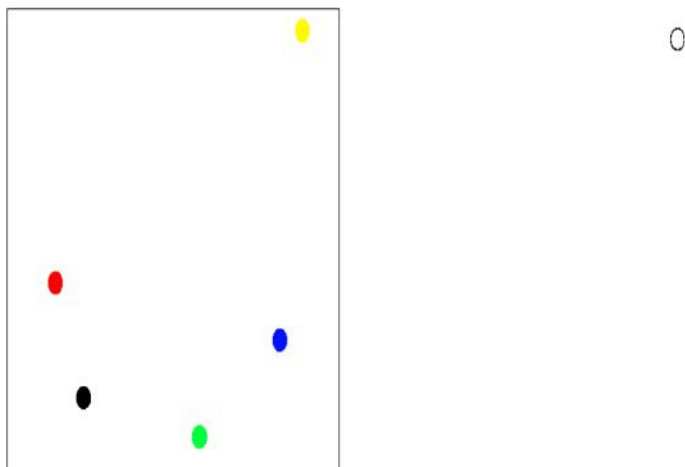
Essas aproximações fazem com que o resultado final não seja exato, devido ao fato de que algumas interações estão sendo agrupadas em apenas uma. Apesar disso, o impacto final acaba não sendo muito grande, pois como a força na Lei da Gravitação Universal diminui com o quadrado da distância, corpos que estão muito distantes já não têm muita influência na movimentação do corpo que estamos calculando. Essas aproximações permitem que a simulação evite processar interações do corpo que estamos analisando com todos os corpos que foram aproximados, acelerando tremendamente a execução do programa.

2.3.2.1 Divisão do espaço utilizando a estrutura de dados árvore

Para identificar quais corpos devem ou não ser agrupados em um único corpo durante a simulação, o algoritmo faz uso da estrutura de dados árvore que divide o espaço em quadrantes (ou octantes em simulações de três dimensões). Primeiro, inicia-se com um nó raiz que representa todo o espaço da simulação. Na Figura 4, o círculo do lado direito é o nó que representa todo o espaço envolvido pelo quadrado do lado esquerdo.

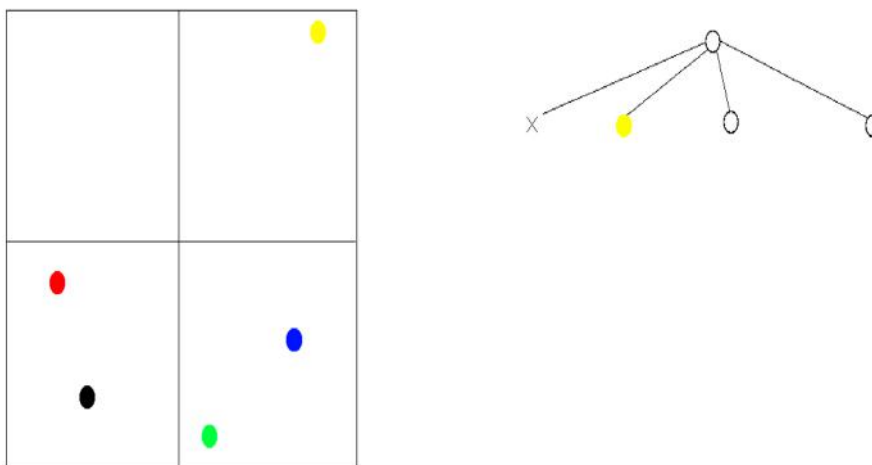
A partir do nó raiz, o espaço é continuamente subdividido até que cada corpo esteja sozinho em seu próprio quadrante, conforme ilustrado nas Figuras 5 e 6. Conforme podemos ver na Figura 5, o espaço total da simulação foi dividido em 4 novos quadrantes

Figura 4 – Nó raiz que representa todo o espaço da simulação.



pois na repartição inicial todos os corpos ocupavam o mesmo quadrante. No lado direito da Figura o nó raiz possui um filho vazio, pois não há nenhum corpo dentro daquele quadrante, uma folha que é o corpo amarelo, pois é o único dentro daquele quadrante, e dois filhos que são outros nós que representam os dois quadrantes restantes, pois cada um possui mais do que um corpo dentro do seu espaço.

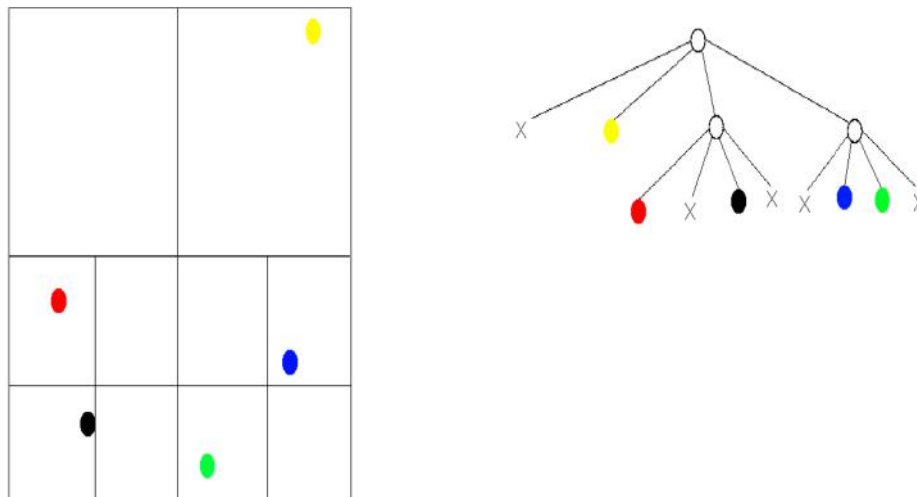
Figura 5 – Primeira subdivisão do espaço.



Em seguida, na Figura 6, vemos que os corpos vermelho e preto ainda estão contidos dentro do mesmo quadrante, assim como os corpos azul e verde. Devido a isso, o processo se repete para cada um dos dois nós que foram criados e os corpos azul, preto, verde e

vermelho são adicionados como folhas da árvore, pois na segunda subdivisão encontram-se sozinhos em seus quadrantes.

Figura 6 – Segunda e última subdivisão do espaço.



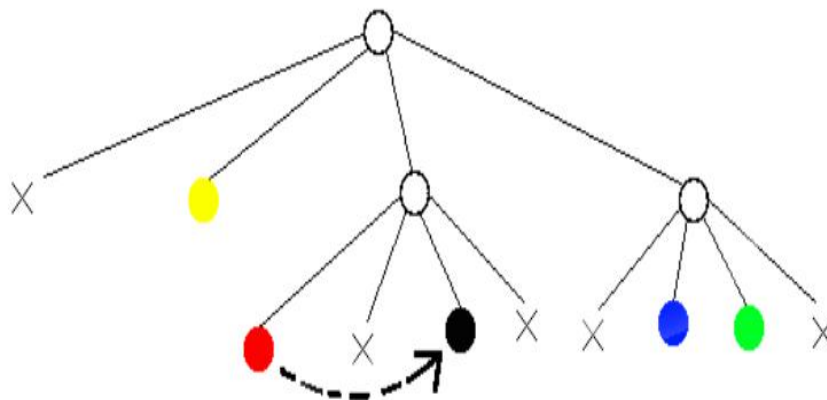
2.3.2.2 Utilizando a estrutura de dados árvore para calcular interações

A partir do momento em que temos a árvore construída, podemos começar a calcular as interações entre os corpos. Para cada corpo cujas interações queremos calcular, iniciamos no nó raiz da árvore e a cada passo avaliamos duas condições:

Condição 1 - O nó que estamos avaliando no momento é uma folha da árvore. Esse é o caso mais simples. Quando chegamos a uma folha da árvore (e essa folha não é o próprio corpo cujas interações estamos calculando), devemos apenas calcular a força que esse corpo exerce no corpo de referência, pois não temos mais como descer na árvore. A Figura 7 ilustra esse caso. O círculo vermelho é o corpo cujas interações estamos calculando e o corpo preto é a folha da árvore que foi alcançada ao percorrê-la. Como não há mais como descer na árvore, é calculada a interação entre o corpo vermelho e o corpo preto.

Condição 2 - O nó que estamos avaliando no momento é um nó interno da árvore. Nesse caso, devemos avaliar se os corpos do espaço que o nó representa estão distantes o suficiente para serem considerados um corpo só. Para isso, calculamos a razão $\frac{s}{d}$, onde “ d ” é a distância entre o corpo e o nó e “ s ” é o comprimento do espaço representado pelo nó. Se a razão $\frac{s}{d} < \theta$, sendo θ um parâmetro que definimos no início da simulação, calculamos a força que o nó exerce sobre o corpo. Caso contrário, aplicamos o mesmo procedimento recursivamente em todos os filhos do nó. As Figuras 8 e 9 ilustram esse caso. O corpo

Figura 7 – Interação entre o corpo e uma outra folha da árvore.



amarelo é o corpo cujas interações estamos calculando e, durante o percorrimento da árvore, foi encontrado o nó interno que representa o espaço ocupado pelos corpos preto e vermelho. Deve-se, portanto, verificar se é necessário continuar percorrendo a árvore ou não. Aplicando o procedimento descrito anteriormente com os valores exemplificados na Figura 9, identificamos que os corpos preto e vermelho estão suficientemente distantes e podem ser aproximados em um único corpo.

