Micael Peters Xavier

Implementação Paralela em um Ambiente de Múltiplas GPUs de um Modelo 3D do Sistema Imune Inato

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional, da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Mestre em Modelagem Computacional.

Orientador: Prof. D.Sc. Marcelo Lobosco

Coorientador: Prof. D.Sc. Rodrigo Weber dos Santos

Juiz de Fora

2013

Ficha catalográfica elaborada através do Programa de geração
automática da Biblioteca Universitária da UFJF,
com os dados fornecidos pelo(a) autor(a)

Xavier, Micael Peters.
Implementação Paralela em um Ambiente de Múltiplas GPUs de um Modelo 3D do Sistema Imune Inato / Micael Peters Xavier. -- 2013.
100 p. : il.

Orientador: Marcelo Lobosco
Coorientador: Rodrigo Weber dos Santos
Dissertação (mestrado acadêmico) - Universidade Federal de Juiz de Fora, ICE/Engenharia. Programa de Pós-Graduação em Modelagem Computacional, 2013.

1. Computação de Alto Desempenho. 2. Sistema Imunológico Humano Inato. 3. Equações Diferenciais Parciais. 4. Ambiente de Memória Distribuída. I. Lobosco, Marcelo, orient. II. Santos, Rodrigo Weber dos, coorient. III. Título.

Micael Peters Xavier

Implementação Paralela em um Ambiente de Múltiplas GPUs de um Modelo 3D do Sistema Imune Inato

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional, da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Mestre em Modelagem Computacional.

Aprovada em 26 de Agosto de 2013.

BANCA EXAMINADORA

Prof. D.Sc. Marcelo Lobosco - Orientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Rodrigo Weber dos Santos - Coorientador
Universidade Federal de Juiz de Fora

Prof. D.Sc. Rafael Sachetto Oliveira
Universidade Federal de São João Del-Rei

Pesq. D.Sc. Eduardo Lucio Mendes Garcia
Laboratório Nacional de Computação Científica

Dedico este trabalho à minha noiva Laís e à minha família pelo amor e apoio incondicionais.

AGRADECIMENTOS

Ter a quem agradecer é uma graça divina, pois isso significa que ao longo dessa caminhada não estive sozinho e que comigo estiveram todos aqueles que de alguma forma desejaram esta vitória. Agradeço primeiramente a Deus. Meus sinceros agradecimentos à minha família, principalmente minha mãe Cláudia, pai Antônio, irmão Maicom e avó Lourdes por terem me dado todo o amor e educação durante minha vida, sempre me ajudando e apoiando em minhas decisões. Agradeço de forma especial à minha noiva Laís, por seu amor, carinho e apoio oferecidos a mim de forma incondicional em todos os momentos de nosso relacionamento. Agradeço a todos os meus amigos e colegas do mestrado e do laboratório FISIOCOMP pelos momentos de ajuda, descontração e companheirismo. Aos meus orientadores, por toda paciência, orientação e ensinamentos dados durante toda a minha vida acadêmica. Muito obrigado Lobosco e Rodrigo. Por fim, agradeço a todos aqueles que de alguma forma fizeram parte dessa trajetória, mas não foram citados aqui, as palavras faltam para agradecê-los.

'Those who cannot remember the past are condemned to repeat it.'

George Santayana

RESUMO

O desenvolvimento de sistemas computacionais que simulam o funcionamento de tecidos ou mesmo de órgãos completos é uma tarefa extremamente complexa. Um dos muitos obstáculos relacionados ao desenvolvimento de tais sistemas é o enorme poder computacional necessário para a execução das simulações. Por essa razão, o uso de estratégias e métodos que empregam computação paralela são essenciais. Este trabalho foca na simulação temporal e espacial, em uma seção tridimensional de tecido, do comportamento de algumas das células e moléculas que constituem o sistema imunológico humano (SIH) inato. Com o objetivo de reduzir o tempo necessário para realizar a simulação, foram utilizadas múltiplas unidades de processamento gráfico (*Graphics Processing Unit*, GPUs) em um ambiente de agregados computacionais. Apesar do alto custo de comunicação imposto pelo uso de múltiplas GPUs, as abordagens e técnicas utilizadas neste trabalho para implementar as versões paralelas do simulador mostraram-se efetivas para alcançar o objetivo de redução do tempo de simulação.

Palavras-chave: Computação de Alto Desempenho. Sistema Imunológico Humano Inato. Equações Diferenciais Parciais. Ambiente de Memória Distribuída.

ABSTRACT

The development of computer systems that simulate the behavior of tissues or even whole organs is an extremely complex task. One of the many obstacles related to the development of such systems is the huge computational resources needed to execute the simulations. For this reason, the use of strategies and methods that employ parallel computing are essential. This work focuses on the spatial-temporal simulation of some human innate immune system (HIS) cells and molecules in a three-dimensional section of tissue. Aiming to reduce the time required to perform the simulation, multiple graphics processing units (GPUs) were used in a cluster environment. Despite of high communication cost imposed by the use of multiple GPUs, the approaches and techniques used in this work to implement parallel versions of the simulator proved to be very effective in their purpose of reducing the simulation time.

Keywords: High Performance Computing. Innate Immune System. Partial Differential Equations. Distributed Memory Environment.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

APC *Antigen-Presenting Cell*, 24

API *Application Programming Interface*, 50

CPU *Central Processing Unit*, 37

CUDA *Compute Unified Device Architecture*, 40

DMA *Direct Memory Access*, 46

EDP Equação Diferencial Parcial, 26

GPGPU *General-Purpose Graphics Processing Unit*, 38

GPU *Graphics Processing Unit*, 38

ID Identificador Único, 42

LPS Lipopolissacarídeo, 17

MPI *Message Passing Interface*, 51

MPICH *MPI Chameleon*, 51

OpenMP *Open Multi Processing*, 49

P2P *Peer-to-peer*, 49

PAMP *Pathogen-Associated Molecular Pattern*, 24

PCI-e *Peripheral Component Interconnect Express*, 46

PMN *Polymorphonuclear Neutrophils*, 21

PRR *Pattern Recognition Receptors*, 24

RAM *Random Access Memory*, 75

SIH Sistema Imunológico Humano, 17

SIMD *Single Instruction Multiple Data*, 38

SIMT *Single Instruction Multiple Threads*, 42

SM *Streaming Multiprocessor*, 41

SP *Stream Processor*, 41

UVA *Unified Virtual Addressing*, 48

1 INTRODUÇÃO

1.1 Motivação

O sistema imunológico (ou sistema imune) é de fundamental importância na manutenção da vida de diversas espécies de organismos. Sua principal função é atuar na identificação e eliminação de quaisquer agentes patogênicos externos que tentem invadir o organismo do ser vivo. Ao fazê-lo, tais agentes patogênicos podem ocasionar uma série de doenças no organismo ou até mesmo levá-lo à morte. Além do combate a diversos patógenos, o sistema imunológico também desempenha tarefas de manutenção do organismo, atuando na remoção de células mortas, renovação de determinadas estruturas e eliminação de células anormais que porventura possam dar origem a tumores ou outros danos.

Para atingir tais objetivos, o sistema imune conta com uma complexa rede de células e substâncias que atuam constantemente para promover o perfeito funcionamento e bem estar do organismo. Logo, devido às diferentes relações existentes entre os mais diversos componentes deste sistema em variados níveis de interação, a compreensão de seu total funcionamento é, de fato, uma tarefa extremamente complexa. No entanto, seu entendimento é de fundamental importância no desenvolvimento de vacinas e medicamentos contra diversos tipos de doenças que venham a lesionar o organismo. Uma ferramenta muito útil que pode auxiliar neste objetivo são os modelos matemático-computacionais. Comunidades científicas e pesquisadores da área biomédica fazem uso de tais modelos para realizar várias simulações e testar uma grande quantidade de medicamentos em um curto espaço de tempo sem a necessidade de estudos *in vivo*.

Porém, devido aos altos custos computacionais para a resolução destes modelos, a sua execução em um único computador geralmente é vista como inviável devido à grande quantidade de tempo necessário para se obter os resultados da simulação. Para lidar com este problema, faz-se necessário o emprego de computação paralela de modo que tais modelos possam ser executados mais rapidamente.

1.2 Objetivos

O trabalho de Pigozzo [2] aborda a modelagem da dinâmica de algumas células e moléculas em uma seção unidimensional de tecido humano envolvidas no processo de resposta a um patógeno no sistema imunológico humano (SIH) inato. Devido ao alto custo de execução deste modelo, a abordagem de Rocha [3] faz o uso de uma GPGPU (*General-Purpose Graphics Processing Unit*) para acelerar as simulações em uma seção de tecido expandida para três dimensões. O grande poder de processamento das GPGPUs possibilita a redução no tempo necessário para a execução de uma aplicação paralela, entretanto, dependendo dos parâmetros utilizados no modelo, o uso de múltiplas GPGPUs se torna necessário, de modo a diminuir ainda mais o tempo de execução.

O presente trabalho é baseado em Pigozzo [2] e Rocha [3], e tem por objetivos:

- Criar um novo modelo 3D do SIH inato, ao incorporar ao modelo 3D desenvolvido por Rocha [3] a dinâmica de mais células e substâncias especificadas no modelo unidimensional originalmente proposto por Pigozzo [2];
- Paralelizar a aplicação com o intuito de utilizar múltiplas GPGPUs para reduzir ainda mais o tempo necessário para a execução de uma simulação no novo modelo criado;
- Fazer uso de técnicas e abordagens que visem reduzir o custo de comunicação imposto pelo uso de múltiplas GPUs em um ambiente de agregados computacionais (*cluster*).

1.3 Método

Com a finalidade de atingir os objetivos deste trabalho, em um primeiro momento foi criado um novo modelo 3D do SIH inato, baseado nos trabalhos de Pigozzo [2] e Rocha [3], que simula o comportamento temporal e espacial de oito componentes do SIH inato em uma seção tridimensional de tecido: LPS (uma endotoxina, substância altamente imunogênica), macrófagos ativos, macrófagos *resting*, neutrófilos, neutrófilos apoptóticos, citocina anti-inflamatória, citocina pró-inflamatória e grânulos proteicos.

A partir daí, foi realizada a paralelização da aplicação em uma abordagem híbrida utilizando CUDA (*Compute Unified Device Architecture*), OpenMP (*Open Multi*

Processing) e MPI (*Message Passing Interface*) para que esta pudesse ser executada em um ambiente de agregados computacionais com computadores que possuem múltiplas GPGPUs, acarretando em uma redução no tempo de execução do modelo.

Finalmente, foram criadas três diferentes versões da aplicação: a primeira realiza transferências de dados entre distintas GPGPUs usando a memória principal como espaço intermediário de armazenamento; já a segunda versão utiliza um recurso disponível em alguns modelos de GPGPUs, que possibilita o acesso direto aos dados de cada uma e, por fim, a versão mais elaborada foi desenvolvida na tentativa de sobrepor parte da comunicação realizada pelas GPGPUs com computação dos pontos do tecido.

1.4 Organização

Este trabalho está organizado da seguinte forma. Os próximos três capítulos abordam os principais temas necessários para o entendimento e desenvolvimento deste trabalho. O Capítulo 2 apresenta uma breve explicação sobre o funcionamento geral do sistema imunológico humano, bem como alguns de seus principais componentes. O Capítulo 3 aborda o modelo matemático utilizado neste trabalho, descrevendo todas as suas equações. No Capítulo 4 é apresentado o conceito de computação paralela e são abordadas as tecnologias utilizadas na implementação paralela do modelo 3D. Seguindo a linha de desenvolvimento, o Capítulo 5 descreve como o modelo matemático foi implementado, bem como a paralelização deste, apresentando também as alternativas criadas para lidar com o problema de comunicação entre GPUs. Os resultados obtidos pelo emprego de múltiplas GPUs em um ambiente de agregados computacionais são apresentados no Capítulo 6. Concluindo este trabalho o Capítulo 7 traz as considerações finais e trabalhos futuros.

2 SISTEMA IMUNOLÓGICO HUMANO INATO

No presente capítulo, o sistema imunológico humano (SIH) inato será apresentado, sendo abordados seus principais conceitos e características. A compreensão do funcionamento de seus diversos tipos de células, compostos químicos produzidos por estas, bem como suas interações é de fundamental importância para o entendimento deste trabalho.

Uma vasta literatura existe sobre o papel e funcionamento do sistema imune. Portanto, ao longo deste texto serão apresentados vários conceitos retirados da literatura clássica da área da imunologia [4, 5, 6].

Este capítulo está organizado da seguinte forma. Inicialmente apresenta-se uma visão geral do funcionamento do SIH. Na sequência, será apresentada a endotoxina LPS (lipopolissacarídeo), utilizada neste trabalho para simular uma infecção bacteriana no organismo humano. Por fim, os principais componentes do SIH inato são detalhados.

2.1 Visão Geral do SIH

O corpo humano pode ser descrito como uma máquina que está em constante guerra com o mundo externo. Essa analogia pode ser feita pois o corpo está sob constante ataque de diversos elementos externos que tentam lhe fazer mal, como toxinas, bactérias, fungos, parasitas e vírus. Todos esses agentes são capazes de causar desde pequenas lesões até sérios danos a diversas partes do corpo e, se não fossem devidamente combatidos, o corpo humano não seria capaz de funcionar. Logo, o objetivo do SIH é o de atuar como o próprio exército do corpo, sendo o responsável por sua defesa contra o grande número de agentes e toxinas do meio externo que podem causar infecções.

O SIH é composto pelo sistema imune inato e pelo sistema imune adaptativo ou adquirido. Cada um desses sistemas desempenha um papel fundamental na proteção do corpo humano contra agentes externos, contribuindo significativamente para o perfeito funcionamento e bem estar do organismo.

O primeiro mecanismo de defesa do SIH são as superfícies corporais humanas,

compostas por epitélios. A pele, considerada o maior e mais pesado órgão humano, em conjunto com outros tecidos epiteliais, como os tratos gastrointestinal, respiratório e geniturinário, são as principais barreiras físicas entre os meios interno e externo. É através desse conjunto de barreiras que o corpo humano tenta se proteger do mundo externo.