A escolha para o valor de θ irá depender dos propósitos da simulação que está sendo feita. Quanto mais próximo de 0 for, menos aproximações serão feitas ao longo da simulação e mais preciso será o resultado final, porém, conseqüentemente, maior será o tempo total de execução. Em contrapartida, quanto maior for o valor de θ , mais aproximações serão feitas e mais rápido será executado o programa, porém a precisão irá diminuir. Essa relação pode ser visualizada em (HEER, 2017).

2.3.3 Implementações sequenciais do algoritmo de Barnes-Hut

Devido à sua aplicabilidade em diversas áreas distintas, o algoritmo de Barnes-Hut foi foco de muitas pesquisas ao longo dos anos ((YI RUOYU LI, 2017), (C. PINEL N., 2014), (REUTER et al., 2017)). Como exemplo, podemos citar o trabalho realizado por (MAATEN, 2014), onde o algoritmo foi implementado para acelerar o t-SNE (*t-distributed Stochastic Neighbor Embedding*), uma técnica de *embedding* utilizada para redução de dimensionalidade de dados de modo a auxiliar sua visualização em gráficos de dispersão tradicionais. Em seus resultados, é possível ver que a versão do algoritmo t-SNE que

Figura 8 – Interação entre o corpo e um nó interno da árvore.

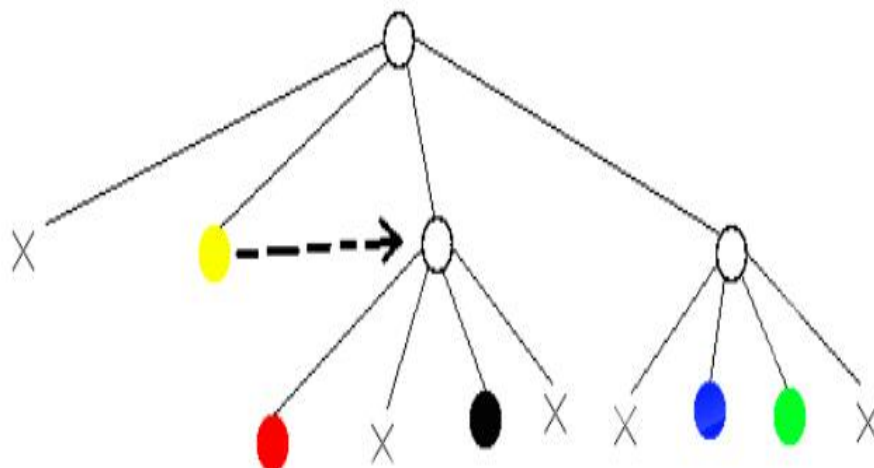
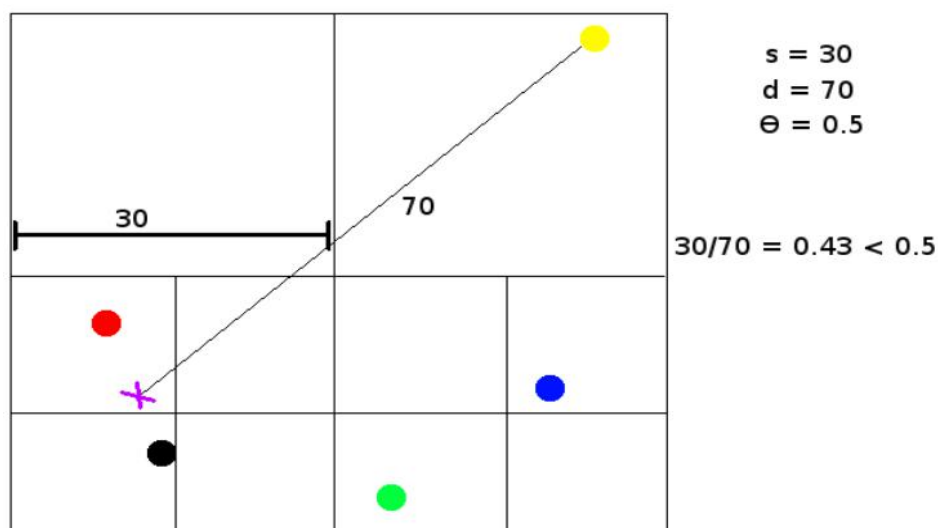


Figura 9 – Identificando que os corpos do nó devem ser aproximados em um só corpo.



utiliza Barnes-Hut é, aproximadamente, até duas ordens de magnitude mais rápida do que a implementação padrão do t-SNE original.

2.3.4 Implementações paralelas do algoritmo de Barnes-Hut em CPU

Além dos esforços para aplicar o algoritmo de Barnes-Hut em diferentes problemas, também foram desenvolvidas, ao longo dos anos, implementações paralelas do algoritmo em CPU. Como exemplos desses esforços podemos citar o trabalho de Grama et al. (GRAMA; KUMAR; SAMEH, 1998), onde os autores realizaram uma implementação

paralela em CPU do algoritmo de Barnes-Hut para realizar simulações nas quais a densidade espacial dos corpos é irregular, obtendo bons resultados em termos de aceleração do processamento conforme mais processadores eram adicionados à simulação.

Outro exemplo que podemos citar é o trabalho de Munier et al. (MUNIER et al., 2020), onde o algoritmo de Barnes-Hut foi implementado de forma paralela em 3 bibliotecas de Java (*Java thread API*, *JavaSymphony* e *MPJ Express*), de modo a realizar um *benchmark* entre elas e identificar a viabilidade de Java como uma plataforma para desenvolvimento de soluções para computação de alto desempenho. Os resultados obtidos mostraram que a implementação utilizando *JavaSymphony* foi a que obteve o menor tempo de execução entre as implementações testadas e que foi possível simular uma grande quantidade de corpos (até um milhão de corpos).

3 PROGRAMAÇÃO PARALELA EM GPU

Nesse capítulo, apresentaremos uma visão geral a respeito das GPU's, o principal hardware utilizado em nosso trabalho. Detalharemos um pouco da sua história, seus princípios arquiteturais e cuidados necessários durante o desenvolvimento dos programas para se obter o máximo de desempenho possível da aplicação.

3.1 HISTÓRICO E PRINCÍPIOS ARQUITETURAIS DAS GPU'S

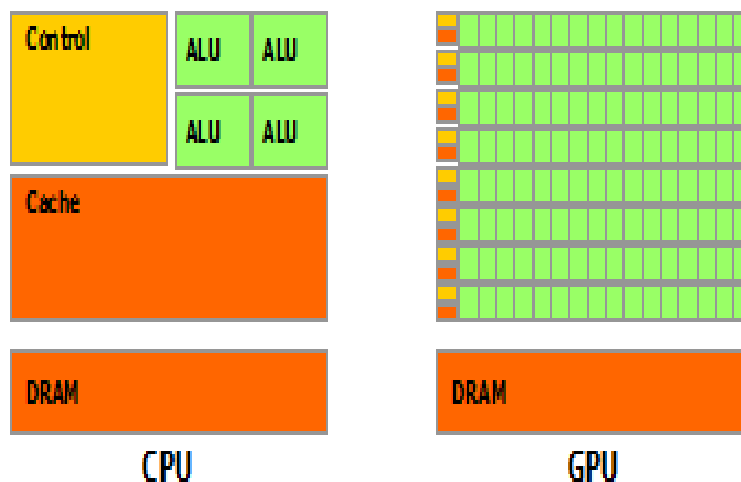
Influenciados pela enorme demanda por processamento de gráficos de alta fidelidade em tempo real, as placas de vídeo (GPU's) têm evoluído drasticamente desde a sua introdução no mercado, no final da década de 90, com o modelo GeForce 256 da Nvidia, até os modelos mais modernos como a GeForce Titan RTX, tornando-se um coprocessador massivamente paralelo capaz de realizar uma tremenda quantidade de cálculos por segundo. Inicialmente as GPU's eram utilizadas como coprocessadores para acelerar cálculos relacionados a computação gráfica, porém, com o tempo, passaram a ser utilizadas para resolver problemas de diversas outras áreas, apesar da dificuldade de programá-las por meio de seus pipelines gráficos.

Uma grande mudança de paradigma ocorreu em 2006 com a introdução da plataforma CUDA (NVIDIA Corporation, 2010) pela Nvidia e, um pouco mais tarde, do OpenCL, um padrão aberto para o desenvolvimento de aplicações que executam em ambientes computacionais heterogêneos (MUNSHI, 2009). Por meio desses novos ambientes, a programação das placas de vídeo tornou-se muito mais simples e permitiu uma expansão da utilização do poder de processamento das mesmas para diversas outras áreas, como inteligência artificial, computação científica, exploração de óleo e gás, entre outras. Devido ao fato de as placas de vídeo serem usadas nas mais variadas áreas, as mesmas também são chamadas atualmente de GPGPU's (*General Purpose Graphics Processing Unit*), ou Unidade de Processamento Gráfico de Propósito Geral.