Apesar dos epitélios formarem um bloqueio efetivo contra agentes externos, em alguns momentos eles podem ser atravessados ou colonizados por patógenos, e estes podem originar uma série de infecções. Felizmente, existem meios de se combater esses agentes invasores do corpo humano: através das respostas imunes.

As células e moléculas que compõem o SIH inato são as responsáveis por desenvolverem uma primeira resposta aos patógenos invasores, tentando destruí-los imediatamente. Entretanto, em determinados momentos os patógenos conseguem transpor ou evadir essa defesa imune inata, e é nesse momento que o SIH adaptativo precisa agir. O SIH adaptativo é capaz de eliminar a infecção com mais eficiência do que o SIH inato, porém, há um certo período de espera para a criação e ativação de determinados componentes que irão efetuar a neutralização e o combate aos patógenos. Esse período é geralmente superior a uma ou duas semanas, variando de indivíduo para indivíduo. Tal condição faz com que o SIH inato seja, de fato, muito importante para a defesa do organismo humano, pois é ele que irá desempenhar, de imediato, o primeiro combate aos invasores e, caso não obtenha sucesso, ativa o SIH adaptativo.

Tabela 2.1: Diferenças entre SI Inato e SI Adaptativo

SI Inato	**SI Adaptativo**
- Encontrado em **quase todas as formas de vida**	- Encontrado **em vertebrados apenas**
- **Sempre funcionando**	- Grande parte do tempo está **inativo**
- **Primeira** linha de defesa	- **Segunda** linha de defesa
- A resposta **independe** do antígeno	- Resposta **depende** do antígeno
- A resposta é **imediatamente** montada e efetuada	- Gasta-se **tempo** para montar e efetuar a resposta
- Reconhecimento de patógenos de maneira **genérica**	- Reconhecimento de antígenos de maneira **altamente específica**
- Resposta de **mesma intensidade** a reexposições	- Resposta **mais eficaz** a reexposições (**memória imunológica**)

Os componentes do SIH adaptativo geralmente estão desativados, entretanto, quando ativados, estes se adaptam à presença dos agentes infecciosos, aperfeiçoando sua resposta com a criação, proliferação e ativação de diversos mecanismos para a neutralização e/ou eliminação dos patógenos. É através do reconhecimento altamente específico do patógeno que a resposta adaptativa consegue obter um melhor resultado contra uma infecção. Após a eliminação dos patógenos, essa resposta aprimorada é então mantida na forma de memória imunológica, e permite que o SIH adaptativo monte uma resposta mais eficiente e em menos tempo a cada vez que o mesmo patógeno específico é encontrado [7].

Na Tabela 2.1 são destacadas algumas diferenças básicas entre os dois tipos de sistema imunológico presentes no organismo humano. Como este trabalho utiliza um modelo que simula o comportamento de algumas células e moléculas do SIH inato, o foco será dado a este tipo de sistema.

Com o intuito de simular uma infecção bacteriana no organismo humano, foi utilizada a endotoxina LPS, que será apresentada a seguir.

2.2 O LPS

O lipopolissacarídeo (LPS) é uma potente endotoxina imunoestimulante encontrada na parede celular de alguns tipos de bactérias e pode induzir uma resposta inflamatória aguda no organismo de uma forma similar a uma infecção bacteriana. Uma resposta inflamatória aguda é aquela produzida pelo corpo no momento em que este se depara com um estresse biológico agudo como, por exemplo, uma infecção causada por micro-organismos. No instante em que as células do SIH inato começam a combater os micróbios invasores, ocorre a liberação da endotoxina LPS no meio, por conta da ruptura da membrana celular desses micro-organismos. Como consequência, a resposta inflamatória é intensificada e outros tipos de células do SIH que auxiliam nesse processo são ativadas. Estudos mostram que o uso de antibióticos pode contribuir eficientemente para a liberação do LPS presente nas bactérias [8].

2.3 Elementos do SIH Inato

Os leucócitos, também chamados glóbulos brancos, são um grupo de células originadas das células-tronco e fazem parte do sistema imunológico humano. Sua função é a de combater

e eliminar micro-organismos e outros elementos que são estranhos ao organismo, atuando em sua captura ou na produção de anticorpos. Os leucócitos podem dar origem a diversos tipos celulares como neutrófilos, monócitos, linfócitos, basófilos e eosinófilos, e, ao fazerem isso, adquirem características e funções específicas. Determinados tipos de células como os neutrófilos, monócitos e macrófagos compõem o SIH e recebem o nome de fagócitos pois sua atuação é na ingestão de bactérias, células mortas e outras substâncias estranhas ao organismo, processo conhecido como fagocitose. A Figura 2.1 ilustra uma célula do tipo fagócito e o processo de fagocitose realizado por ela.

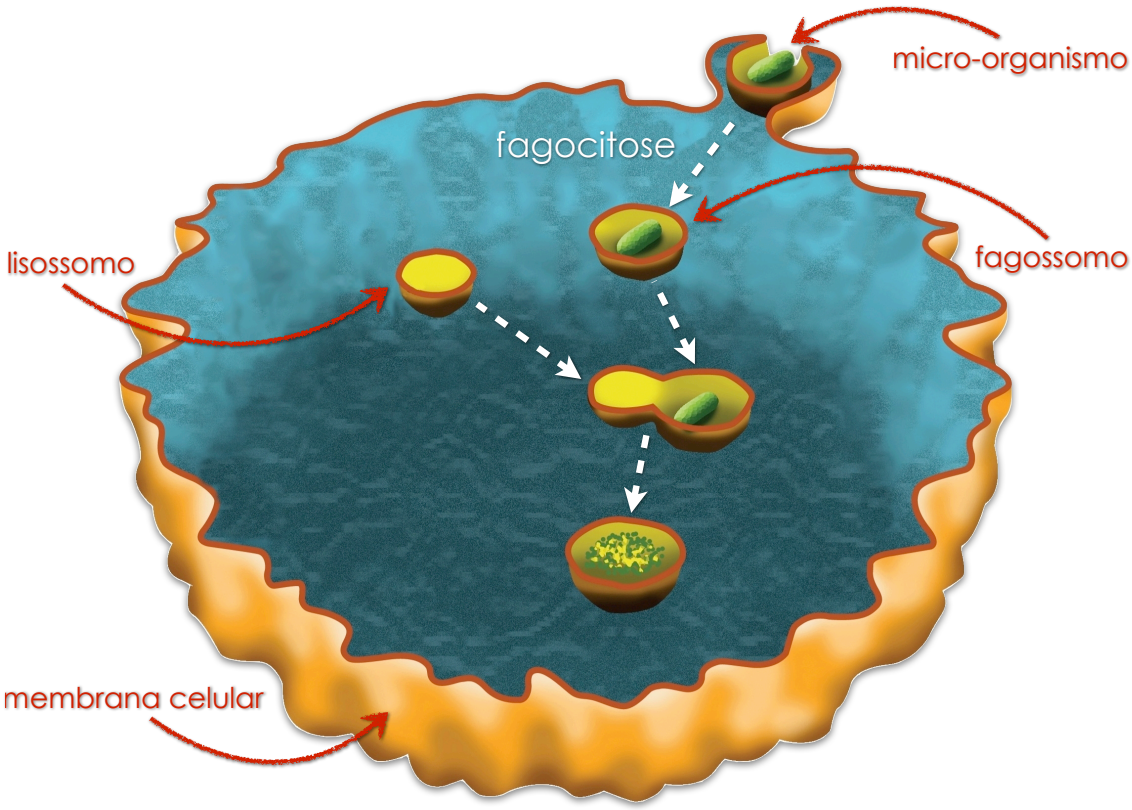


Figura 2.1: Fagócito e o processo de fagocitose (adaptado de [1])

Ao perceber a presença de um micro-organismo em seu entorno, o fagócito começa a expandir sua membrana plasmática de modo a englobar tal partícula, trazendo-a para dentro da célula em uma bolsa denominada fagossomo. A partir daí, uma organela chamada lisossomo, que possui diversas enzimas, se junta ao fagossomo de modo a destruir o micro-organismo por completo, fragmentando-o em diversas partes que posteriormente poderão ser excretadas pela célula.

Além de realizarem a fagocitose, algumas células do SIH também secretam diversas substâncias, como citocinas e quimiocinas, que auxiliam no combate aos invasores,

recrutando mais células ao ponto onde as substâncias foram secretadas. As próximas seções detalham os diversos componentes do SIH inato, como neutrófilos, macrófagos, e substâncias como citocinas, quimiocinas e grânulos proteicos.

2.3.1 Neutrófilos

Os neutrófilos constituem o maior subgrupo de células brancas (leucócitos) presentes no corpo humano. Com a habilidade de reconhecer e neutralizar uma grande variedade de ameaças biológicas, essas são as principais células do SIH inato. Também conhecidos como neutrófilos polimorfonucleares (PMNs - *Polymorphonuclear Neutrophils*), eles realizam a fagocitose de diversos micro-organismos de forma rápida e eficiente graças ao uso de substâncias antimicrobianas armazenadas em seus grânulos citoplasmáticos. Ao realizarem a fagocitose, os neutrófilos podem ou não sofrer apoptose, ou morte celular programada, tornando-se neutrófilos apoptóticos e deixando de realizar algumas de suas atividades, como por exemplo, fagocitar células.

Quando comparados a outros tipos de leucócitos, os neutrófilos apresentam um período de vida relativamente curto, cerca de cinco dias [4], sendo necessária a produção de bilhões de novos neutrófilos a cada dia no organismo humano.

Neutrófilos geralmente são encontrados circulando livremente pela corrente sanguínea no estado inativo, e estão entre as primeiras células do SIH a responderem a um ataque externo contra o organismo. Quando uma ameaça é detectada, tal como uma bactéria, os neutrófilos se tornam ativos e migram para o local de infecção com o objetivo de combater os antígenos invasores.

Por meio da liberação de substâncias originadas de diversos tipos de bactérias, a ativação desses neutrófilos também se torna possível. A capacidade dos neutrófilos serem ativados por citocinas tem sido fonte de estudos em modelos *in vivo* de depleção de neutrófilos [9, 10].

Além de combater os antígenos e produzir substâncias como citocinas, os neutrófilos também são responsáveis pela produção de substâncias chamadas grânulos proteicos que são de grande relevância na resposta imune.

2.3.2 Grânulos proteicos

Os grânulos proteicos são substâncias produzidas pelos neutrófilos que desempenham um papel fundamental na resposta imune. Sua importância é confirmada por estudos que identificam o efeito quimiotático que essas substâncias exercem em determinadas células do SIH, como os monócitos [11, 12, 13]. Isso permite a direta ativação e consequente adesão de mais monócitos ao endotélio dos vasos sanguíneos, contribuindo com o recrutamento de mais monócitos no tecido infectado [14, 15]. Além disso, os grânulos proteicos aumentam a expressão de moléculas de adesão no endotélio, aprisionando de maneira eficiente os monócitos nas veias inflamadas [15].

2.3.3 Citocinas

As citocinas são um conjunto de proteínas secretadas por algumas células que podem alterar tanto o estado quanto o comportamento de determinadas células que possuam receptores adequados ao seu tipo. De acordo com o efeito produzido por sua atuação no local, as citocinas podem ser classificadas como pró-inflamatórias ou anti-inflamatórias. Algumas citocinas, como TNF-α e IL-1, são de muita importância no funcionamento do SIH, pois desempenham papéis centrais na regulação das respostas inflamatórias e imunes [16], atuando como citocinas pró-inflamatórias. Sua produção é realizada principalmente pelos macrófagos, e sua ação no organismo faz com que a permeabilidade vascular seja aumentada, permitindo o recrutamento de novos neutrófilos e macrófagos ao local infectado [17].

Tanto a endotoxina LPS quanto as citocinas como IL-1, IL-2, TNF-α e até mesmo outras substâncias são capazes de fazer com que os neutrófilos sejam ativados. Como consequência, a ação dessas substâncias acaba aumentando consideravelmente a produção da quimiocina IL-8, o que contribui para o recrutamento de novas células [18, 19].

Algumas células como os macrófagos, quando na presença de neutrófilos apoptóticos, produzem citocinas de caráter anti-inflamatório. As citocinas anti-inflamatórias são uma série de moléculas imunoreguladoras que controlam a resposta pró-inflamatória. Citocinas deste tipo, como a IL-10, são importantes na resposta imune, pois impedem a produção de várias citocinas pró-inflamatórias [20], como também inibem a ativação e funções efetoras das células T, monócitos e macrófagos [21].

2.3.4 Quimiocinas

As quimiocinas são uma família de pequenas citocinas ou proteínas de sinalização secretadas por células que causam respostas celulares, incluindo um processo conhecido como quimiotaxia, que direciona células vizinhas a um local desejado no organismo. É através de receptores de quimiocinas que células como neutrófilos e monócitos percebem a presença de quimiocina no meio em que se encontram e se orientam na direção de maior concentração de quimiocina. De acordo com a sua funcionalidade, uma quimiocina pode ser considerada pró-inflamatória ou homeostática. A primeira pode ser induzida durante uma resposta imune para auxiliar o recrutamento de células do SIH ao local de infecção, enquanto que a segunda está envolvida no controle de migração de células durante processos normais de manutenção de tecidos ou desenvolvimento.

A quimiocina IL-8, mencionada anteriormente, é produzida principalmente pelos neutrófilos, macrófagos e outros tipos celulares como as células epiteliais [22]. É classificada como pró-inflamatória, logo possui grande importância no recrutamento de novos neutrófilos e macrófagos ao local da infecção [19]. Experimentos também sugerem que a quimiocina IL-8 esteja relacionada ao processo de reação pró-inflamatória na resposta imune [23, 24, 25].