A discrepância das GPU's com relação às CPU's, em termos de capacidade de processamento paralelo, se deve ao fato de que as primeiras optam por oferecer um grande número de unidades de processamento com baixa complexidade de controle, permitindo que as trocas de contexto entre os fluxos de execução sejam feitas de forma rápida. A Figura 10 ilustra as diferenças entre as GPU's e as CPU's de maneira simplificada. É possível ver a enorme diferença de recursos alocados ao processamento de dados (ALU, em verde) entre a CPU e a GPU:

Nas GPU's, o modelo de execução SIMT (*Single instruction, multiple thread*) é utilizado. Isso significa que as threads executam a mesma instrução em elementos de dados distintos. Por isso, as GPU's são mais adequadas para problemas com alto paralelismo de

Figura 10 – Diferenças estruturais entre CPU e GPU.



Fonte: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#from-graphics-processing-to-general-purpose-parallel-computing__gpu-devotes-more-transistors-to-data-processing

dados. Simulações de interações partícula-partícula, como o problema que o algoritmo de Barnes-Hut computa, são ótimos exemplos de situações em que o poder computacional das GPU's pode ser bem aproveitado. Cada uma das interações entre os corpos é totalmente independente das outras, significando que todas podem ser calculadas ao mesmo tempo e em paralelo.

Apesar do seu grande potencial de processamento paralelo, a programação com GPU's envolve algumas particularidades que, se não tratadas corretamente, podem causar gargalos inesperados, diminuindo o ganho potencial da introdução da GPU no fluxo das operações. Iremos explorar alguns dos problemas mais comuns mais adiante.

3.2 PLATAFORMA DE PROGRAMAÇÃO CUDA

Nessa seção iremos apresentar conceitos básicos da programação com GPU's usando a plataforma CUDA. CUDA é uma API de programação e desenvolvimento exclusiva para as GPU's da Nvidia que permite que desenvolvedores de software utilizem GPU's para programação de propósito geral. Por meio dessa API, programadores podem desenvolver código em linguagens familiares, como C, C++ ou Fortran, e apenas adaptar o programa com diretivas específicas de CUDA para fazer uso do poder computacional da GPU.

3.2.1 Modelo de programação de CUDA

A execução de código na GPU é feita por meio de chamadas a funções especiais denominadas *kernels*. *Kernels* são funções utilizadas para executar porções paralelas do

programa na GPU, cujo conteúdo é o código que será executado pelas *threads*. Todas as *threads* executam o mesmo código, porém há a possibilidade de distingui-las entre si através de um identificador único para gerenciar o fluxo de execução.

Em um cenário típico, as GPU's executam um *kernel* de cada vez, em sequência, os quais são independentes um dos outros (as *threads* de um *kernel* não se comunicam com as *threads* de outro *kernel*). Devido a isso, os *kernels* podem ser utilizados como barreiras globais durante a execução do código completo de uma aplicação para garantir a sincronização de diferentes partes desse código. Desse modo, uma tarefa A precisa necessariamente executar antes da tarefa B, construir um *kernel* separado para cada uma irá garantir que as tarefas irão executar sequencialmente.

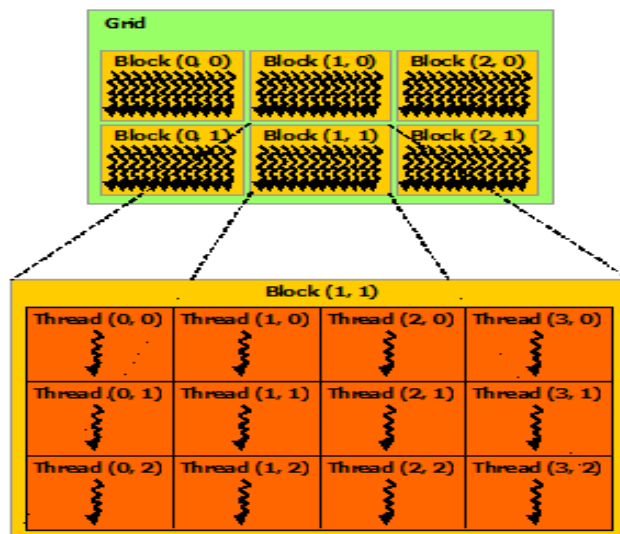
Por se tratar de um coprocessador, durante a execução de programas as GPU's modernas ainda são tratadas como uma entidade separada do sistema principal. O processador principal (*Host*) não tem como ler ou escrever na memória da GPU, assim como a GPU não consegue fazer o mesmo com a memória do sistema. Isso significa que o programa inicia sua execução com todos os dados em memória do sistema e, durante a execução do programa, os mesmos devem ser transferidos para a GPU antes que sejam processados por ela, e posteriormente devolvidos com o resultado dos cálculos realizados. Um exemplo de fluxo básico de um programa, com parte da execução na GPU, segue o seguinte formato:

- 1 - Iniciar o programa na CPU e inicializar os dados;
- 2 - Transferir os dados da memória do sistema para a memória da GPU;
- 3 - Executar os *kernels* para realizar os cálculos e obter os resultados a partir dos dados transferidos;
- 4 - Transferir os resultados de volta para a memória do sistema;
- 5 - Exibir resultados e finalizar.

3.2.2 Hierarquia de *threads*, blocos e grade

A unidade fundamental de execução dos *kernels* na arquitetura CUDA são as *threads*, porém, diferente das CPU's, a organização das mesmas segue outro paradigma. *Threads* são agrupadas em **blocos**, e os blocos, por sua vez, são organizados em uma **grade**. A Figura 11 ilustra o agrupamento de *threads* em blocos e dos blocos em uma grade.

As dimensões da grade e dos blocos são definidas como parâmetros pelo programador e dependem de alguns fatores. Em geral busca-se criar *threads* e blocos suficientes para, ao mesmo tempo, englobar todos os dados do problema e maximizar a ocupação da GPU, ou seja, garantir que a quantidade de *threads* ativas a qualquer momento seja a maior possível. Os principais fatores que influenciam a ocupação da GPU são o tamanho dos blocos, ou seja, a quantidade de *threads* por bloco, quantidade de memória compartilhada utilizada e quantidade de registradores utilizados (ver modelo de execução 3.2.3).

Figura 11 – Hierarquia de grade, blocos e *threads*.

Fonte: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy>

3.2.3 Modelo de execução de CUDA

No início da execução do programa, os blocos de *threads* criados são divididos entre os multiprocessadores da GPU. Cada multiprocessador da GPU que utilizamos nesse trabalho, a Tesla K80, pode ter até 16 blocos e um máximo de 2048 *threads* alocadas a ele simultaneamente. Blocos que não puderam ser alocados no início da execução ficam em espera até que os blocos residentes nos multiprocessadores terminem de executar e o multiprocessador tenha recursos suficientes para alocar para o novo bloco.

Os multiprocessadores executam as *threads* de um bloco em grupos de 32 *threads*, chamados *warps*, que fazem uso do modelo de execução SIMT (*Single Instruction Multiple Thread* (ou Instrução Única, Múltiplas *Threads*)). Todas as *threads* de um *warp* executam a mesma instrução a cada ciclo de *clock*, cada uma processando um elemento (ou subconjunto de elementos) em um posição diferente da memória.

Quando uma *thread* não possui os recursos necessários para executar uma instrução (por exemplo, um dado que precisa ser carregado da memória global da GPU), ao invés de aguardar por esses recursos, é realizada uma troca de contexto. A troca de contexto é realizada a nível de *warp*, portanto o estado de todas as *threads* do *warp* é salvo e um novo *warp* com todos os recursos em mãos é colocado em seu lugar para continuar a execução. Desse modo, é ideal que o volume de trabalho seja grande o suficiente para que sempre que um *warp* tenha que buscar recursos para suas *threads*, haja um outro pronto para continuar sua execução de modo a minimizar a penalidade de latência de memória.

A ocupação da GPU exerce um papel importante no desempenho da aplicação e está relacionada aos conceitos mencionados acima. Cada multiprocessador da GPU possui uma

quantidade finita de registradores, memória compartilhada, assim como um limite máximo de blocos e *threads* ativos ao mesmo tempo. Cada um desses valores é determinado de acordo com a *compute capability* (NVIDIA Corporation, 2010) da GPU.

Devido ao fato de os *warps* executarem sempre em grupos de 32 *threads*, recomenda-se garantir que a quantidade de *threads* por bloco seja um valor múltiplo de 32, de modo a maximizar a sua ocupação. Se um bloco possui 50 *threads*, o multiprocessador ainda irá alocar recursos para dois *warps* de 32 *threads*, mesmo que o segundo *warp* não esteja completo.

Além disso, deve-se tentar dimensionar a quantidade de *threads* por bloco com base nos limites máximos da *compute capability* da GPU sendo utilizada. Para a Tesla K80, por exemplo, se forem utilizadas 128 *threads* por bloco, é possível alocar 16 blocos para cada multiprocessador antes de atingir o limite de 2048 *threads*, alcançando ocupação de 100%. Utilizando 256 *threads*, é possível ocupar o multiprocessador com apenas 8 blocos, porém como o limite de 2048 *threads* foi atingido, a ocupação também é de 100%. Ao utilizar 64 *threads* por bloco, porém, o limite de 16 blocos por multiprocessador é atingido quando apenas 1024 *threads* estão sendo executadas, diminuindo a ocupação para 50%.