2.3.5 Macrófagos

Macrófago é um tipo de célula originada no processo de diferenciação de um monócito que deixa a corrente sanguínea e migra para um tecido corporal que foi infectado. Atraídos por substâncias químicas, através da quimiotaxia, esses monócitos migram para o local da infecção e se transformam em macrófagos. A liberação de tais substâncias no meio, como as citocinas e quimiocinas, é realizada por uma variedade de estímulos, incluindo os agentes patogênicos, células danificadas e macrófagos que já residem no local.

Os macrófagos são células de grandes dimensões e podem ser imaginados como "grandes máquinas comedoras de células", já que seu nome, de origem grega, significa "grande comedor". Diferentemente dos neutrófilos, que geralmente possuem apenas alguns dias de vida, os macrófagos apresentam um tempo de vida relativamente longo, podendo sobreviver no organismo por vários meses. São comumente chamados de células sentinelas e podem ser encontrados na maioria dos tecidos corporais, como debaixo da pele, nos tecidos intestinais, nos pulmões e em todas as áreas do corpo humano que estejam

expostas ao mundo exterior.

Os macrófagos desempenham importantíssimas funções no SIH e na manutenção do organismo, participando ativamente em ambas as respostas imunes inata e adaptativa. As tarefas dos macrófagos serão mencionadas abaixo, e logo a seguir, algumas explicações mais detalhadas sobre esses processos serão dadas. É característico dos macrófagos:

- Fagocitar (engolfar e digerir) os micro-organismos invasores;
- Eliminar células mortas e outros restos celulares;
- Secretar proteínas de sinalização (citocinas e quimiocinas) que auxiliam no recrutamento de novas células ao local da infecção;
- Atuar como células apresentadoras de antígenos (APCs - *Antigen-Presenting Cell*);
- Auxiliar no aumento da permeabilidade do tecido epitelial;
- Estimular linfócitos e outras células imunes para responderem ao patógeno.

Os mecanismos que os macrófagos e outras células do SIH dispõem para identificar a presença de um intruso no organismo são alguns receptores presentes em sua superfície. Embora não sejam muitos, esses receptores de reconhecimento de padrões (PRRs - *Pattern Recognition Receptors*) conseguem identificar uma gama muito diversificada de micro-organismos e outras substâncias não próprias do organismo. Isso torna-se possível pois geralmente patógenos possuem em sua composição algumas estruturas que já são conhecidas como padrões moleculares associados a patógenos (PAMPs - *Pathogen-Associated Molecular Pattern*), facilitando sua identificação por parte das células do SIH. Estruturas como os oligossacarídeos ricos em manose, lipopolissacarídeos, peptideoglicanos e DNA CpG não metilado, estão presentes em diversos patógenos, mas não são componentes encontrados nas células do organismo humano.

Existem basicamente dois estados de prontidão dos macrófagos: ativado ou em repouso. Os chamados macrófagos *resting*, ou em repouso, são aqueles que realizam tarefas de limpeza do organismo, eliminando células mortas, proteínas desconhecidas e outros restos celulares. Geralmente são células imóveis, entretanto, podem ser ativadas e se tornar móveis quando estimulados por inflamações. Ao serem ativados por estímulos diversos, como a presença de uma determinada citocina ou o próprio LPS, os macrófagos *resting* passam para um estado onde se tornam ativados.

Na forma ativa, os macrófagos se tornam maiores e conseguem realizar diversas atividades como: a) fagocitar substâncias e células maiores; b) atuar como células apresentadoras de antígenos (APCs) para as células T e B; e c) produzir citocinas e quimiocinas, como a IL-1 e TNF-α, que aumentam a permeabilidade do tecido epitelial e auxiliam no recrutamento de mais células do SIH ao local infeccionado, como também acabam participando do processo de indução da inflamação [26, 27]. Quando estão no estado ativado, nota-se também um significativo aumento na taxa de fagocitose, já que, após ativados, os macrófagos se dedicam principalmente à eliminação do antígeno invasor e à produção de citocinas.

Ao atuarem como APCs, os macrófagos encaminham os antígenos aos linfócitos T e B, para que sejam reconhecidos e devidamente tratados. Linfócitos são células do SIH adaptativo que desempenham um papel importante na resposta imune adaptativa e representam a maioria dos componentes celulares que compõem esse tipo de sistema imune. Pode-se dizer que suas funções primárias são de reconhecer especificamente um antígeno invasor, que pode ter sido apresentado ao linfócito por intermédio dos macrófagos, bem como montar respostas exclusivas para eliminar eficientemente os patógenos específicos e até mesmo células do próprio organismo que foram infectadas por eles.

3 MODELO MATEMÁTICO

A modelagem de um fenômeno natural, por mais simples que este possa ser, é uma tarefa muito complexa. Tal complexidade se dá pela existência de diversas variáveis e fatores que contribuem positiva ou negativamente para que o fenômeno aconteça, e com o SIH (Sistema Imunológico Humano) não é diferente. Devido ao grande número de componentes e suas várias interações, seu entendimento é complexo, exigindo para sua modelagem conhecimento multidisciplinar proveniente da biologia, física e matemática.

Neste trabalho é utilizada uma representação tridimensional de tecido humano desenvolvida por Rocha [3] e um conjunto de EDPs (Equações Diferenciais Parciais) para simular a dinâmica de alguns componentes do SIH quando este é exposto ao lipopolissacarídeo (LPS). Vale ressaltar que o modelo em três dimensões [3] foi derivado de um trabalho anterior de Pigozzo [2], que propôs inicialmente um sistema de EDPs para descrever a dinâmica da resposta imune inata ao LPS em uma seção unidimensional de tecido.

Uma das motivações que levaram ao desenvolvimento do modelo do SIH inato é a existência de inúmeras doenças que surgem como consequência do mau funcionamento deste sistema. Assim, o modelo pode, a longo prazo, ser usado para se estudar as origens desse mau funcionamento como também para testar *in silico* medicamentos que possam auxiliar no processo de sua recuperação. Aliás, é o sistema imune inato o responsável pelas defesas iniciais do organismo, bem como por acionar o sistema imune adaptativo, ressaltando ainda mais sua importância para que o organismo funcione de maneira correta.

No trabalho de Pigozzo [2] foram criados dois modelos: um mais simples, chamado de modelo reduzido, e outro mais detalhado, o modelo estendido. Ambos os modelos basearam-se no trabalho de Su [28] e têm como objetivo descrever a dinâmica de determinados elementos do SIH inato no tecido ao longo do tempo. Ambos os modelos têm como objetivo descrever a dinâmica de determinados elementos do SIH inato no tecido ao longo do tempo. Na versão reduzida somente o LPS, neutrófilos e citocinas são levados em conta em sua construção. Já na abordagem estendida são também considerados os macrófagos, a produção de grânulos proteicos pelos neutrófilos e a resposta anti-inflamatória dinâmica.

Quando comparado ao trabalho de Rocha [3], este trabalho utiliza um novo modelo mais completo e detalhado do SIH inato, que simula a dinâmica de células e moléculas em uma seção tridimensional de tecido humano: em Rocha há somente uma representação do LPS e componentes do SIH como macrófagos, neutrófilos e citocina pró-inflamatória, enquanto que neste trabalho, além das células e substâncias anteriormente citadas, são também representados os neutrófilos apoptóticos, grânulos proteicos e citocina anti-inflamatória. O emprego de uma seção tridimensional de tecido diferencia o modelo usado neste trabalho do modelo proposto originalmente por Pigozzo [2], que utiliza apenas uma seção unidimensional de tecido. Porém a dinâmica de cada um dos componentes simulados é a mesma. Logo é importante destacar que os trabalhos anteriores [2, 3] ofereceram significativas contribuições para que este pudesse ser desenvolvido.

O modelo matemático estendido [2] tem por objetivo simular o comportamento temporal e espacial de determinados elementos no SIH inato quando este é invadido pelo LPS. Através de um sistema de EDPs, o modelo descreve a dinâmica de alguns tipos de células e moléculas do sistema imunológico durante a resposta imune ao LPS no tecido. EDPs são comumente utilizadas na modelagem de diversos fenômenos físicos, pois conseguem capturar mudanças que ocorrem no fenômeno no tempo e espaço.

A seguir, os elementos do SIH inato simulados no modelo estendido [2] como também no modelo criado neste trabalho são especificados juntamente com sua sigla representativa. São eles: endotoxina LPS que faz o papel de um antígeno invasor (LPS), citocina anti-inflamatória (CA), citocinas pró-inflamatórias (CH), grânulos proteicos (G), macrófagos ativados (ou ativos) (MA), macrófagos em repouso (ou *resting*) (MR), neutrófilos (N) e neutrófilos apoptóticos (ND).

A construção do modelo estendido [2] foi baseada no trabalho de Su [28], porém, não há a representação de células dendríticas, células T efetoras, T regulatórias e suas respectivas citocinas como em Su. Outra diferença está relacionada às condições de contorno do LPS, neutrófilos e macrófagos: originalmente foram propostos valores constantes para as populações mencionadas apenas nas bordas do tecido (condição de contorno de Dirichlet) [28], ao passo que o modelo estendido, usado neste trabalho, emprega valores baseados na derivada de uma solução no contorno do domínio (condição de contorno de Neumann).

Logo, para estabelecer esta condição de contorno, foi criada uma equação que modela

a permeabilidade do endotélio e possibilita o cálculo da quantidade de neutrófilos e macrófagos que irão extravasar para todos os pontos do tecido e não somente nas bordas. Fatores como a concentração de citocinas pró-inflamatórias e grânulos proteicos influenciam diretamente no cálculo dessa permeabilidade endotelial.

Dentre os elementos do SIH acima especificados, encontram-se os macrófagos, que são representados no modelo em dois estados distintos: a) em repouso (MR), e b) ativado (MA). No momento em que os macrófagos em repouso identificam a presença do LPS no tecido, estes passam imediatamente para o estado ativado. Os diferentes subconjuntos de grânulos proteicos [29] liberados pelos neutrófilos durante sua transmigração da corrente sanguínea para os tecidos são representados por uma única equação [2].

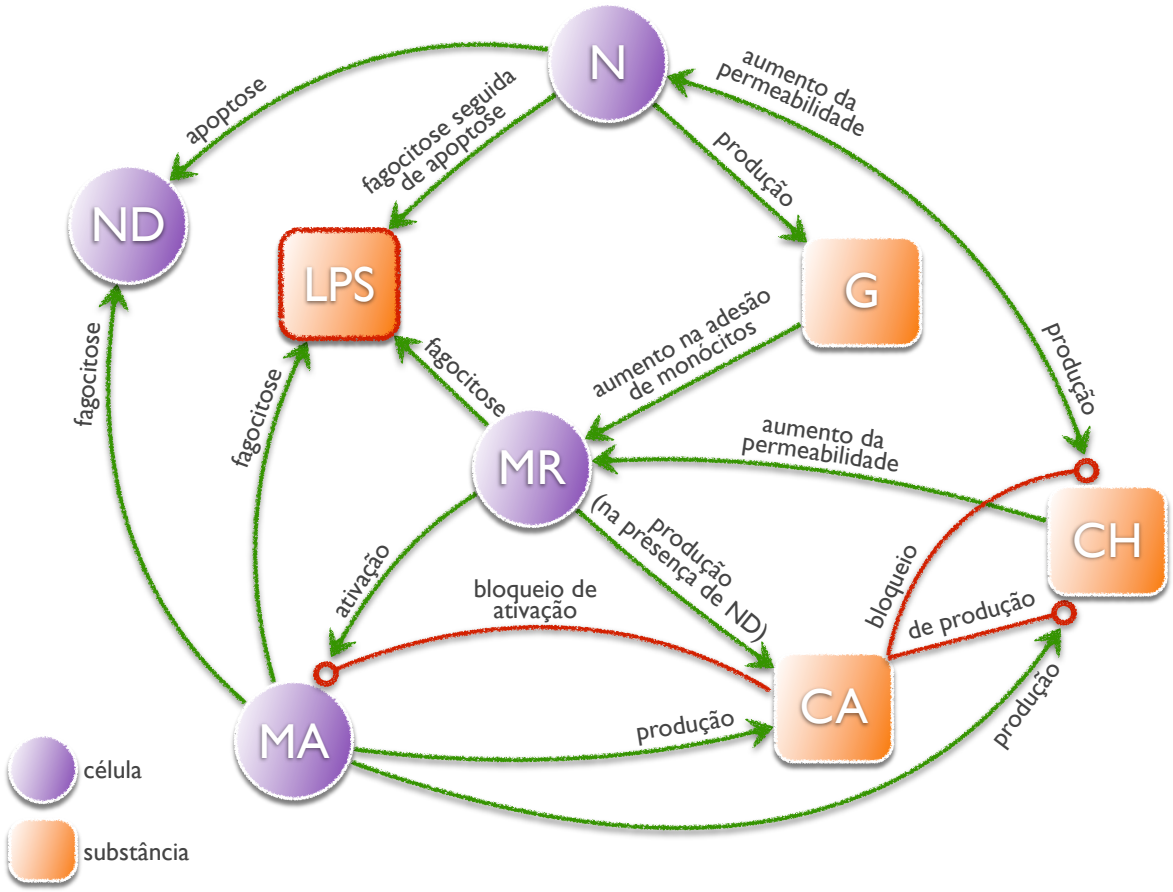


Figura 3.1: Diagrama de relações entre os elementos do modelo (adaptado de [2])

A Figura 3.1 ilustra as relações existentes entre todos os elementos que são simulados no modelo matemático criado neste trabalho. Os macrófagos ativados produzem a citocina anti-inflamatória normalmente, enquanto que os macrófagos *resting* precisam estar na presença de neutrófilos apoptóticos para realizarem tal produção. A citocina

anti-inflamatória IL-10 é importante, pois inibe a ativação e funções efetoras das células T, monócitos e macrófagos [21], logo ela é descrita por uma equação específica.

Já a citocina TNF-α e quimiocina IL-8, ambas pró-inflamatórias, são modeladas em conjunto por uma única equação. Sua produção é realizada pelos neutrófilos e macrófagos ativos, e são responsáveis por aumentar a permeabilidade dos vasos sanguíneos, atuando como uma substância quimioatraente para os monócitos e neutrófilos. Esse aumento de permeabilidade contribui para o recrutamento de uma maior quantidade dessas células ao local da infecção e possibilita que uma resposta mais eficiente ao antígeno invasor seja efetuada.

No momento em que um neutrófilo ou macrófago ativado entra em contato com uma citocina anti-inflamatória, a produção de citocinas pró-inflamatórias realizada por esses tipos celulares automaticamente começa a diminuir. Um outro fenômeno que também ocorre com a ação da citocina anti-inflamatória no meio é a interrupção no processo de ativação dos macrófagos *resting*.

Os neutrófilos são normalmente encontrados na corrente sanguínea e uma de suas funções é a de fagocitar os antígenos, produzir os grânulos proteicos e as citocinas pró-inflamatórias TNF-α e IL-8. O papel que os grânulos proteicos desempenham é permitir a ativação e adesão dos monócitos no endotélio dos vasos sanguíneos, contribuindo para a migração desses monócitos para os tecidos. Quando os monócitos migram da corrente sanguínea para os tecidos, estes recebem o nome de macrófagos.

Os macrófagos ativos e *resting*, em conjunto com os neutrófilos, realizam a fagocitose dos antígenos. A apoptose dos neutrófilos pode se originar ou não do processo de fagocitose dos antígenos. Com a apoptose, os neutrófilos passam para um estado onde deixam de fagocitar células e de produzir as citocinas pró-inflamatórias, se tornando neutrófilos apoptóticos. A quantidade de neutrófilos apoptóticos existente em um determinado local do tecido pode fazer com que a resposta à invasão cause dano ao mesmo. Isso ocorre porque após um certo período no qual os neutrófilos se tornaram apoptóticos, sua fagocitose é necessária. São os macrófagos ativos os responsáveis pela fagocitose dos neutrófilos, e, caso esse processo não ocorra, os neutrófilos apoptóticos acabam sofrendo necrose. Com a necrose os grânulos citotóxicos e as enzimas de degradação são liberadas no meio, podendo causar grandes danos ao tecido.

De uma forma resumida, podem-se destacar as principais características deste

modelo: a) neutrófilos e macrófagos atuam em conjunto para criar uma resposta mais efetiva contra o LPS; b) é através da produção da citocina anti-inflamatória e da fagocitose de neutrófilos apoptóticos que os macrófagos ativados conseguem controlar a resposta imune; c) a citocina anti-inflamatória desempenha um papel crucial na resposta inflamatória: ela evita que a inflamação persista mesmo após o momento em que os antígenos são completamente eliminados do corpo, e d) as concentrações de citocina pró-inflamatória e de grânulos proteicos influenciam diretamente na permeabilidade do endotélio, contribuindo positivamente para o recrutamento de mais neutrófilos e monócitos ao local da infecção.

A seguir, as equações matemáticas que modelam cada componente do SIH no modelo matemático são especificadas, a começar pela equação do LPS. A ordem em que essas equações serão definidas será a seguinte: LPS, macrófagos *resting*, macrófagos ativados, neutrófilos, citocina pró-inflamatória, neutrófilos apoptóticos, grânulos proteicos e citocina anti-inflamatória.

O LPS, é definido pela Equação 3.1:

$$\begin{cases} \dfrac{\partial LPS}{\partial t} &= -\mu_{LPS} LPS - [(\lambda_{N|LPS} N + \lambda_{MA|LPS} MA) * LPS] - \\ & - MR_{activation} + D_{LPS} \Delta LPS \\ MR_{activation} &= \left(\frac{1}{1+\theta_{CA} CA}\right) * \phi_{MR|LPS} * MR * LPS \\ LPS(x,y,z,0) &= LPS_0, \frac{\partial LPS(.,.,.,t)}{\partial n}|_{\partial\Omega} = 0 \end{cases} \tag{3.1}$$

Nesta equação, $\mu_{LPS} LPS$ modela o decaimento do LPS no tecido, e μ_{LPS} representa a taxa de decaimento do LPS. No momento em que um macrófago *resting* reconhece o LPS, ele é ativado e, em seguida, fagocita o LPS reconhecido. O termo representado por $MR_{activation}$ modela a ativação desses macrófagos em repouso, com taxa de ativação $\phi_{MR|LPS}$, e a fagocitose do LPS por eles. A ativação dos macrófagos em repouso é limitada pela ação da citocina anti-inflamatória presente no tecido, ou seja, quanto maior for a concentração de citocina anti-inflamatória no local, menor será a quantidade de macrófagos *resting* que serão ativados, logo, para produzir tal efeito, a parcela $\left(\frac{1}{1+\theta_{CA} CA}\right)$ foi introduzida na equação.

O termo $\lambda_{N|LPS} N$ também simboliza a fagocitose do LPS, mas quando esta é realizada pelos neutrófilos, onde $\lambda_{N|LPS}$ é a taxa de fagocitose por neutrófilos. A fagocitose realizada

pelos macrófagos ativos é representada por $\lambda_{MA|LPS}MA$, com taxa $\lambda_{MA|LPS}$. Logo, o sinal negativo que precede os termos $MR_{activation}$ e $[(\lambda_{N|LPS}N + \lambda_{MA|LPS}MA) * LPS]$ na Equação 3.1 modela a fagocitose, ou consumo, do LPS pelos neutrófilos e macrófagos, simulando a redução na quantidade de LPS no tecido.

Um termo que estará presente na modelagem de todos os componentes do modelo e é de fundamental importância na descrição da dinâmica de cada um é a difusão. A difusão é um fenômeno de transporte de partículas que pode ser definida como a propagação de partículas de regiões com maior concentração para regiões de menor concentração da mesma. O termo $D_{LPS}\Delta LPS$ simboliza a difusão do LPS no tecido, com coeficiente de difusão D_{LPS}. A representação pontual de cada população no espaço tridimensional do tecido é dado pelas coordenadas (x, y, z). Ao final, as condições iniciais e as condições de contorno de Neumann são definidas.

Na Equação 3.2, a dinâmica dos macrófagos em repouso (MR) é modelada. Através da ação das citocinas pró-inflamatórias, o termo $permeability^1_{MR}$ modela o aumento na permeabilidade do endotélio em relação aos monócitos que, ao extravasarem para o tecido, diferenciam-se em macrófagos *resting*. Analogamente, tem-se que $permeability^2_{MR}$ também modela esse aumento de permeabilidade, porém, quando este é induzido pela ação dos grânulos proteicos. Em ambas as equações em que a permeabilidade é modelada, a Equação de Hill [30] foi utilizada como base, sendo possível expressar tais equações em função das concentrações locais de citocinas pró-inflamatórias e grânulos proteicos, respectivamente.

$$\begin{cases} permeability^1_{MR} &= \left(P^{max}_{MR} - P^{min}_{MR}\right) * \left(\frac{CH}{CH+keqch}\right) + P^{min}_{MR} \\ permeability^2_{MR} &= \left(P^{max}_{MR\_g} - P^{min}_{MR\_g}\right) * \left(\frac{G}{G+keqg}\right) + P^{min}_{MR\_g} \\ source_{MR} &= \left(permeability^1_{MR} + permeability^2_{MR}\right) * \\ & \quad * [M^{max} - (MR + MA)] \\ \dfrac{\partial MR}{\partial t} &= -\mu_{MR}MR - MR_{activation} + source_{MR} + \\ & \quad + D_{MR}\Delta MR - \nabla.(\chi_{MR}MR\nabla CH) \\ MR(x,y,z,0) &= MR_0, \frac{\partial MR(.,.,.,t)}{\partial n}|_{\partial\Omega} = 0 \end{cases} \tag{3.2}$$

Além dessas concentrações, outros parâmetros estão envolvidos no cálculo dessas permeabilidades: P^{min}_{MR} e P^{max}_{MR} representam, respectivamente, as taxas mínima e máxima

de aumento da permeabilidade do endotélio aos monócitos, induzido pelas citocinas pró-inflamatórias; as parcelas $P^{min}_{MR\_g}$ e $P^{max}_{MR\_g}$ também modelam as taxas de aumento de permeabilidade, porém, quando estão relacionadas aos grânulos proteicos. Outro termo importante é *keqch*, que representa a concentração de citocina pró-inflamatória que exerce 50% do efeito máximo no aumento da permeabilidade. O mesmo se aplica a *keqg*, mas este está relacionado aos grânulos proteicos.

Com a finalidade de representar a quantidade de monócitos que deixam os vasos sanguíneos e migram para o tecido, se tornando macrófagos *resting*, foi criado o termo fonte $source_{MR}$. Para o cálculo do termo fonte $source_{MR}$ foram levados em conta fatores como a permeabilidade do endotélio, representada por $\left(permeability^1_{MR} + permeability^2_{MR}\right)$, e a capacidade máxima de macrófagos M^{max} suportada pelo tecido, além da quantidade de macrófagos ativados e *resting* já existente no local, calculado por $(MR + MA)$.

A parcela $\mu_{MR}MR$ modela a apoptose dos macrófagos *resting*, onde μ_{MR} é a taxa de apoptose. O termo $MR_{activation}$ é o mesmo já apresentado na Equação 3.1, representando ambas a ativação dos macrófagos *resting* e a fagocitose do LPS por eles. Com o intuito de simbolizar a difusão no tecido dos macrófagos em repouso, foi introduzido o termo $D_{MR}\Delta MR$, onde D_{MR} é o coeficiente de difusão dos macrófagos *resting*.

A quimiotaxia dos macrófagos em repouso é representada pelo fator $\nabla.(\chi_{MR}MR\nabla CH)$, com a taxa de quimiotaxia χ_{MR}. O fenômeno da quimiotaxia é um processo ocasionado pela ação das quimiocinas no organismo, que fazem com que determinadas células de um local sejam direcionadas a uma região onde há uma maior concentração dessas substâncias, geralmente o local da infecção. Finalizando a equação, a condição inicial e a condição de contorno para os pontos do domínio são fornecidas.

Os macrófagos ativos (MA) têm sua dinâmica representada pela Equação 3.3 descrita abaixo:

$$\begin{cases} \dfrac{\partial MA}{\partial t} &= -\mu_{MA}MA + MR_{activation} + D_{MA}\Delta MA - \\ & \quad - \nabla.(\chi_{MA}MA\nabla CH) \\ MA(x,y,z,0) &= MA_0, \frac{\partial MA(.,.,.,t)}{\partial n}|_{\partial\Omega} = 0 \end{cases} \tag{3.3}$$

A apoptose dos macrófagos ativados é definida pela parcela $\mu_{MA}MA$, onde μ_{MA}

é a taxa de apoptose. O termo $MR_{activation}$ é o mesmo encontrado anteriormente nas Equações 3.1 e 3.2, contudo, ao ser precedido por um sinal positivo, indica uma contribuição para o crescimento da população de macrófagos ativados. Na parcela $D_{MA}\Delta MA$ encontra-se representada a difusão dos macrófagos ativados, onde D_{MA} é a taxa de difusão. A expressão $\nabla.(\chi_{MA}MA\nabla CH)$ é muito importante, pois modela a quimiotaxia dos macrófagos ativados, com taxa de quimiotaxia χ_{MA}. Similar às equações anteriores, uma condição inicial e uma condição de contorno de Neumann são estabelecidas.

Outra equação a ser definida (Equação 3.4) é a que modela os neutrófilos (N):

$$\begin{cases} permeability_N &= \left(P_N^{max} - P_N^{min}\right) * \left(\frac{CH}{CH+keqch}\right) + P_N^{min} \\ source_N &= permeability_N * (N^{max} - N) \\ \frac{\partial N}{\partial t} &= -\mu_N N - \lambda_{LPS|N} LPS * N + source_N + \\ & \quad + D_N \Delta N - \nabla.(\chi_N N \nabla CH) \\ N(x,y,z,0) &= N_0, \frac{\partial N(.,.,.,t)}{\partial n}|_{\partial\Omega} = 0 \end{cases} \tag{3.4}$$

A definição de $permeability_N$, que modela a permeabilidade do endotélio aos neutrófilos devido à ação da citocina pró-inflamatória, é análoga à expressão anterior de $permeability_{MR}^1$ da Equação 3.2: uso de taxas mínima e máxima (P_N^{min} e P_N^{max}) para o aumento da permeabilidade, em conjunto com o mesmo $keqch$, previamente definido. Logo, o termo $source_N$ modela o extravasamento de neutrófilos da corrente sanguínea para o tecido, baseando-se na permeabilidade $permeability_N$, na capacidade máxima N^{max} de neutrófilos suportada pelo tecido e na quantidade de neutrófilos N já existente no local.

A parcela $\mu_N N$ representa a apoptose dos neutrófilos, com taxa de apoptose μ_N. A apoptose dos neutrófilos, quando induzida pela fagocitose do LPS, é modelada pelo termo ($\lambda_{LPS|N} LPS * N$), onde $\lambda_{LPS|N}$ é a taxa da apoptose induzida. Já $D_N \Delta N$ simboliza a difusão dos neutrófilos no tecido, com D_N representando o coeficiente de difusão. Finalizando a definição da Equação 3.4, a parcela $\nabla.(\chi_N N \nabla CH)$ modela a quimiotaxia dos neutrófilos com taxa de quimiotaxia χ_N e as condições inicial e de contorno são dadas.