Por fim, é necessário considerar a quantidade de memória compartilhada e registradores utilizados pelo código a ser executado pela GPU. Cada multiprocessador possui uma quantidade finita de memória compartilhada disponível para todos os blocos que irão residir nele (112KB no caso da Tesla K80). Novos blocos só são alocados ao multiprocessador quando o mesmo dispõe de recursos suficientes, portanto se cada bloco utilizar, por exemplo, 50KB de memória compartilhada, apenas dois blocos poderão ser alocados de cada vez por multiprocessador. O mesmo vale para a quantidade de registradores.

3.2.4 Hierarquia de memória

Além do conceito de blocos, é importante também mencionar a organização de memória da GPU. Similar ao esquema de *cache* de um processador comum, a hierarquia de memória é dividida da seguinte maneira:

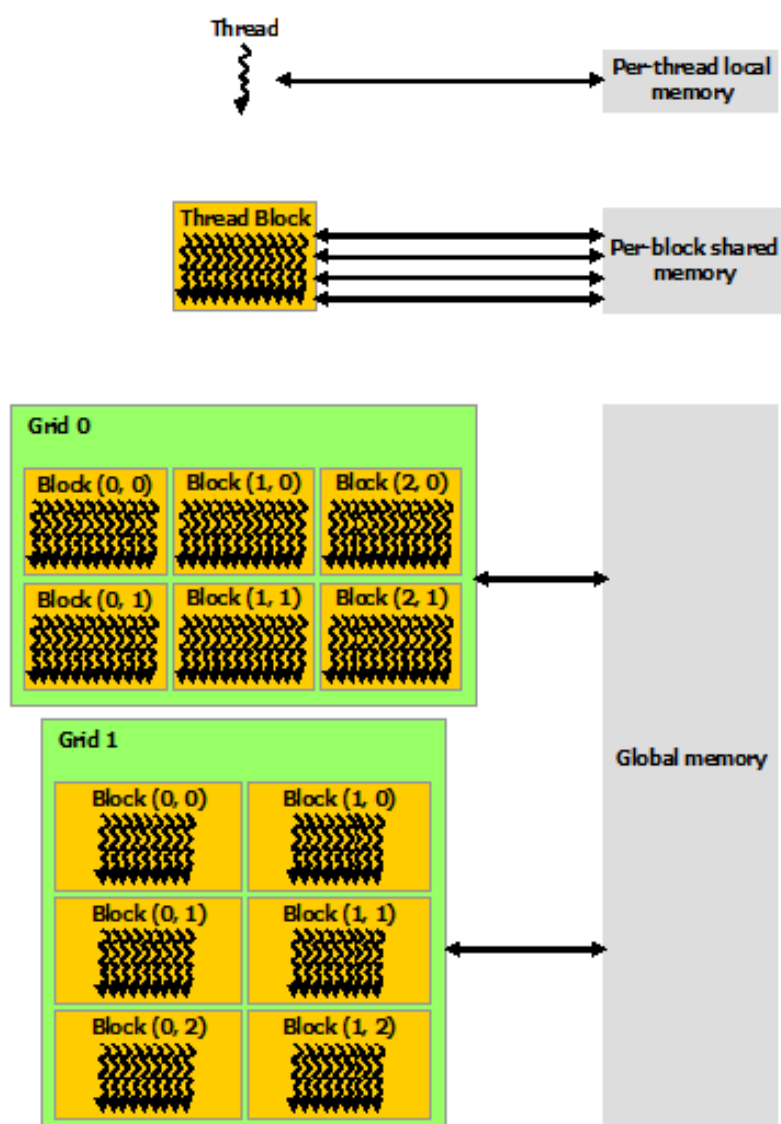
1 - Memória Global: Essa é a memória que estamos acostumados a ver quando analisamos as especificações de uma GPU. GPU's modernas podem chegar a ter 32GB de memória global, porém a capacidade tem como contrapartida o tempo de acesso mais lento entre os tipos de memória. Pode ser acessada por todas as *threads* de qualquer bloco da grade e seu conteúdo persiste entre as chamadas dos *kernels* de uma aplicação.

2 - Memória Compartilhada: Essa é a memória que é alocada para os blocos de execução da GPU. Mais rápida do que a memória global, mas também limitada em capacidade (GPU's modernas chegam a ter até 164KB de memória por multiprocessador). A memória compartilhada é exclusiva a cada bloco durante a execução do programa e apenas as *threads* que residem dentro de um mesmo bloco podem acessá-la.

3 - Registradores: Os dados armazenados nos registradores são privados de cada *thread*. São um recurso extremamente limitado, mas com o tempo de acesso mais rápido de todos. São exclusivos a uma única *thread* durante a execução.

A Figura 12 ilustra a relação entre a hierarquia de *threads*, blocos e grades e a hierarquia de memória. Podemos ver que os registradores de uma *thread* só podem ser referenciados por ela mesma, a memória compartilhada pode ser acessada por todas as *threads* dentro de um único bloco, pois a memória compartilhada é exclusiva para cada bloco, e a memória global pode ser referenciada por qualquer *thread* dentro de qualquer bloco da grade.

Figura 12 – Hierarquia de memória



Fonte: Site da NVIDIA

Disponível em: <<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>>

Disponível em: 23 de novembro de 2019.

3.2.5 Estratégias básicas de otimização

Conforme já mencionado anteriormente, a programação com GPU's requer cuidados para que o programador não se depare com gargalos comuns que podem surgir na hora de executar os programas. Abaixo estão listados alguns dos problemas com os quais desenvolvedores costumam se deparar ao construir uma aplicação com GPU.

1. Otimizar transferências de memória

O custo das transferências de dados entre diferentes tipos de memória das GPU's varia drasticamente. Uma das operações mais lentas é a transferência de dados entre a memória RAM do sistema e a memória global da GPU, portanto deve ser sempre minimizada ao máximo. Normalmente é feita a transferência de todos os dados de uma vez só no início do programa e, ao final, é retornado apenas o resultado final dos cálculos para evitar que sejam realizadas repetidas transferências desse tipo. Para efeitos de comparação, a transferência de dados entre a memória do sistema e a memória da GPU (ou entre memória de duas ou mais GPU's no mesmo sistema) é feita através do conector PCI Express da placa mãe. A taxa máxima teórica da conexão PCIE 4.0, para um link com tamanho de 16 faixas (x16), é em torno de 32 GB/s. Já a taxa máxima de transferência teórica entre diferentes áreas da memória global das GPU's K80 da Nvidia usadas nesse trabalho chega a 480 GB/s (NVIDIA Corporation, 2014).

2. Maximizar coalescência de memória

A coalescência de memória é importante pois permite que acessos à memória de múltiplas *threads* de um *warp* sejam combinados em uma única transação. Em GPU's da Nvidia, vetores alocados em memória são alinhados pelo *driver* do CUDA em segmentos de 256 *bytes* e acessos à memória podem ser feitos em transações de 32, 64 ou 128 *bytes*. Dessa forma, se 32 *threads* realizarem um acesso à memória em que as *threads* 0, 1, 2, 3, ..., 31 acessem as posições 0x0, 0x4, 0x8, 0xC, ..., 0x80, todos os acessos serão realizados em uma única transação. Se o mesmo acesso fosse feito, mas começando da posição 0x4, uma transação separada teria que ser feita para que a *thread* 31 pudesse acessar o elemento em memória desejado.

3. Volume de dados adequado

Apesar de as transferências de dados serem lentas, as GPU's costumam compensar esse tempo durante o processamento, pois, como conseguem realizar uma quantidade muito maior de cálculos simultaneamente do que os processadores comuns, conseguem terminar muito mais rapidamente as tarefas que lhes são atribuídas. Um erro comum é tentar aplicar o poder computacional das GPU's para toda e qualquer situação que parece poder ser paralelizável. Em casos como simulações partícula-partícula seria realizar simulações com um número muito reduzido de partículas.

Nesses casos é possível que não haja ganho nenhum ou até mesmo haja perda de desempenho, pois o tempo necessário para transferir os dados para a GPU e depois transferir o resultado de volta para a memória do sistema já é maior do que o tempo que levaria para o processador terminar de realizar os cálculos.

4. Paralelismo de dados adequado

Mesmo que o problema que estamos nos propondo a resolver possua um volume de dados suficientemente grande para que o poder computacional das GPU's possa ser aproveitado é necessário que haja paralelismo de dados. Isso significa que é necessário que a mesma operação possa ser executada em paralelo em diversos elementos de memória, sem que haja dependência entre os resultados. O problema de N-Corpos se encaixa nesse perfil pois, a cada iteração do problema, todas as operações de cálculos de forças entre os corpos podem ser feitas ao mesmo tempo e de maneira independente. Um exemplo em que o paralelismo de dados não se aplica seria na criação de um vetor em que o valor de cada posição deve ser substituído pela soma de todas as anteriores. Apesar de todos os elementos do vetor serem modificados ao fim da execução, para determinar o valor de um deles necessariamente precisamos que todos os elementos anteriores tenham sido calculados para que a soma com o valor da posição atual possa ser feita, tornando esse problema pouco paralelizável a nível de dados.

4 IMPLEMENTAÇÃO PARALELA DO ALGORITMO DE BARNES-HUT COM UM NÚMERO ARBITRÁRIO DE GPU'S

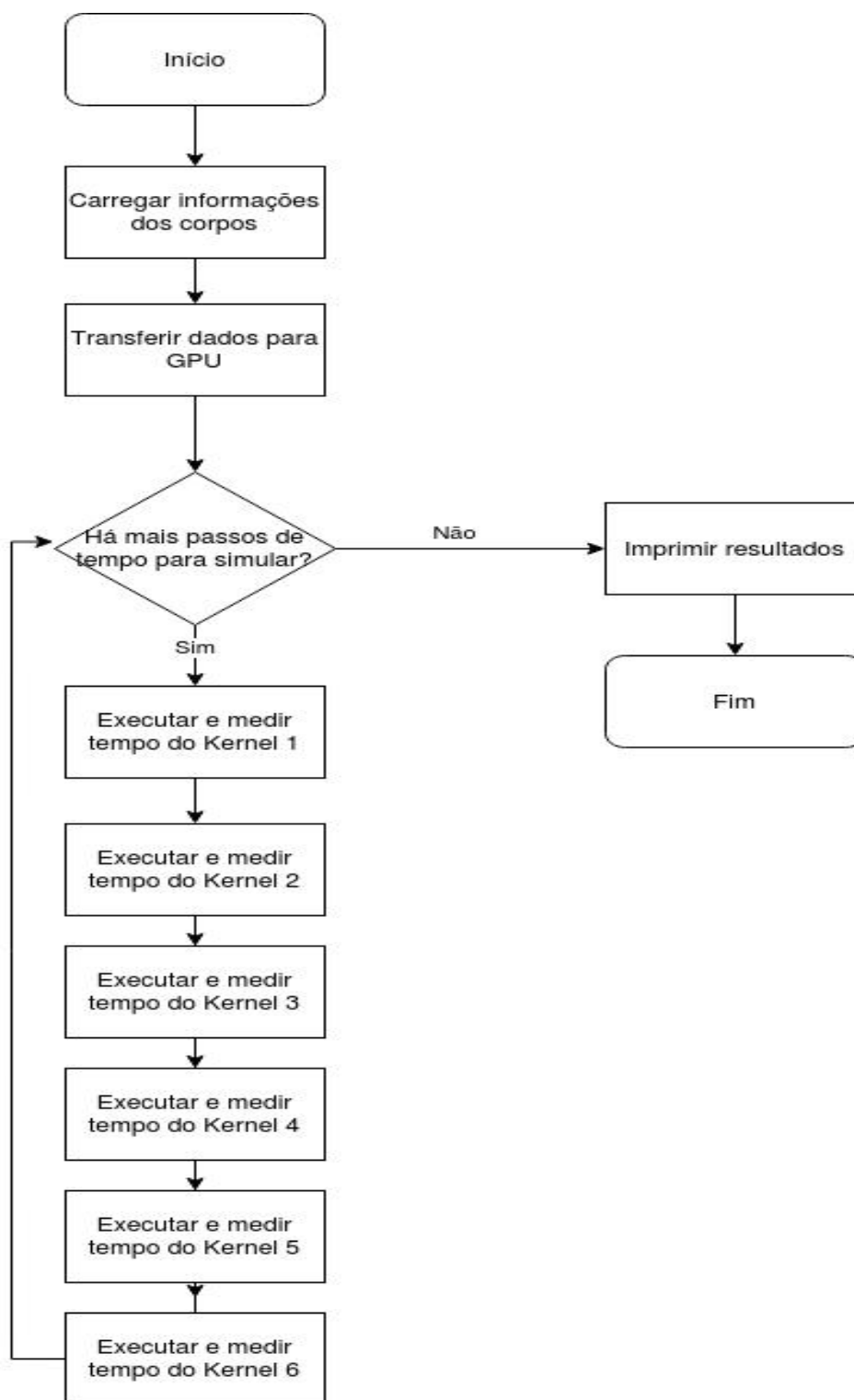
Conforme mencionado anteriormente, neste trabalho tomamos como referência uma implementação do algoritmo de Barnes-Hut em CUDA para execução em GPU (BURTSCHER; PINGALI, 2011a). Neste capítulo, iremos inicialmente apresentar a estrutura dessa implementação de referência (a qual chamaremos de “código original”), apresentando algumas das características dos seus *kernels*. Em seguida, discutiremos a proposta de extensão desenvolvida, na qual alteramos o código original para permitir que um número arbitrário de GPU's possam ser utilizadas para acelerar os cálculos, levando em consideração as dificuldades encontradas, e, por fim, o que foi feito para resolvê-las.

4.1 ESTRUTURA DO CÓDIGO ORIGINAL

O código original foi dividido em 6 *kernels* que executam sequencialmente nas GPU's, seguindo aproximadamente o passo a passo do algoritmo de Barnes-Hut descrito no capítulo 2 para o cálculo das interações entre os corpos. Essa divisão é necessária porque precisamos de uma barreira global entre cada uma das etapas do algoritmo (o código não pode começar a executar o cálculo das interações entre os corpos sem antes ter finalizado totalmente a construção da árvore que representa o espaço, por exemplo). Abaixo segue uma breve descrição de cada um dos *kernels* e a Figura 13 ilustra o fluxo completo do programa:

- *Kernel 1* - Determinar a caixa delimitadora do espaço
 - Esse *kernel* simplesmente identifica o tamanho total do espaço ocupado pelos corpos em cada iteração.
- *Kernel 2* - Construir a *octree* que organiza hierarquicamente os corpos
 - Nesse *kernel* é montada a *octree*, com todos os corpos e nós intermediários (nós que representam todos os corpos naquela subdivisão do espaço).
- *Kernel 3* - Atualizar nós criados no *kernel 2*
 - Percorre a árvore criada, computando o centro de gravidade e a massa de todos os nós intermediários criados no *kernel* anterior.
- *Kernel 4* - Ordenar os corpos em memória
 - Esse *kernel* não é necessário para a corretude do algoritmo em si, mas ajuda a acelerar o *kernel 5* a seguir. Ordena todos os corpos em memória para

Figura 13 – Fluxo de execução original.



garantir que corpos que estejam próximos no espaço também estejam próximos na memória da GPU, maximizando a coalescência das *threads* no acesso à memória no *kernel* seguinte.

- *Kernel 5* - Calcular as forças entre os corpos
 - Percorre a árvore do nó até as folhas calculando as interações para cada corpo da simulação. Para cada corpo, o *Kernel* percorre a árvore calculando as forças e a aceleração resultante em forma vetorial.
- *Kernel 6* - Atualizar informações dos corpos
 - Utiliza as acelerações calculadas no *kernel* anterior para atualizar as velocidades e posições dos corpos envolvidos na simulação.

4.2 EXTENSÃO DO CÓDIGO ORIGINAL PARA EXECUÇÃO EM UM NÚMERO ARBITRÁRIO DE GPU'S

No artigo original publicado por Burtscher e Pingali foi realizada uma análise do desempenho individual de cada um dos *kernels* e ficou aparente que o *kernel 5* representava mais de 80% do tempo de execução total do programa (Figura 14, tabela 6.2 do artigo). Devido a isso, decidimos dividir apenas o processamento do *kernel 5*, pois o *overhead* de transferências de dados da memória de uma GPU para outra poderia anular o benefício da execução em paralelo se os outros *kernels* do programa também fossem paralelizados.

Figura 14 – Medidas de desempenho do código original.

	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5	Kernel 6
CPU runtime	50.0	2,160.0	430.0	310.0	382,840.0	990.0
GPU runtime	0.8	868.0	100.3	38.6	4,202.8	4.1
CPU/GPU	62.5	2.5	4.3	8.0	91.1	241.5

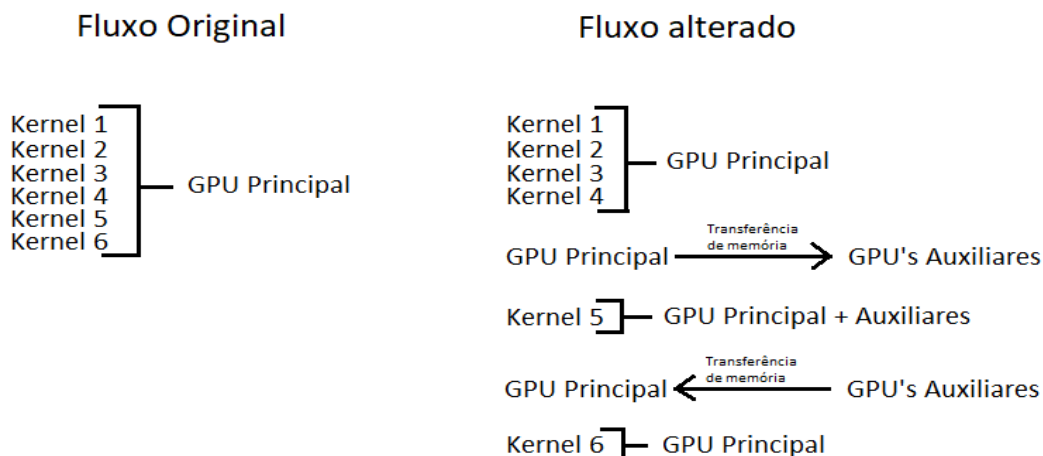
Fonte: (BURTSCHER; PINGALI, 2011b)

Uma das GPU's fica encarregada de executar os *kernels* 1,2,3,4 e 6, e todas dividem a carga e aceleram o processamento do *kernel 5* apenas. Isso garante que os recursos sejam utilizados de maneira eficiente e minimiza a custosa transferência de dados entre as GPU's a cada passo de tempo. A Figura 15 ilustra a diferença entre as execuções do código proposto com relação ao código original.