A Equação 3.5 foi criada com o intuito de representar um modelo para as citocinas e quimiocinas pró-inflamatórias (CH), que atuam como substâncias quimioatraentes para os macrófagos ativados, monócitos e neutrófilos, recrutando uma quantidade maior desses

elementos ao local infectado no tecido.

$$\begin{cases} \dfrac{\partial CH}{\partial t} & = \left[LPS * \left(\beta_{CH|N} N + \beta_{CH|MA} MA\right) * \left(1 - \frac{CH}{chInf}\right) * \right. \\ & \left. * \left(\frac{1}{1+\theta_{CA} CA}\right)\right] - \mu_{CH} CH + D_{CH} \Delta CH \\ CH(x,y,z,0) & = CH_0, \frac{\partial CH(.,.,.,t)}{\partial n}|_{\partial\Omega} = 0 \end{cases} \tag{3.5}$$

O termo que modela o decaimento de citocinas pró-inflamatórias é $\mu_{CH} CH$, com taxa de decaimento μ_{CH}. A produção de citocinas pró-inflamatórias realizada pelos neutrófilos e macrófagos ativados é calculada respectivamente pelas parcelas $\beta_{CH|N} N$ e $\beta_{CH|MA} MA$. Os valores de $\beta_{CH|N}$ e $\beta_{CH|MA}$ representam, respectivamente, as taxas de produção de citocina pró-inflamatória realizada pelos neutrófilos e macrófagos em aproximadamente um dia de atividade. Um ponto importante nessa equação é a definição de uma saturação na produção de citocinas pró-inflamatórias, calculada pela parcela $\left(1 - \frac{CH}{chInf}\right)$, onde $chInf$ representa a concentração máxima de citocinas pró-inflamatórias suportada pelo tecido. Outra característica importante é a presença de citocina anti-inflamatória no meio. Ela faz com que ocorra uma diminuição na produção de citocina pró-inflamatória, atuando como uma reguladora da resposta imune. Logo, para reproduzir tal efeito a parcela $\left(\frac{1}{1+\theta_{CA} CA}\right)$ foi inserida na equação.

A difusão das citocinas pró-inflamatórias no tecido é especificada pelo termo $D_{CH} \Delta CH$, com taxa de difusão D_{CH}. Ao final, são estabelecidas as condições inicial e de borda para todos os pontos do domínio.

A apoptose dos neutrófilos, oriunda ou não do processo de fagocitose do LPS, faz com que os neutrófilos passem para um estado onde deixam de fagocitar células e de produzir as citocinas pró-inflamatórias, tornando-se neutrófilos apoptóticos (ND). Sua dinâmica é definida logo a seguir, pela Equação 3.6:

$$\begin{cases} \dfrac{\partial ND}{\partial t} & = \mu_N N + \lambda_{LPS|N} LPS * N - \lambda_{ND|MA} ND * MA + \\ & + D_{ND} \Delta ND \\ ND(x,y,z,0) & = ND_0, \frac{\partial ND(.,.,.,t)}{\partial n}|_{\partial\Omega} = 0 \end{cases} \tag{3.6}$$

Nesta equação, ambos os termos $\mu_N N$ e $(\lambda_{LPS|N} LPS * N)$ representam a apoptose dos neutrófilos, sendo que o último é devido à fagocitose do LPS, com uma taxa de apoptose $\lambda_{LPS|N}$. Para modelar a fagocitose dos neutrófilos apoptóticos pelos macrófagos

ativados foi criada a parcela ($\lambda_{ND|MA}ND * MA$), onde $\lambda_{ND|MA}$ é a taxa de fagocitose. Outro termo importante é $D_{ND}\Delta ND$, que simboliza a difusão dos neutrófilos apoptóticos com coeficiente de difusão D_{ND}. A condição inicial e a condição de contorno são estabelecidas ao final da Equação 3.6.

A equação do grânulo proteico (G) é representada abaixo pela Equação 3.7:

$$\begin{cases} \dfrac{\partial G}{\partial t} &= -\mu_G G + \alpha_{G|N} * source_N * \left(1 - \frac{G}{gInf}\right) + D_G \Delta G \\ G(x,y,z,0) &= G_0, \frac{\partial G(.,.,.,t)}{\partial n}|_{\partial\Omega} = 0 \end{cases} \tag{3.7}$$

Através do sinal negativo que precede o termo $\mu_G G$, o decaimento dos grânulos proteicos pode ser especificado, onde a taxa de decaimento é μ_G. Como previamente visto na Equação 3.5, aqui na Equação 3.7 há também uma expressão que irá definir um ponto de saturação na produção, só que agora a produção é de grânulos proteicos. A expressão ($\alpha_{G|N} * source_N$) representa essa produção de grânulos proteicos realizada pelos neutrófilos devido ao seu extravasamento dos vasos sanguíneos para o tecido infectado, onde $\alpha_{G|N}$ é uma constante adimensional. Tal produção está sujeita a uma saturação definida pela parcela $\left(1 - \frac{G}{gInf}\right)$, com $gInf$ representando a concentração máxima de grânulos proteicos suportada pelo tecido. O termo $D_G\Delta G$ especifica a difusão dos grânulos proteicos, com taxa de difusão D_G. A definição da Equação 3.7 termina com uma condição inicial e uma condição de borda.

Finalizando o conjunto de equações do modelo matemático, a última equação a ser definida é aquela que irá modelar a ação da citocina anti-inflamatória (CA) no meio (Equação 3.8).

$$\begin{cases} \dfrac{\partial CA}{\partial t} &= \left(\beta_{MR|ND} MR * ND + \alpha_{CA|MA} MA\right) * \left(1 - \frac{CA}{caInf}\right) - \\ & \quad - \mu_{CA} CA + D_{CA} \Delta CA \\ CA(x,y,z,0) &= CA_0, \frac{\partial CA(.,.,.,t)}{\partial n}|_{\partial\Omega} = 0 \end{cases} \tag{3.8}$$

O termo que modela o decaimento da citocina anti-inflamatória é simbolizado por $\mu_{CA}CA$, onde μ_{CA} é a taxa de decaimento. A parcela ($\beta_{MR|ND}MR*ND$) representa a produção de citocina anti-inflamatória pelos macrófagos em repouso quando na presença de neutrófilos apoptóticos, com taxa de produção $\beta_{MR|ND}$. Quando a produção de citocina anti-inflamatória é realizada pelos macrófagos ativados, utiliza-se o termo $\alpha_{CA|MA}MA$,

onde $\alpha_{CA|MA}$ é a taxa de produção. Outra vez percebe-se uma saturação nessas produções, sendo definida por $\left(1 - \frac{CA}{caInf}\right)$, com $caInf$ representando a concentração máxima de citocinas anti-inflamatórias suportada pelo tecido. Finalizando as definições da última equação, $D_{CA}\Delta CA$ modela a difusão da citocina anti-inflamatória com coeficiente de difusão D_{CA} e as condições inicial e de contorno de Neumann são estabelecidas.

Nesse capítulo foram definidas as equações matemáticas que descrevem a dinâmica de alguns elementos do SIH inato simulados neste trabalho. Os parâmetros que fazem parte das equações apresentadas anteriormente podem ser encontrados no apêndice deste trabalho, na Tabela A.2. Todos os parâmetros utilizados foram retirados de trabalhos anteriores [2, 3, 31, 32].

4 COMPUTAÇÃO PARALELA

Atualmente a ciência computacional pode ser considerada um dos pilares da ciência, já que está presente em grande parte das atividades de pesquisa do século 21. O uso de computadores em quaisquer áreas do conhecimento é hoje de caráter fundamental para a análise e o entendimento de diversos fenômenos, bem como para a descoberta de tecnologias inovadoras sob vários aspectos. Com seu alto poder de processamento, essas máquinas evoluíram de tal maneira que são capazes de processar uma grande quantidade de dados e fornecer resultados em questão de segundos. Comunidades científicas utilizam os computadores com muitos propósitos, dentre eles a modelagem de fenômenos físicos, testes de medicamentos, análise de grande quantidade de dados, testes de diferentes cenários e solução de problemas de alta complexidade.

Embora a CPU (*Central Processing Unit* ou Unidade Central de Processamento) dos computadores atuais consiga realizar uma quantidade de cálculos da ordem de bilhões de operações por segundo, existem diversas aplicações que demandam um poder computacional ainda muito maior que este, obrigando o uso da computação paralela.

A computação paralela é uma forma de processar diversas instruções de uma maneira simultânea, o que pode contribuir significativamente para a redução do tempo necessário para a completa execução da aplicação. Comparado à computação sequencial, a computação paralela é muito mais adequada para expressar uma ampla classe de problemas do mundo real, tais como modelos e simulações de mudanças climáticas, movimentos planetários, robótica, placas tectônicas, corpo humano, física nuclear, exploração de petróleo, etc. A seguir são listadas algumas das principais razões em se utilizar a computação paralela:

- Reduzir o tempo necessário para solucionar um problema;
- Resolver problemas mais complexos e de grande escala;
- Prover concorrência na computação, ou seja, permitir a execução de tarefas simultaneamente;
- Ultrapassar as limitações impostas a um único computador como quantidade de memória, poder de processamento, etc.

Para que uma aplicação seja executada de forma paralela, torna-se necessário o emprego de técnicas de programação que irão definir o que deverá ser feito por cada elemento participante da computação, de modo a explorar o processamento simultâneo.

Um modelo de computação largamente empregado em computação paralela é o SIMD (*Single Instruction Multiple Data*) definido pela taxonomia clássica de Flynn [33]. Embora essa classificação tenha sido originalmente criada para caracterizar um tipo de arquitetura de computador em que uma única instrução é capaz de processar simultaneamente múltiplos dados, sua definição é comumente utilizada como abordagem de operação em computação paralela.

Além do modelo de computação a ser escolhido, é também necessário levar em consideração a arquitetura do *hardware* utilizado, sendo que em muitos casos este pode orientar na escolha de qual técnica de programação utilizar.

Ao longo deste capítulo serão abordados temas relacionados à computação em paralelo, como tipos de arquiteturas e técnicas de paralelismo utilizadas atualmente.

Este capítulo está organizado da seguinte forma. Inicialmente será apresentado o *hardware* de uma placa gráfica, com ênfase em suas características e nos recursos empregados no desenvolvimento deste trabalho. A Seção 4.2 discorre sobre o uso de *threads* em computação paralela com memória compartilhada e, finalizando o capítulo, a última seção faz uma introdução sobre o uso de um padrão de comunicação para realizar computação em agregados computacionais (*clusters*), onde cada elemento de processamento possui seu próprio espaço de endereçamento.

4.1 GPGPU

A unidade de processamento gráfico de propósito geral (GPGPU - *General-Purpose Graphics Processing Unit*), ou apenas GPU (*Graphics Processing Unit*), é uma placa de vídeo que possui grande poder de processamento, isso graças ao grande número de unidades de processamento disponíveis para processar dados simultaneamente. Embora primeiramente projetadas com a exclusiva função de processamento gráfico, as GPUs acabaram se tornando processadores paralelos para computação de propósito geral.

Em relação ao número de núcleos de processamento (ou *cores*), essas placas se destacam, contando com centenas de *cores* contra apenas algumas unidades de

processamento presentes nas CPUs (Figura 4.1).

Ao contrário de uma CPU, a grande maioria dos componentes de uma GPU são unidades dedicadas a operações de cálculos, contendo apenas poucos elementos de controle lógico e uma pequena quantidade de memória *cache*. Tal característica implica que uma

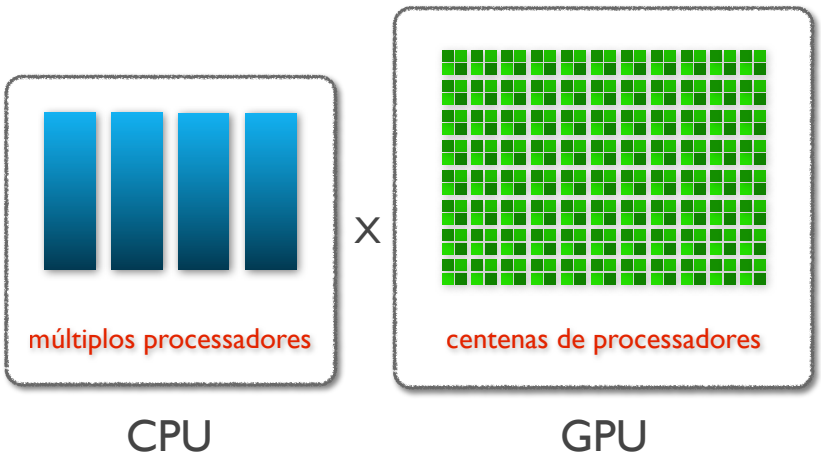


Figura 4.1: Comparação do número de núcleos de processamento entre CPU e GPU

grande porção da área do *chip* físico de uma GPU seja ocupada por processadores, deixando um pequeno local para a unidade de controle lógico e memória *cache*. Isso reflete em uma ótima relação custo x benefício, visto que uma simples GPU pode conter o mesmo poder computacional de dezenas de CPUs. Esse é um dos principais atrativos que vem fazendo com que as GPUs sejam largamente utilizadas em aplicações paralelas.

Entretanto, não são todas as aplicações que podem tirar proveito dos recursos providos pelas GPUs. É necessário que existam trechos na aplicação onde o processamento dos dados possa ser efetuado de forma independente, logo estes cálculos podem ser paralelizados. Em aplicações que resolvem problemas da matemática e engenharia é típica a presença de várias partes paralelizáveis e outras poucas não-paralelizáveis ou sequenciais, isto é, que não podem ser calculadas de modo simultâneo.

Para ilustrar um simples caso onde o emprego da computação paralela é típica, imagine um programa que realiza o cálculo da soma de dois vetores A e B e gera como resultado um vetor C, todos de mesmo tamanho. Qualquer elemento c_i de C, será obtido pelo cálculo da soma dos elementos a_i e b_i dos respectivos vetores A e B:

$$c_i = a_i + b_i$$

Logo a computação de cada elemento c_i do vetor C pode ser realizada de maneira totalmente independente das demais, tornando-se possível o uso de técnicas de paralelização neste caso.

Atualmente, quando o assunto é computação em GPUs, uma das arquiteturas mais utilizadas para o desenvolvimento de códigos paralelos é NVIDIA CUDA.