4.3 PROJETO DA EXPANSÃO DA SOLUÇÃO ORIGINAL

O projeto de expansão desenvolvido nesse trabalho realiza uma divisão igualitária do trabalho do *kernel 5* entre todas as GPU's envolvidas de modo a melhorar o desempenho

Figura 15 – Fluxo de execução original × Fluxo de execução alterado.



final da aplicação. No início da execução do *kernel* 5, todos os dados são transferidos da GPU principal para cada uma das outras GPU's. Em seguida, cada GPU calcula as interações para cada um dos corpos pelo qual é responsável e, por fim, cada GPU transfere de volta para a GPU principal os resultados obtidos. A transferência final é realizada através de uma combinação de uma transferência de memória entre a GPU auxiliar e um vetor temporário na GPU principal e a execução de um sétimo *kernel* para copiar as informações corretas dos corpos para suas devidas posições de memória.

O *kernel* da última etapa é necessário devido ao fato de o código utilizar um vetor separado para realizar a ordenação em memória ao invés de ter que reordenar todos os outros vetores. Toda vez que uma interação deve ser calculada, o código procura no vetor ordenado qual é a posição do corpo nos outros vetores para que os valores sejam atualizados na posição correta. A Figura 16 ilustra essa divisão.

No *kernel* 5, quando o trabalho entre as GPU's é dividido, o que está sendo repartido é o vetor de ordenação. A partir dele é que as GPU's irão determinar quais posições de memória no vetor original elas irão ter que alterar. Com esse comportamento, podemos ver que, ao final do *kernel* 5, cada GPU terá alterado uma porção diferente dos corpos da simulação, mas as alterações não necessariamente estarão em posições contíguas de memória. A Figura 17 demonstra esse comportamento com 2 GPU's.

A criação do *kernel* 7, ver Apêndice A (6) foi necessária para garantir que os resultados de cada GPU sejam copiados para as posições corretas na memória. Esse *kernel* utiliza o vetor ordenado para identificar quais posições de memória foram alteradas com os resultados dos corpos alocados àquela GPU e transfere apenas as posições relevantes para o vetor final. Essa solução necessita de um vetor auxiliar temporário para que os resultados possam ser copiados para a GPU principal, resultando em um consumo maior de memória

Figura 16 – Representação da ordenação em memória (o vetor 1 guarda as informações dos corpos na ordenação inicial, enquanto o vetor 2 armazena a ordenação dos corpos após o *kernel* 4).

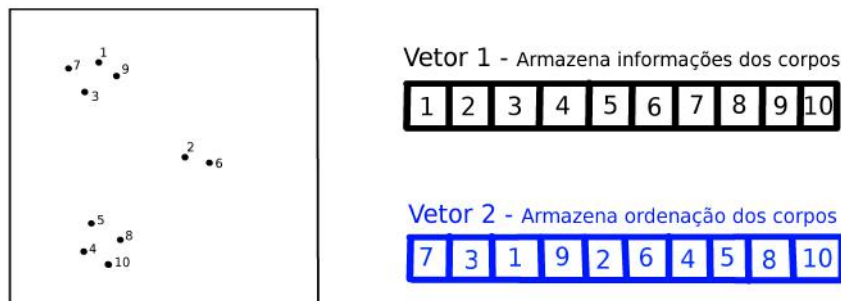
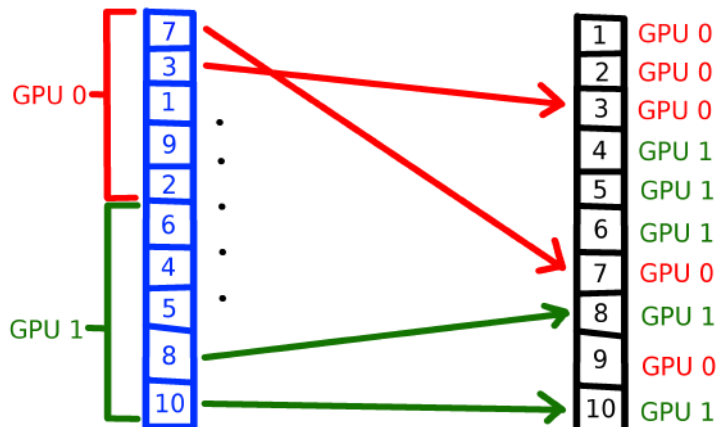


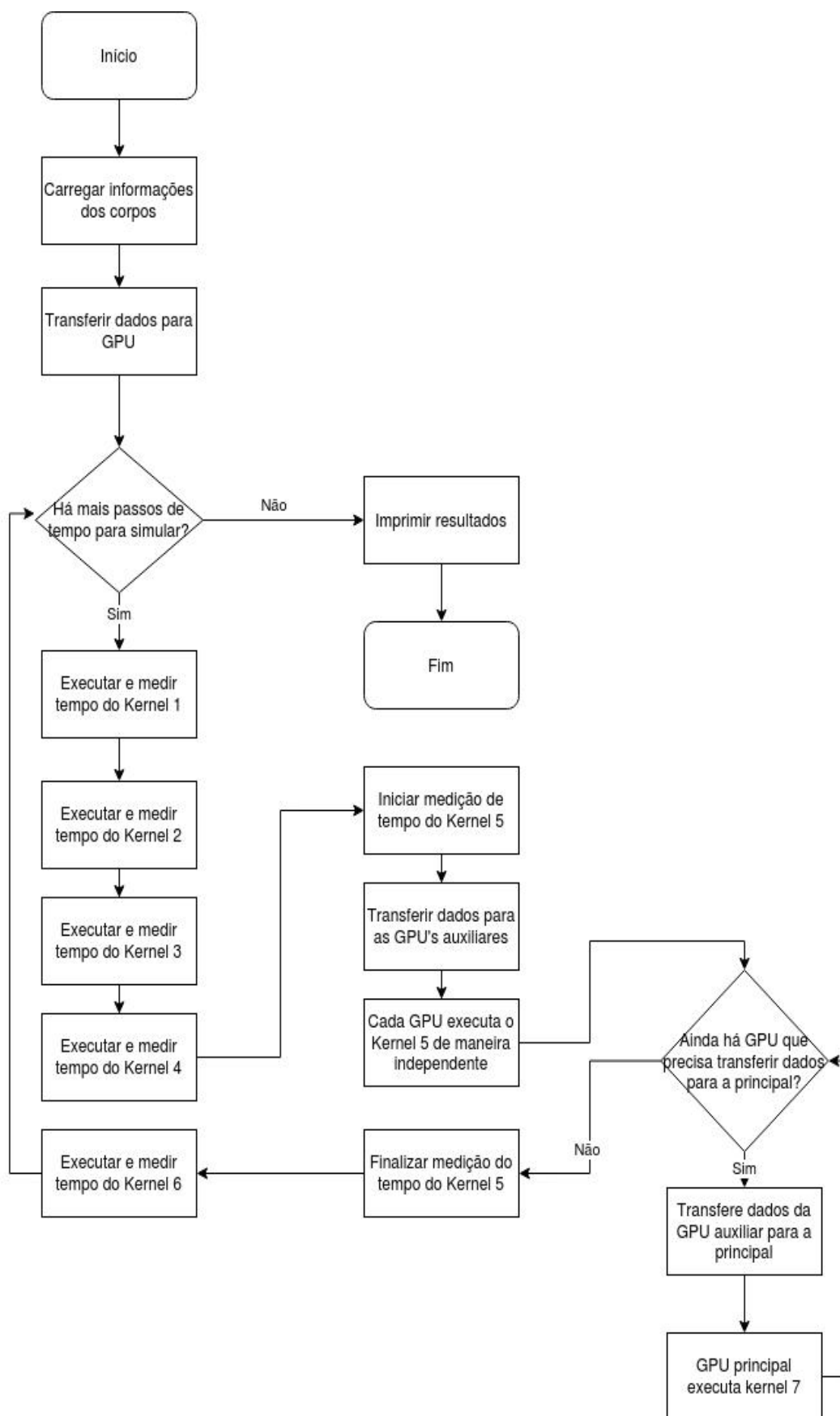
Figura 17 – Alterações feitas pelas GPU's (após cada GPU calcular as interações, os resultados não se encontram sequencialmente em memória).



durante a execução. O código original é capaz de simular até aproximadamente 70 milhões de corpos, enquanto a nossa solução proposta é capaz de simular até 62 milhões de corpos, aproximadamente (uma diminuição de 11.5%).

Em resumo, podemos visualizar as alterações propostas de maneira simplificada por meio do fluxograma da Figura 18. É possível ver que a quantidade de passos aumenta significativamente devido a maior complexidade de gerenciar as transferências de memória entre as GPU's, porém espera-se superar essa maior complexidade com o poder computacional que cada nova GPU adiciona.

Figura 18 – Fluxo de execução alterado.



5 AVALIAÇÃO

Nesse capítulo, será avaliada a solução proposta de execução do algoritmo de Barnes-Hut para solução do problema N-corpos em um número variável de GPUs. Será apresentado o ambiente onde os códigos foram executados, detalhando o equipamento utilizado, as métricas utilizadas para avaliar a diferença de desempenho de cada solução e, por fim, os resultados obtidos.

5.1 PLATAFORMA DE TESTES

Todas as simulações foram executadas na plataforma de computação na Azure, da Microsoft¹. Foi alugada uma instância de máquina virtual da categoria NC24 com as seguintes especificações:

- Processador
 - 24 núcleos virtuais (vCPU) do processador Intel Xeon E5-2690 v3 2.60GHz
- Memória
 - 224GB
- GPU's
 - 4 GPU's Tesla K80
- Sistema Operacional
 - Linux Ubuntu 18

Devido às diversas variações possíveis de conexões entre as GPU's, realizamos alguns testes para identificar a topologia utilizada no servidor da Azure. A Nvidia disponibiliza diversas ferramentas para medição básica de desempenho e gerenciamento das GPU's do sistema. Duas das ferramentas que utilizamos são a Nvidia *System Management Interface* (nvidia-smi) e uma das amostras de código desenvolvidas e disponibilizadas pela própria Nvidia que faz a medição de velocidades de diferentes tipos de transferências de memória (*bandwidthTest*).

Fazendo uso da ferramenta Nvidia *System Management Interface*, podemos utilizar os argumentos *topo -m* para obtermos uma matriz de topologia onde é indicado como cada GPU se comunica com as demais para as transferências de memória. A Figura 19 ilustra o resultado da execução desse comando.

¹ Site da plataforma: <https://azure.microsoft.com/pt-br/>

Figura 19 – Topologia do servidor

```
tccbruno@TCCBruno:~$ nvidia-smi topo -m
      GPU0  GPU1  GPU2  GPU3  CPU Affinity
GPU00  X    NODE  NODE  NODE  0-11
GPU01  NODE X    NODE  NODE  0-11
GPU02  NODE NODE  X    NODE  0-11
GPU03  NODE NODE  NODE  X    0-11

Legend:

X      = Self
SYS    = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
NODE   = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
PHB    = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
PXB    = Connection traversing multiple PCIe switches (without traversing the PCIe Host Bridge)
PIX    = Connection traversing a single PCIe switch
NV#    = Connection traversing a bonded set of # NVLinks
```

De acordo com a matriz e a legenda mostradas na Figura 19, é possível identificar que as comunicações entre as GPU's, além de passar pelas conexões PCIe, também devem passar pelo *host* antes de chegar ao seu destino, limitando o desempenho máximo que pode ser obtido nas transferências de dados. A segunda ferramenta, a *bandwidthTest*, nos permite mensurar a velocidade máxima das transferências que as GPU's podem alcançar. A Figura 20 mostra o resultado da execução dessa ferramenta.

Figura 20 – Velocidade de transferência de memória

```
tccbruno@TCCBruno:~/tmp/cuda-samples/Samples/bandwidthTest$ ./bandwidthTest --memory=pageable --htod --dtoh
[CUDA Bandwidth Test] - Starting...
Running on...

Device 0: Tesla K80
Quick Mode

Host to Device Bandwidth, 1 Device(s)
PAGEABLE Memory Transfers
  Transfer Size (Bytes)      Bandwidth(GB/s)
  32000000                   6.0

Device to Host Bandwidth, 1 Device(s)
PAGEABLE Memory Transfers
  Transfer Size (Bytes)      Bandwidth(GB/s)
  32000000                   6.3
```

Como é possível ver, a taxa de transferência máxima que as GPU's atingem é em torno de 6GB/s devido ao fato de que é necessário que as mesmas passem pelo *host*, não podendo ser feitas diretamente entre as GPU's através das conexões PCIe.

5.2 MÉTRICAS DE AVALIAÇÃO

Como nossa proposta envolve basicamente acelerar a execução do código por meio da adição de GPU's ao processamento, as métricas que serão utilizadas para avaliar o desempenho da solução proposta são o tempo de execução do programa e a aceleração obtida com as alterações. O código original faz medições de tempo de cada *kernel* separadamente e, para obter o tempo total, realiza a soma dos tempos finais de cada um deles. Procedimentos de alocação de memória para CPU/GPU e transferência inicial das informações para a GPU não são considerados.

Para evitar adicionar incertezas na medição e na comparação dos tempos de execução do código original com os tempos de execução da nossa proposta, e considerando que nossa alteração se limita a apenas um dos *kernels*, resolvemos que seria melhor manter a estrutura original. No código original são utilizadas funções específicas de medição de tempo antes e depois da execução para calcular o tempo de execução de cada *kernel* e o tempo total de execução é a soma desses tempos.

De modo a evitar possíveis inconsistências entre as medições e manter o código em um formato que seja possível comparar os resultados com o código original (e considerando que as únicas alterações feitas foram no *kernel 5*), todas as operações realizadas na proposta desse trabalho, incluindo transferências de memória entre GPU's e execução dos *kernels 5* e *7* foram incluídas no código entre as funções de medição de tempo do *kernel 5* original. Dessa forma, o tempo de execução do *kernel 5* com todas as alterações, pode ser diretamente comparado com o do código original, evitando o overhead de realizar medições adicionais para considerar o *kernel 7* e as transferências de memória criadas.

Como ao fim de cada simulação nós tínhamos a informação do tempo de execução de cada um dos *kernels* separados, para fins de comparação, serão apresentados resultados comparando a aceleração relativa obtida se considerarmos o tempo total da simulação, ou seja, a soma de todos os *kernels*, assim como apenas do *kernel 5*. Para cada uma das simulações, foram realizadas três execuções e o tempo final considerado foi a média aritmética das mesmas.

5.3 RESULTADOS OBTIDOS

Todos os resultados apresentados nas tabelas 1 e 2 foram obtidos por simulações com 40 milhões de corpos, gerados automaticamente, da mesma forma como é feito no código original, e 100 intervalos de tempo. Foram escolhidos esses parâmetros para garantir o maior volume possível de trabalho para as simulações com 4 GPU's, maximizando o resultado do paralelismo e mantendo o tempo total de execução das simulações dentro de um limite razoável. O número máximo de corpos que a GPU utilizada (Tesla K80) pode comportar em memória na versão expandida do algoritmo é em torno de 62 milhões de corpos, porém, com essas configurações, cada simulação levaria aproximadamente 28 horas

para finalizar. Cada intervalo de tempo pode representar valores diferentes, dependendo do objetivo da simulação, e não impactam muito na duração total da simulação. A quantidade de passos de tempo sendo simulados é que, de fato, tem um impacto direto no tempo total da simulação. Um aumento de 100 passos de tempo para 1000 passos de tempo, significaria que a simulação demoraria aproximadamente 10 vezes mais tempo para terminar.

Tabela 1 – Métricas obtidas (execução completa)

Quantidade de GPU's	Tempo Médio (s)	Desvio Padrão	Variância
Original (1 GPU)	10596,9853	3,6399	13,2493
1 GPU	10782,7997	0,9765	0,9536
2 GPU's	5509,9688	0,0444	0,0020
3 GPU's	3806,4498	2,4618	6,0605
4 GPU's	3055,6450	4,0977	16,7913

Tabela 2 – Métricas obtidas (apenas do *kernel* alterado)

Quantidade de GPU's	Tempo Médio (s)	Desvio Padrão	Variância
Original (1 GPU)	10525,5040	3,6475	13,3043
1 GPU	10711,4480	0,9059	0,8206
2 GPU's	5438,6607	0,0450	0,0020
3 GPU's	3734,8892	2,4565	6,0346
4 GPU's	2984,1350	4,0969	16,7848

A Figura 21 apresenta um gráfico com as acelerações obtidas, relativas ao tempo de execução do código original e a Figura 22 apresenta um gráfico do tempo de execução de cada uma das simulações realizadas. Foram divididos em duas categorias, a primeira, mediu a aceleração quando considerado o tempo total da simulação (todos os *kernels*), e a segunda mediu a aceleração obtida quando apenas o *kernel* alterado foi considerado, ou seja, compara o *kernel* 5 do código original com a soma do tempo das transferências de

memória para as outras GPU's, execução do *kernel* 5, transferências de memória de volta para a GPU principal, e execução do *kernel* 7 criado.

Podemos ver que a diferença entre as medições é pequena pois os *kernels* 1,2,3,4 e 6 têm um impacto tão pequeno no tempo total que desconsiderá-los não resulta em uma diferença muito grande no resultado final. Apesar disso, é possível claramente ver um benefício substancial a cada GPU que é adicionada, até a quarta GPU. Quando chegamos na quarta GPU, podemos ver que o benefício de adicioná-la corresponde apenas à metade do seu total teórico (3.5 vs 4).