4.1.1 NVIDIA CUDA

A arquitetura unificada de dispositivos de computação (CUDA - *Compute Unified Device Architecture*) [34] é uma plataforma de computação paralela e modelo de programação criada pela companhia de tecnologia NVIDIA, integrando *hardware* e *software*. CUDA oculta dos programadores o complexo *hardware* das GPUs, oferecendo uma série de ferramentas e bibliotecas para desenvolvimento de aplicações de alto desempenho em linguagens como por exemplo C. Através da chamada de funções e procedimentos, programadores conseguem obter acesso às funcionalidades que as GPGPUs oferecem, podendo desfrutar de seu massivo poder de processamento em paralelo.

Para executar uma aplicação em uma GPU, o usuário necessita de uma placa gráfica da NVIDIA que ofereça suporte ao CUDA, visto que esta tecnologia é de exclusividade da NVIDIA. Aliás, torna-se também necessária a instalação de um compilador específico para as GPUs da NVIDIA e bibliotecas do CUDA. A ferramenta CUDA *Toolkit* é um pacote que reúne o compilador, diversas bibliotecas matemáticas e ferramentas para depurar e otimizar o desempenho das aplicações, podendo ser obtido gratuitamente na área do desenvolvedor no *website* da NVIDIA[1]. Logo, basta que o usuário realize o *download* e a devida instalação de tal pacote para estar apto a criar e executar aplicações em CUDA.

4.1.2 Arquitetura de uma GPU NVIDIA

A seguir algumas características básicas do *hardware* das GPUs da NVIDIA são apresentados. Para fins de exemplificação, em determinados locais do texto, serão informadas características da placa gráfica Tesla^{TM} C1060 utilizada nas simulações deste trabalho. Esta GPU conta com um total de 240 núcleos de processamento para cálculos envolvendo precisão simples e 30 núcleos de processamento para cálculos de precisão dupla,

[1] `https://developer.nvidia.com/cuda-toolkit`

além de uma memória global de 4 GB GDDR3 e um espaço de memória de 65 KB para constantes (variáveis somente para leitura).

Embora existam vários modelos e tipos de GPUs da NVIDIA, a arquitetura encontrada em seu *hardware* segue praticamente o mesmo modelo, diferindo apenas em suas tecnologias e dados técnicos, como número de processadores, tamanho da memória, velocidades de cópia de dados, limitações impostas em seus recursos, etc.

A computação de uma aplicação em CUDA é realizada por uma função especial que é executada na GPU de um computador: o *kernel*. No momento em que é invocada na CPU (*host*), esta função de caráter paralelo passa então a ser executada na GPU (*device*) por várias unidades básicas de processamento (ou *threads*) que realizam a computação de maneira simultânea.

Na hierarquia proposta pela arquitetura CUDA, a placa gráfica é baseada em um vetor escalável de multiprocessadores chamados de *Streaming Multiprocessors* (SMs). Cada SM é composto por vários núcleos de processamento chamados *Stream Processors* (SPs). Estes são responsáveis pela execução das *threads*.

O dispositivo Tesla^{TM} C1060 possui um total de 30 multiprocessadores, cada um constituído por:

- SPs com *clock* de 1,30 GHz, sendo 8 para cálculos de precisão simples e 1 para cálculos de precisão dupla;
- Um espaço de memória compartilhada de 16 KB para uso dos SPs;
- Uma unidade de *cache* para dados classificados como texturas ou constantes.

As *threads* que irão executar o *kernel* são organizadas hierarquicamente em dois níveis compostos por blocos de *threads* e um *grid* de blocos. Os blocos podem ser de uma, duas ou três dimensões e estão dispostos em um *grid* uni- ou bidimensional.

Para que a CPU consiga chamar a função *kernel*, é necessário especificar a quantidade de *threads* que serão criadas em tempo de execução para processar os dados. Tal processo, formalmente conhecido como configuração de execução, influencia de forma direta o desempenho de uma aplicação. A configuração de execução é bastante flexível em relação à hierarquia do CUDA para *threads*, blocos de *threads* e *grid* de blocos, entretanto existem algumas limitações relacionadas ao número e às dimensões desses componentes.

As especificações da GPU Tesla^{TM} C1060 restringem o número de *threads* nas dimensões de x, y e z de um bloco com valores de 512, 512 e 64, respectivamente. Outra restrição imposta diz respeito ao valor máximo de 512 *threads* por bloco, independentemente das dimensões que este possua. A quantidade de blocos no *grid* na dimensão x multiplicado pela quantidade em y não deverá exceder o limite de 65.535 blocos estabelecido pelo *chip*.

Quando o *host* define as configurações de execução e invoca um *kernel*, cada bloco de *threads* é automaticamente escalonado em um SM disponível. O *hardware* do SM utiliza uma arquitetura proposta pelo CUDA chamada SIMT (*Single Instruction Multiple Threads*), responsável por criar, gerenciar, escalonar e executar as *threads* de um bloco em grupos de 32 *threads* paralelas, denominados *warps*. A unidade SIMT emite uma única instrução por vez para cada *warp*, fazendo com que todas as suas *threads* componentes comecem a executar juntas a partir de um mesmo local, porém, estas são livres para divergirem em seus caminhos de execução. Caso as *threads* de um *warp* divirjam em um caminho de execução, através de um desvio condicional por exemplo, o *warp* executa sequencialmente cada desvio tomado e as *threads* que não se encontram naquele caminho têm sua execução desabilitada. Um cenário ideal seria aquele onde todas as 32 *threads* de cada *warp* concordassem em um mesmo caminho de execução, obtendo uma eficiência total do *hardware*.

Já que todas as *threads* executam o mesmo trecho de código por consequência do modelo de computação SIMD, cada uma precisa definir um identificador único (ID). Os IDs serão utilizados para distinguir cada *thread* das demais como também para determinar a porção de dados que cada uma irá processar. Para tal, a arquitetura CUDA provê mecanismos úteis para obtenção de IDs para as *threads*: duas variáveis internas criadas em tempo de execução indicam de maneira única a posição de uma *thread* em um bloco (*threadIdx*) e a posição de um bloco de *threads* no *grid* (*blockIdx*). Essas duas variáveis internas podem ser acessadas dentro das funções *kernel*, retornando seus devidos valores a serem utilizados na obtenção de um ID.

No *chip* Tesla^{TM} C1060, cada *thread* possui o seu próprio conjunto de registradores, 16 KB de memória compartilhada no bloco, 65 KB de memória compartilhada no *grid* para as constantes e um total de 4 GB de memória global.

Devido às diferentes características das memórias encontradas em uma GPU, a

plataforma CUDA especificou uma hierarquia de modo a classificá-las basicamente em cinco tipos: a) registradores; b) memória local; c) memória compartilhada; d) constantes; e e) memória global. A Figura 4.2 relaciona os diferentes tipos de memória encontrados em uma placa gráfica da NVIDIA com suas respectivas velocidades de acesso a dados em um *grid* e blocos unidimensionais.

A qualquer momento na execução do *kernel*, todas as *threads* podem realizar operações de leitura e escrita em seus registradores, sua memória local, na memória compartilhada pelo bloco de *threads* e na memória global compartilhada pelo *grid* de blocos, porém, para a região de constantes somente é permitida a leitura dos dados. Para cada um desses distintos locais de armazenamento existe uma determinada velocidade de acesso.

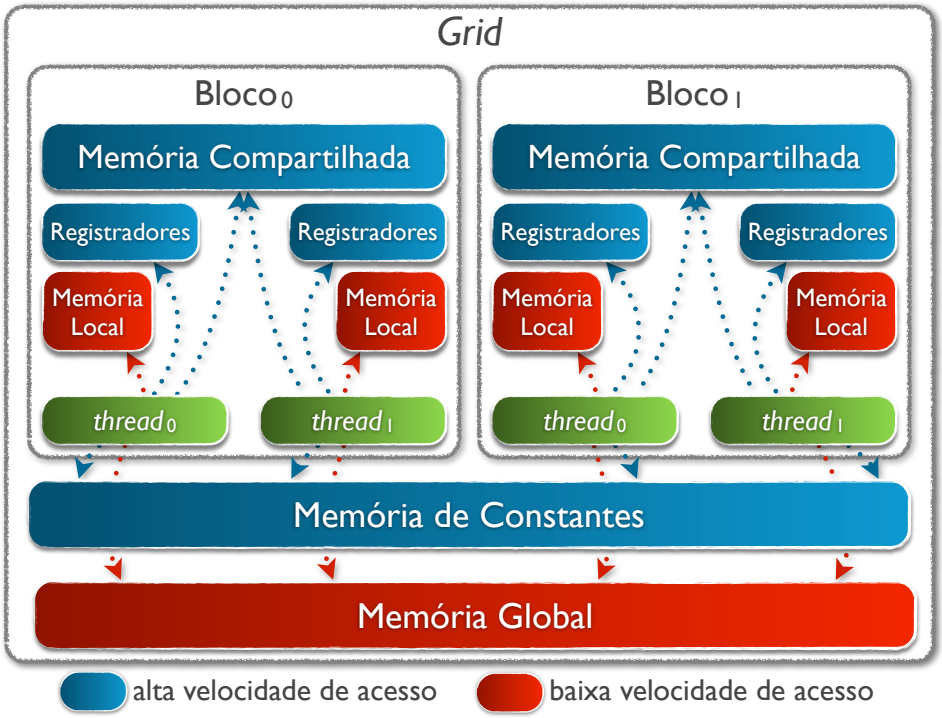


Figura 4.2: Hierarquia de memória em uma GPU

Ambos registradores e memória local são privativos de cada *thread*. Sua diferença está no tipo de variável e na velocidade de acesso: enquanto os registradores são atribuídos a variáveis escalares e possuem as maiores velocidades de acesso, a memória local é atribuída a variáveis do tipo vetor e possui uma velocidade de acesso lenta. Entretanto, em GPUs NVIDIA da classe Fermi ou em modelos mais atuais, os dados de uma variável do tipo vetor, uma vez acessados podem ser copiados na memória *cache* do SM que possui grande velocidade de acesso. Dados técnicos da Tesla^{TM} C1060 informam que o número total de registradores disponíveis por bloco de *threads* é de 16.384 unidades.

A memória compartilhada pode ser acessada por todas as *threads* contidas em um bloco. Com capacidade de 16 KB de armazenamento, este tipo de memória oferece uma velocidade de acesso equivalente aos registradores, sendo muito rápida. É geralmente utilizada para a comunicação entre *threads* de um bloco como também em tarefas de armazenamento intermediário de cálculos e leitura e escrita em *buffers* com o intuito de alcançar melhores padrões de acesso à memória.

A memória destinada ao armazenamento de constantes é do tipo somente leitura, e uma vez acessada tem seus dados copiados na memória *cache* do SM que é muito rápida. Logo, este tipo de memória possui uma velocidade de acesso semelhante a um registrador ou memória compartilhada.

Por fim, tem-se que a memória global é a detentora da maior capacidade de armazenamento na hierarquia. Com significativos 4 GB de espaço total, ela pode ser acessada por todas as *threads* do *grid* e consegue armazenar uma grande quantidade de dados, porém o acesso a esse tipo de memória é comparado à memória local, ou seja, de grande latência.

Uma variável declarada na memória da GPU pode residir em um dos locais especificados anteriormente, adquirindo determinadas permissões a leitura e gravação de dados, escopo de uso e tempo de vida. A arquitetura CUDA é flexível quanto a isso e oferece recursos para que em alguns casos um programador possa determinar a região de memória que sua variável irá residir. Para tal, CUDA criou os qualificadores de variáveis: *device*, *shared* e *constant*. A Tabela 4.1 exemplifica a declaração na linguagem C de uma variável em uma determinada região de memória da GPU e se esta permite ou não seu acesso por parte da CPU.

Tabela 4.1: Qualificadores de variáveis CUDA

Declaração	**Memória**	**Escopo**	**Tempo de Vida**	**CPU pode acessar?**
int var;	registrador	*thread*	*thread*	não
int var[10];	local	*thread*	*thread*	não
__**shared**__ **int** var;	compartilhada	bloco	bloco	não
__**constant**__ **int** var;	constante	*grid*	aplicação	sim
__**device**__ **int** var;	global	*grid*	aplicação	sim

Como consequência das características técnicas, como velocidade de acesso aos recursos de uma placa gráfica, pode-se deduzir que para alcançar um bom desempenho na execução da aplicação, é fundamental que seja realizado um mapeamento eficiente da computação nos diferentes tipos de memória oferecidos pela arquitetura. É sabido que a configuração de execução também causa grande impacto no desempenho da aplicação.

Algoritmo 1 Soma de dois vetores em CUDA

```
#include < stdio.h >

#define N 100 // número de elementos do vetor

// kernel executado na GPU
__global__ void soma_vetores(float *A, float *B, float *C)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < N) C[id] = A[id] + B[id];
}

// programa principal executado na CPU
int main(int argc, char *argv[ ])
{
    float A_h[N], B_h[N], C_h[N]; // declaração dos vetores A, B e C no host

    cudaSetDevice(0); // inicializar a GPU

    // declaração e alocação dos vetores A, B e C no device
    float *A_d, *B_d, *C_d; // ponteiro para os vetores no device
    size_t size = N * sizeof(float);
    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    // inicialização dos vetores A e B na CPU
    for (int i = 0; i < N; i++) {
        A_h[i] = (float) i;
        B_h[i] = (float) (i * i);
    }

    // cópia dos vetores A e B da CPU para a GPU
    cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

    // configuração de execução e chamada do kernel
    int block_size = 4;
    int n_blocks = N/block_size + (N%block_size == 0 ? 0 : 1);
    soma_vetores <<< n_blocks, block_size >>>(A_d, B_d, C_d);

    // cópia do vetor C calculado na GPU para a memória da CPU
    cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

    // imprime os resultados na tela
    for (int i = 0; i < N; i++) printf("\n C[%d] = %f", i, C_h[i]);

    // liberação da memória alocada na GPU
    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);

    return 0;
}
```

O Algoritmo 1 exemplifica um código CUDA na linguagem C que realiza a soma de dois vetores A e B e armazena o resultado no vetor C, como no problema caracterizado como programação paralela dado no início desta seção.