No momento da elaboração desse TCC, a plataforma Azure não dispunha de servidores com mais de 4 GPU's, portanto não conseguimos dizer com certeza em que ponto a adição de mais uma GPU não resulta em melhora no desempenho. Apesar disso, por meio dos dados que coletamos, podemos fazer uma extrapolação e prever que a partir da sexta GPU já não haveria nenhum benefício e a partir da sétima provavelmente começaríamos a observar uma diminuição do desempenho devido ao *overhead* adicional de transferências de memória para todas essas GPU's.

Figura 21 – Resultados de aceleração

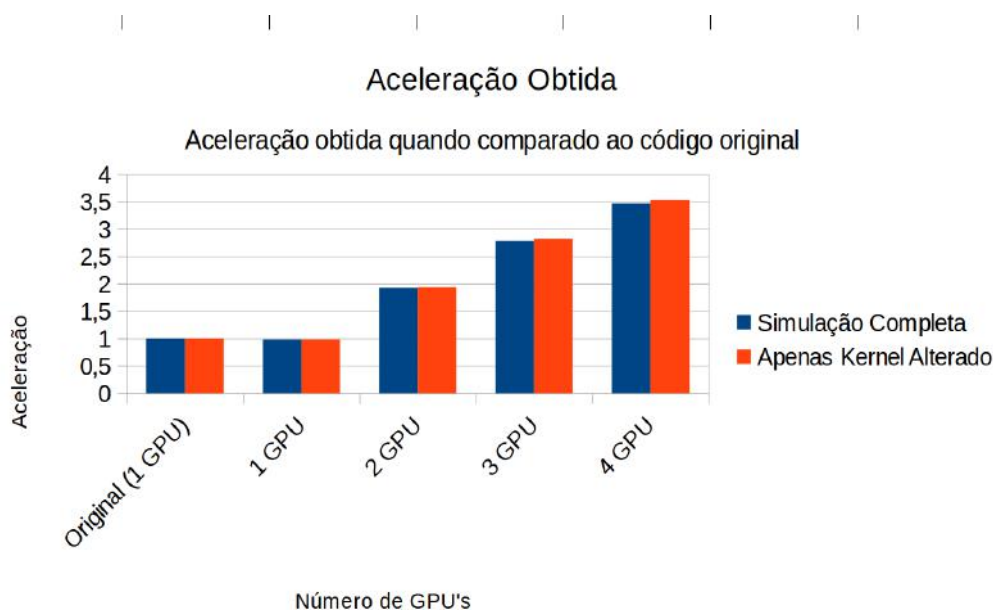


Figura 22 – Resultados de tempo de execução



6 CONCLUSÃO

Neste trabalho, nossa proposta foi implementar alterações no código do trabalho realizado em (BURTSCHER; PINGALI, 2011a) de modo a permitir que um número arbitrário de GPU's possa ser utilizado para simular o problema de N-corpos, em paralelo, utilizando o algoritmo de Barnes-Hut. Foram feitas modificações no *kernel* 5 do trabalho, que representava mais de 80% do tempo de execução, para incluir transferências de memória entre todas as GPU's e dividir igualmente a carga de trabalho entre elas.

A principal conclusão que podemos tirar deste trabalho é que obtivemos sucesso em expandir o código original para utilizar um número arbitrário de GPU's, mas com ressalvas. Assim como em qualquer outro ambiente paralelo, quando incluímos componentes adicionais na execução, há a necessidade de gerenciar diversos aspectos relacionados à interação entre os componentes, e isso gera *overhead* de execução.

Isso ficou óbvio nos resultados apresentados, onde a cada GPU adicionada, o ganho de desempenho obtido não atingiu um valor próximo do máximo teórico devido ao *overhead* extra gerado. Não tivemos recursos para fazer simulações com mais de 4 GPU's, mas é provável que esse comportamento se agrave ainda mais daquele ponto em diante.

Também podemos atribuir os resultados obtidos, em parte, ao fato de que o principal gargalo de desempenho era extremamente localizado em um único *kernel*. Caso o tempo de execução de todos os *kernels* fossem similares, além de apresentar um desafio maior para a paralelização como um todo, poderíamos nos deparar com uma situação em que gerenciar a execução de todos os *kernels* em todas as GPU's fizesse com que o *overhead* fosse tão grande que não veríamos efeito positivo ao adicionar nada além de uma GPU extra.

Além disso, a inclusão de vetores temporários, necessários para realizar as transferências dos resultados das GPU's auxiliares para a GPU principal, resultaram em uma diminuição da quantidade total de corpos que podem ser simulados ao mesmo tempo.

Os infortúnios desse trabalho foram não termos conseguido acesso ao equipamento necessário para testar o desempenho da aplicação com mais de 4 GPU's, além de sermos obrigados a executar nossos testes com uma quantidade bastante limitada de casos de modo a manter os custos de aluguel do servidor sob controle. Apesar disso, acreditamos que as alterações apresentadas nesse trabalho têm a possibilidade de permitir que outros pesquisadores utilizem o potencial completo do equipamento que lhes é disponível, não limitando-os a utilizar apenas uma GPU quando seus servidores podem dispor de duas ou mais GPU's para serem utilizadas.

REFERÊNCIAS

- BARNES, J.; HUT, P. A hierarchical $O(N \log N)$ force-calculation algorithm. **Nature**, Nature Publishing Group, v. 324, n. 6096, p. 446–449, 1986.
- BURTSCHER, M.; PINGALI, K. An efficient CUDA implementation of the tree-based Barnes-Hut N-body algorithm. In: **GPU computing Gems Emerald edition**. [S.l.]: Elsevier, 2011. p. 75–92.
- BURTSCHER, M.; PINGALI, K. **Github Implementation**. [S.l.]: GitHub, 2011. <https://github.com/IntelligentSoftwareSystems/GaloisGPU>.
- C. PINEL N., V. N. L. Alignment-free visualization of metagenomic data by nonlinear dimension reduction. **Scientific Reports**, v. 4, n. 4516, 2014.
- GRAMA, A.; KUMAR, V.; SAMEH, A. Scalable parallel formulations of the Barnes–Hut method for N-body simulations. **Parallel Computing**, v. 24, n. 5, p. 797 – 822, 1998. ISSN 0167-8191.
- HEER, J. **Github Implementation**. [S.l.]: GitHub, 2017. <https://jheer.github.io/barnes-hut/>.
- MAATEN, L. van der. Accelerating t-sne using tree-based algorithms. **Journal of Machine Learning Research**, v. 15, n. 93, p. 32213245, 2014. Disponível em: <http://jmlr.org/papers/v15/vandermaaten14a.html>.
- MEURER, B. B. et al. **Implementação paralela em GPU do algoritmo de Barnes-Hut para solução do problema N-corpos**. 2018. IV Escola Regional de Alto Desempenho do Rio de Janeiro, Fórum de Iniciação Científica, Niteroi, RJ.
- MUNIER, B. et al. On the parallelization and performance analysis of Barnes–Hut algorithm using Java parallel platforms. **SN Applied Sciences**, Springer, v. 2, n. 4, p. 1–13, 2020.
- MUNSHI, A. The OpenCL specification. In: IEEE. **2009 IEEE Hot Chips 21 Symposium (HCS)**. [S.l.], 2009. p. 1–314.
- NVIDIA Corporation. **NVIDIA CUDA C Programming Guide**. 2010. Accessed: 2020-03-21.
- NVIDIA Corporation. **Tesla K80**. 2014. <https://www.nvidia.com/pt-br/data-center/tesla-k80/>. Accessed: 2020-06-29.
- REUTER, M. A. et al. Hiv-specific cd8+ t cells exhibit reduced and differentially regulated cytolytic activity in lymphoid tissue. **Cell reports**, Elsevier, v. 21, n. 12, p. 3458–3470, 2017.
- SINKAROV, A. et al. SaC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 26, n. 4, p. 952–971, 2014.

YI RUOYU LI, M. S. X. Generating chinese classical poems with rnn encoder-decoder. **Chinese Computational Linguistics and Natural Language Processing Based on Naturally Annotated Big Data**, v. 10565, p. 211–223, 2017.

APÊNDICES

APÊNDICE A – CÓDIGO DO KERNEL 7 CRIADO PARA TRANSFERÊNCIA
DE MEMÓRIA.

Código 1 – Kernel 7

```
void TransferKernel(int startnbodiesd, int endnbodiesd, volatile int *
    __restrict sortd, volatile float * __restrict accxd, volatile float *
    __restrict accyd, volatile float * __restrict acczd, volatile float
    * __restrict temp_accxd, volatile float * __restrict temp_accyd,
    volatile float * __restrict temp_acczd){
int tid, i;

for(tid = startnbodiesd + threadIdx.x + blockIdx.x * blockDim.x; tid <
    endnbodiesd; tid += blockDim.x * gridDim.x){
    i = sortd[tid];

    accxd[i] = temp_accxd[i];
    accyd[i] = temp_accyd[i];
    acczd[i] = temp_acczd[i];

}
}
```