Para executar uma aplicação na GPU o programador deverá criar uma função *kernel* e seguir alguns passos, que podem ser visualizados no Algoritmo 1:

1. Inicializar a GPU (linha 17);
2. Alocar a memória na GPU (linhas 22, 23 e 24);
3. Copiar os dados da CPU para a GPU (linhas 33 e 34);
4. Realizar a configuração de execução (linhas 37 e 38);
5. Invocar o *kernel* (linha 39);
6. Após a execução do *kernel*, copiar os resultados de volta para a CPU (linha 42);
7. Liberar a memória alocada na GPU (linhas 48, 49 e 50).

As cópias de dados entre CPU e GPU podem ser realizadas por meio da chamada da função *cudaMemcpy*, que inicia uma transferência DMA (*Direct Memory Access*) da memória do *host* para o *device* através do barramento PCI *Express*, sendo que o sentido da cópia é especificado por um dos parâmetros da função, assim como tamanho da cópia e localização das memórias.

Contudo, devido ao *overhead* causado pelos altos custos de transferências de dados entre *host* e *device*, uma aplicação pode ter um desempenho altamente prejudicado. Uma tentativa de contornar esse problema, é o uso de *streams* CUDA, que oferecem alternativas de se realizar sobreposição de tarefas (*overlap*).

4.1.3 Stream CUDA

Na arquitetura CUDA, uma *stream* pode ser definida como uma sequência de comandos que irão executar na GPU na ordem em que foram chamadas [35]. Todas as operações como chamada do *kernel* e transferências de dados entre CPU e GPU são executadas em uma *stream*. Enquanto as operações de uma determinada *stream* são executadas obrigatoriamente na ordem em que foram enviadas, comandos que estejam em diferentes

streams podem ser facilmente intercalados e até mesmo executados concorrentemente. Isso permite que tarefas, como cópia de dados, possam ser sobrepostas com computação e o *hardware* possa ser utilizado ao máximo e fique menos tempo em estado de espera (ou *idle*), fatores esses que impactam negativamente no desempenho da aplicação.

Existe uma *stream* padrão que é automaticamente associada a chamadas de *kernels* e cópias de dados entre *host* e *device*, quando nenhuma outra é especificada. Também chamada de *stream* nula ou *stream* '0', ela é diferente das outras por ser uma sincronizadora genérica das operações em um *device*: uma operação vinculada a ela deverá terminar para que outra operação em qualquer outra *stream* possa começar, e nenhuma operação nessa *stream* será iniciada até que todas as operações em todas as outras *streams* tenham terminado. A arquitetura oferece uma função para que um programador possa criar suas próprias *streams*, a *cudaStreamCreate*.

Do ponto de vista do *host*, quando um *kernel* associado a uma determinada *stream* é chamado, a GPU começa a executá-lo e imediatamente retorna o controle à *thread* do *host*, de modo que este possa realizar outra computação enquanto o *kernel* está sendo processado pelo *device*. Isso ocorre porque uma chamada de *kernel* é, por padrão, de comportamento assíncrono ou não bloqueante, ou seja, a CPU não precisa ficar bloqueada até que a GPU termine sua computação, podendo então realizar outras atividades. Visto que todas as operações em uma *stream* que não seja a padrão são de caráter não bloqueante com relação ao *host*, existem diversas situações onde será necessária a sincronização do *host* com o término das operações na *stream*. A função *cudaStreamSynchronize* pode ser utilizada para tal propósito, sendo passado como argumento a variável que representa a *stream* a ser sincronizada com a CPU, fazendo com que esta última fique em estado *idle* até o término das operações associadas à *stream*.

Uma grande aplicação de *streams* é na realização de *overlaps*, sobrepondo com computação a comunicação e transferência de dados entre processos. O presente trabalho tem como característica utilizar técnicas para lidar com tal problema, visto que para a correta execução do modelo aqui simulado, os processos que são participantes da computação devem trocar pequena parte de seus dados computados a todo momento. Essa troca de dados só poderá ser realizada após o término da execução do *kernel* e a posterior cópia de dados entre GPU e CPU. Como somente uma pequena porção dos dados calculados precisa ser enviada, seu cálculo pode ser decomposto em duas partes: a

primeira processa aqueles dados utilizados na troca entre os processos e a segunda realiza o cálculo dos dados restantes.

O uso de *streams* permite que tal abordagem seja implementada, associando o *kernel* de cada parte e sua respectiva cópia de dados a uma *stream* diferente. Logo, após o término da primeira *stream*, os respectivos dados podem começar a ser copiados para o *host* com a função *cudaMemcpyAsync* e, na sequência, este pode realizar a troca dos resultados com os outros processos. Enquanto isso, a segunda parte pode estar sendo calculada na GPU por uma outra *stream* de forma simultânea, havendo uma sobreposição da comunicação com computação.

4.1.4 UVA

Endereçamento virtual unificado, ou *Unified Virtual Addressing* (UVA) [36], é um recurso disponível em GPUs NVIDIA da classe Fermi (*compute capability* 2.x) ou em modelos mais atuais, como por exemplo Kepler [35]. A arquitetura de uma placa está associada a

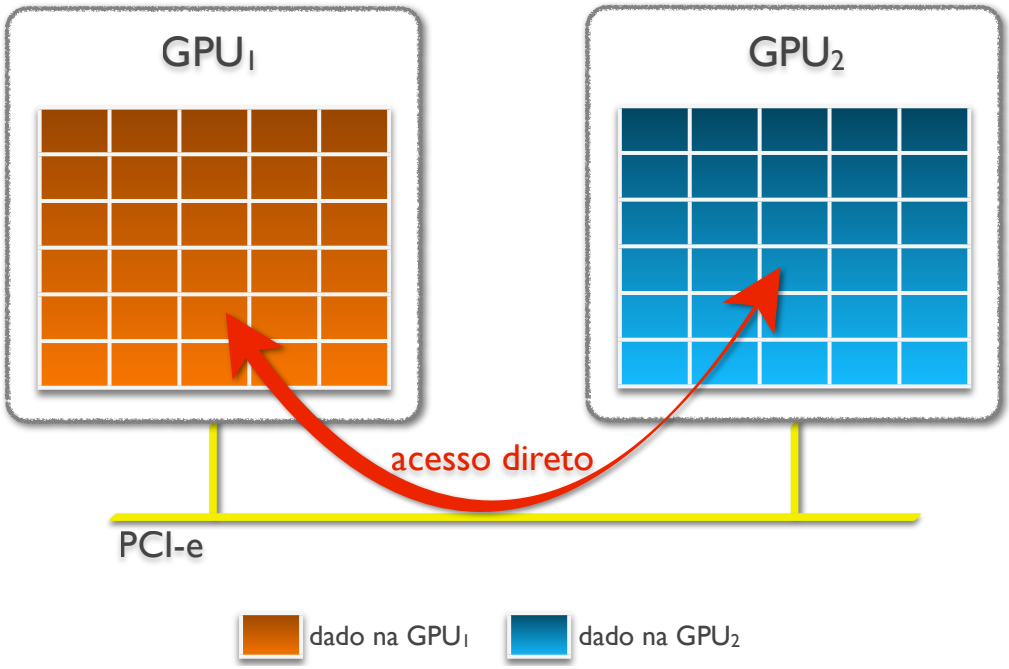


Figura 4.3: UVA - Acesso direto aos dados de uma GPU

um *compute capability*, que nada mais é do que uma classificação da GPU, representada por um valor, quanto ao seu suporte a determinados recursos e funções específicas.

Através do UVA, a memória do *host* e de todos os *devices* instalados é unificada em um único espaço virtual de endereçamento, facilitando seu acesso. Esse inovador

recurso é ativado por meio da função *cudaDeviceEnablePeerAccess* e estabelece uma comunicação ponto-a-ponto (P2P - *Peer-to-peer*) entre duas GPUs, permitindo que a leitura e alteração de dados em ambas seja efetuado de maneira direta através do barramento PCI-e (Figura 4.3). Logo, a necessidade de transferências de dados entre GPUs pode ser reduzida ou até mesmo anulada ao se utilizar tal recurso. Além do acesso direto, há também a possibilidade de realizar a cópia direta de dados de uma GPU para outra, sem a necessidade de passar pela memória do *host*.

Os recursos providos pelo UVA permitem uma comunicação direta entre GPUs, fornecendo mecanismos de acesso e cópia direta de dados, tornando mais eficiente e intuitivo a computação com múltiplas GPUs.

4.2 OpenMP

OpenMP (*Open Multi Processing*) é uma especificação que traz recursos e funcionalidades para programação paralela em multi-plataformas com o uso de memória compartilhada nas linguagens C/C++ e Fortran. Através da disponibilização de bibliotecas, diretivas e funções que permitem a criação e o gerenciamento de *threads* em uma CPU, OpenMP define um modelo portável e escalável que permite o desenvolvimento de aplicações paralelas de uma forma simples e flexível [37].

Com o uso de OpenMP é possível criar programas paralelos de memória compartilhada de uma maneira fácil com o uso de *threads*. Programas em OpenMP basicamente começam com um processo sequencial na CPU, chamado *thread* mestre. A *thread* mestre executa o código sequencialmente até encontrar uma região paralela. A partir daí, é criado um time de *threads* que processam simultaneamente o código delimitado pela região paralela. Após sua execução, todas as *threads* se sincronizam e terminam, restando ativa somente a *thread* mestre que originou tal conjunto (modelo *fork-join*).

Para especificar um bloco de código que será executado em paralelo pelas *threads*, é necessário que o programador utilize a diretiva *#pragma omp* antes do bloco de código. A sintaxe para desenvolver aplicações paralelas em OpenMP na linguagem de programação C/C++ é a seguinte: a cláusula *#pragma omp* é de uso obrigatório, seguida por uma diretiva válida e cláusulas opcionais. Por fim, é necessário a inclusão de uma nova linha.

O Algoritmo 2 exemplifica um código na linguagem C no qual a região paralela é

executada concorrentemente por 5 *threads*. Seu funcionamento é bem simples: um vetor de 5 inteiros é alocado, e na região paralela, cada *thread* irá atribuir o seu identificador à sua respectiva posição no vetor, de modo que a primeira *thread* acesse a primeira posição e coloque o valor 1, a segunda *thread* acesse a segunda posição e coloque o valor 2, e assim por diante. Ao final do programa, os valores armazenados no vetor são exibidos na tela.

Algoritmo 2 Exemplo de programa em OpenMP

```
#include < stdio.h >
#include < omp.h >

#define N 5 // número de elementos do vetor

int main(int argc, char *argv[ ])
{
    // código sequencial executado pela thread mestre
    int var[N] = {0};

    #pragma omp parallel num_threads(N)
    {
        // região paralela executada simultaneamente pelas N threads
        int id = omp_get_thread_num();
        var[id] = id + 1;
    }

    // código sequencial executado pela thread mestre
    for (int i = 0; i < N; i++) printf("\n var[%d] = %d", i, var[i]);

    return 0;
}
```

Caso exista necessidade de sincronização de todas as *threads* em uma parte específica do programa, a diretiva *#pragma omp barrier* pode ser utilizada. As *threads* deverão esperar até que todas elas alcancem a barreira para somente assim poderem continuar com sua execução.

Uma situação que caracteriza o uso das *threads* OpenMP é na gestão de recursos de uma máquina, como por exemplo múltiplas GPUs. Nesta abordagem, cada *thread* do *host* é responsável pelo gerenciamento de uma única GPU, realizando funções como alocação de memória, chamada do *kernel* e transferência de dados nessa GPU.

Típico de aplicações paralelas, o uso de identificadores é essencial para uma correta definição das tarefas a serem desempenhadas por cada *thread*, bem como os dados a serem processados por elas. A função *omp_get_thread_num* pode ser utilizada para a obtenção do identificador único da *thread*, que serve para distinguí-la das demais.

Na aplicação desenvolvida neste trabalho, a API OpenMP foi utilizada com a função de prover o correto gerenciamento das GPUs de uma máquina, de modo que cada *thread* do *host* seja responsável por invocar um *kernel* CUDA em um específico *device*. O uso

de múltiplas *threads* na CPU é necessário para reduzir o desequilíbrio causado quando uma simples *thread* invoca os *kernels* em todas as GPUs presentes. Se um número maior de GPUs estiver disponível, quando a única *thread* do *host* tiver terminado de iniciar o *kernel* na última GPU, provavelmente o primeiro *kernel* da primeira GPU avançou muito em seu processamento, ou até mesmo o terminou.

4.3 MPI (*Message Passing Interface*)

O MPI é um padrão muito conhecido para troca de mensagens. Seu objetivo é estabelecer um padrão flexível, eficiente e portável para o desenvolvimento de aplicações paralelas de alto desempenho que utilizem comunicação entre processos [38]. Existem várias implementações do padrão MPI disponíveis no mercado. Uma das mais utilizadas é o MPICH [39] por se tratar de um *software* livre e de fácil uso que oferece suporte a linguagens de programação como C/C++ e Fortran. Com o uso da implementação MPICH é possível criar aplicações paralelas de memória distribuída, já que os processos envolvidos na computação não compartilham suas memórias em um mesmo espaço (CPU e/ou GPU).

No momento da execução de um programa MPI, o programador deverá especificar o número de processos que participarão da computação. Após criados, cada processo recebe um ID que pode ser utilizado para distingui-lo dentre os demais. Todos estes processos são então organizados em um grupo e utilizam um mesmo comunicador para realizarem suas trocas de mensagens. O ID de cada processo dentro do grupo pode ser obtido pelo uso da primitiva *MPI_Comm_rank*.

Para que seja realizada a comunicação entre dois processos de uma aplicação MPI, é necessário utilizar algumas funções implementadas pela biblioteca MPICH. Enquanto algumas dessas funções são de caráter ponto-a-ponto, ou seja, envolvem apenas dois processos, outras são de caráter coletivo, ou global, envolvendo todo o grupo de processos.

As primitivas *MPI_Send* e *MPI_Recv*, ambas de natureza bloqueante, são consideradas ponto-a-ponto, já que apenas dois processos estarão envolvidos na comunicação: o processo A envia dados ao processo B por meio da chamada de *MPI_Send*, e B recebe os dados de A invocando a função *MPI_Recv*. Uma alternativa à essas funções, na qual o *host* pode prosseguir com sua execução sem que a mensagem tenha sido devidamente enviada

ou recebida, são as primitivas não-bloqueantes *MPI_Isend* e *MPI_Irecv*, que realizam de maneira assíncrona o envio e recebimento de dados, respectivamente. Parâmetros como local da memória, remetente ou destinatário da comunicação, tipo e quantidade de dados a serem enviados ou recebidos são utilizados para especificar uma comunicação P2P a ser realizada.

As funções coletivas são aquelas que envolvem um grupo de dois ou mais processos. Um exemplo é a primitiva *MPI_Bcast*, que envia uma mensagem de um processo A a todos os outros processos do grupo de A.

Caso exista a necessidade de sincronização de todos os processos de um grupo em um local específico do código, a função *MPI_Barrier* pode ser utilizada. Os processos deverão esperar até que todos alcancem a barreira para somente assim prosseguirem com sua execução.

5 IMPLEMENTAÇÃO

O presente capítulo versa sobre a implementação do modelo matemático para o SIH (Sistema Imunológico Humano) inato apresentado no Capítulo 3, bem como a paralelização deste em um ambiente de memória distribuída utilizando CUDA (*Compute Unified Device Architecture*), OpenMP (*Open Multi Processing*) e MPI (*Message Passing Interface*).

Este capítulo está organizado da seguinte forma. Inicialmente será descrito como o modelo matemático previamente definido foi implementado, abordando os métodos numéricos que foram empregados nas resoluções de suas equações. Finalizando o capítulo, a Seção 5.2 discorre sobre as três versões implementadas neste trabalho: a primeira realiza transferências de dados entre distintas GPUs (*Graphics Processing Unit*) usando a memória principal como espaço intermediário de armazenamento; já a segunda versão utiliza o recurso UVA (*Unified Virtual Addressing*) para que as GPUs acessem seus dados diretamente e, por fim, a versão mais elaborada tenta sobrepor as transferências de dados com computação, empregando *streams* CUDA para este propósito. As três versões criadas serão descritas nas subseções 5.2.1, 5.2.2 e 5.2.3, respectivamente.

5.1 Modelo Computacional

O método numérico empregado para implementar o modelo matemático especificado na seção anterior foi o Método das Diferenças Finitas [40]. Este método é comumente utilizado na discretização numérica de EDPs (Equações Diferenciais Parciais), baseando-se na idéia de que as derivadas podem ser calculadas por meio de diferenças finitas.

Com o intuito de simular o fenômeno da difusão de alguns componentes do SIH no tecido em três dimensões [3], o operador Laplaciano foi discretizado utilizando diferenças

finitas:

$$
\begin{aligned}
D_O \left(\frac{\partial^2 O(x,y,z)}{\partial x^2} + \frac{\partial^2 O(x,y,z)}{\partial y^2} + \frac{\partial^2 O(x,y,z)}{\partial z^2} \right) \approx \\
D_O * [(O[x+1,y,z] - 2 * O[x,y,z] + O[x-1,y,z])/deltaX^2)] + \\
+ D_O * [(O[x,y+1,z] - 2 * O[x,y,z] + O[x,y-1,z])/deltaY^2)] + \\
+ D_O * [(O[x,y,z+1] - 2 * O[x,y,z] + O[x,y,z-1])/deltaZ^2)]
\end{aligned}
\tag{5.1}
$$

Na Equação 5.1 definida anteriormente, O representa a discretização de algumas células do SIH, tais como neutrófilos, macrófagos ativos e macrófagos em repouso. O coeficiente de difusão das populações das células é dado por D_O. Para representar a posição de uma célula no espaço tridimensional, foram utilizadas as coordenadas x, y e z com as respectivas discretizações espaciais *deltaX*, *deltaY* e *deltaZ*.

Outro fator importante é o cálculo do fluxo da quimiotaxia em três dimensões nos pontos do tecido. Seu cálculo é feito com base nos diferentes fluxos recebidos nas direções dos eixos x, y e z por um ponto de coordenadas (x, y, z), e o seu valor total é o somatório desses fluxos. A obtenção do fluxo quimiotático que atua no ponto (x, y, z) por meio de um determinado eixo é a soma dos fluxos da direita e esquerda que este ponto (x, y, z) recebe neste mesmo eixo por influência de seus pontos vizinhos. Para fins de exemplificação, um pseudocódigo simplificado que realiza esse tipo de cálculo é mostrado a seguir (Algoritmo 3).

Porém, em sua implementação original, realizada neste trabalho, para que o valor do fluxo da quimiotaxia seja efetuado corretamente, deve-se levar em consideração a localização espacial dos componentes de modo a tratar os casos de borda do domínio. Em tais locais não existirão ambos os fluxos da direita e esquerda no cálculo do fluxo correspondente ao eixo em que tais pontos sejam caracterizados como pontos de borda.

No Algoritmo 3 apresentado, $CH_{x,y,z}$ representa a discretização das citocinas pró-inflamatórias e o termo denotado por $O_{x,y,z}$ é a representação da discretização de algumas células do SIH, ambos com suas coordenadas no espaço definidas pelos valores de (x, y, z). Como já visto, as discretizações espaciais são dadas por *deltaX*, *deltaY* e *deltaZ*.

Logo, com as expressões utilizadas acima para o cálculo da quimiotaxia, o resultado de sua avaliação é dado pela Equação 5.2:

$$
-\nabla.(\chi_O O \nabla CH) \approx -\chi_O * fluxo\_Total, \tag{5.2}
$$

Algoritmo 3 Cálculo da quimiotaxia

```
// Fluxo no eixo-x
se CH_{x,y,z} − CH_{x−1,y,z} > 0 então
        fluxoEsquerda ← −(CH_{x,y,z} − CH_{x−1,y,z}) ∗ O_{x−1,y,z}/deltaX
senão
        fluxoEsquerda ← −(CH_{x,y,z} − CH_{x−1,y,z}) ∗ O_{x,y,z}/deltaX
fim-se
se CH_{x+1,y,z} − CH_{x,y,z} > 0 então
        fluxoDireita ← −(CH_{x+1,y,z} − CH_{x,y,z}) ∗ O_{x,y,z}/deltaX
senão
        fluxoDireita ← −(CH_{x+1,y,z} − CH_{x,y,z}) ∗ O_{x+1,y,z}/deltaX
fim-se
fluxo_X ← (fluxoEsquerda + fluxoDireita)/deltaX

// Fluxo no eixo-y
se CH_{x,y,z} − CH_{x,y−1,z} > 0 então
        fluxoEsquerda ← −(CH_{x,y,z} − CH_{x,y−1,z}) ∗ O_{x,y−1,z}/deltaY
senão
        fluxoEsquerda ← −(CH_{x,y,z} − CH_{x,y−1,z}) ∗ O_{x,y,z}/deltaY
fim-se
se CH_{x,y+1,z} − CH_{x,y,z} > 0 então
        fluxoDireita ← −(CH_{x,y+1,z} − CH_{x,y,z}) ∗ O_{x,y,z}/deltaY
senão
        fluxoDireita ← −(CH_{x,y+1,z} − CH_{x,y,z}) ∗ O_{x,y+1,z}/deltaY
fim-se
fluxo_Y ← (fluxoEsquerda + fluxoDireita)/deltaY

// Fluxo no eixo-z
se CH_{x,y,z} − CH_{x,y,z−1} > 0 então
        fluxoEsquerda ← −(CH_{x,y,z} − CH_{x,y,z−1}) ∗ O_{x,y,z−1}/deltaZ
senão
        fluxoEsquerda ← −(CH_{x,y,z} − CH_{x,y,z−1}) ∗ O_{x,y,z}/deltaZ
fim-se
se CH_{x,y,z+1} − CH_{x,y,z} > 0 então
        fluxoDireita ← −(CH_{x,y,z+1} − CH_{x,y,z}) ∗ O_{x,y,z}/deltaZ
senão
        fluxoDireita ← −(CH_{x,y,z+1} − CH_{x,y,z}) ∗ O_{x,y,z+1}/deltaZ
fim-se
fluxo_Z ← (fluxoEsquerda + fluxoDireita)/deltaZ

// Fluxo total resultante
fluxo_Total ← (fluxo_X + fluxo_Y + fluxo_Z)
```

onde χ_O é a taxa de quimiotaxia da população de cada célula ou molécula representada por O, ∇CH é o termo que representa a velocidade de movimento da população O e $fluxo\_Total$ é a soma dos fluxos nos eixos x, y e z, denotados por $fluxo\_X$, $fluxo\_Y$ e $fluxo\_Z$, respectivamente.

Para utilizar o esquema de diferenças finitas no cálculo do fluxo da quimiotaxia, a localização dos pontos no sistema é de fundamental importância para a escolha de qual método utilizar. Logo, diferenças finitas centrais foram aplicadas nos pontos internos do domínio e diferenças finitas para frente ou para trás foram utilizadas nas bordas do domínio.

A Tabela 5.1 apresenta as discretizações temporal e espacial utilizadas em todas as simulações realizadas.

Tabela 5.1: Discretizações Temporal e Espacial

Parâmetro	**Valor**	**Unidade**	**Discretização**
Tempo	1	dia (representado por 10^6 iterações)	$\delta T = 0.000001$
Eixo-X	1	mm (representado por 10 pontos)	$\delta X = 0.1$
Eixo-Y	1	mm (representado por 10 pontos)	$\delta Y = 0.1$
Eixo-Z	1	mm (representado por 10 pontos)	$\delta Z = 0.1$

Para a discretização da derivada temporal utilizou-se um método explícito, que permite calcular o estado de um sistema em um tempo posterior ao estado atual baseando-se apenas nos dados mais atuais. O método de Euler explícito foi escolhido para este propósito, logo, para calcular uma determinada população no instante de tempo *t+1*, são utilizadas informações apenas do tempo anterior *t*.

Todos os parâmetros das equações que modelam a dinâmica dos elementos do SIH, como também as condições iniciais para as populações de células utilizadas nas simulações, podem ser encontrados no apêndice deste trabalho, nas Tabelas A.1 e A.2, respectivamente. É importante ressaltar que todos os valores dos parâmetros foram retirados da literatura [2, 3, 31, 32].

5.2 Implementação em Múltiplas GPUs

A primeira versão do código, desenvolvida em [32], utilizava uma única GPU para reduzir os tempos gastos na simulação do comportamento espacial e temporal de algumas células e moléculas do SIH inato em uma seção tridimensional de tecido. Tal versão do código que utiliza somente uma GPU foi estendida neste trabalho com o objetivo de acelerar ainda mais a aplicação, utilizando para isso múltiplas GPUs localizadas em diferentes máquinas interligadas por uma rede. Todo o código foi desenvolvido na linguagem C, empregando CUDA, OpenMP e MPI.

O espaço discretizado é dividido entre todas as GPUs, de maneira que cada uma irá operar em uma específica fatia do espaço original. A fração do tecido é dada pelo cálculo da divisão da dimensão x de uma malha (X, Y, Z) que descreve o tecido pelo número total de GPUs que participam da computação, N, restando uma malha $((X+N-1)/N, Y, Z)$ a ser calculada em cada GPU (ou simplesmente *device*). Através do esquema de divisão

adotado, o grupo de GPUs consegue processar todo o tecido.

Para se obter o número total de GPUs, N, que participam da computação, foi utilizada a função *cudaGetDeviceCount*, que retorna o número de *devices* contidos em uma máquina, em conjunto com a primitiva *MPI_Bcast*, que envia um dado de um processo a todos os demais do mesmo grupo. Cada processo MPI irá enviar o valor correspondente à sua quantidade de GPUs disponíveis a todos os demais processos do grupo. Esta operação é necessária porque cada GPU precisa definir a sua posição relativa em relação às demais GPUs de modo a definir qual fatia do tecido deverá processar. Por exemplo, num total de 5 GPUs que processam um tecido (50, 20, 20), se uma GPU descobre que é a terceira na ordem de sucessão, será de sua responsabilidade o cálculo de todos os pontos da malha definidos pelo intervalo fechado de (20, 0, 0) a (29, 19, 19) (Figura 5.1). Tal estratégia foi utilizada com o intuito de promover um gerenciamento de carga de trabalho dinâmico e igualitário entre todas as CPUs (*Central Processing Unit*), visto que podem existir variações no número de GPUs presentes em cada máquina.

A Figura 5.1 ilustra a estratégia de divisão dinâmica para uma malha (50, 20, 20) entre 5 GPUs, de modo que cada uma será responsável por processar uma determinada fatia da malha.

O gerenciamento das GPUs de uma máquina é realizado pela API OpenMP, de modo que cada *thread* da CPU seja responsável por invocar um *kernel* CUDA em uma GPU específica da máquina. Como já mencionado no Capítulo 4, o uso de múltiplas *threads* se faz necessário para reduzir o desequilíbrio causado por uma única *thread* que lança vários *kernels* em GPUs diferentes, já que ao lançar o último *kernel* na última GPU, o primeiro *kernel* da primeira GPU já poderia ter avançado em sua computação ou até mesmo a finalizado. Visto que os *kernels* são executados de maneira independente em cada GPU da máquina, é necessário que a cada passo de tempo da simulação as *threads* OpenMP no *host* sejam sincronizadas para o correto cálculo das populações do modelo. Ao utilizar a diretiva *#pragma omp barrier* ao final de cada iteração do tempo, as *threads* que gerenciam as GPUs da máquina esperam em uma barreira até que todas as suas respectivas GPUs terminem sua computação, para somente assim continuarem na próxima iteração. Além da sincronização das GPUs de uma máquina, também é necessário sincronizar as GPUs de outras máquinas que também participam da computação, porém tal assunto será tratado mais adiante.

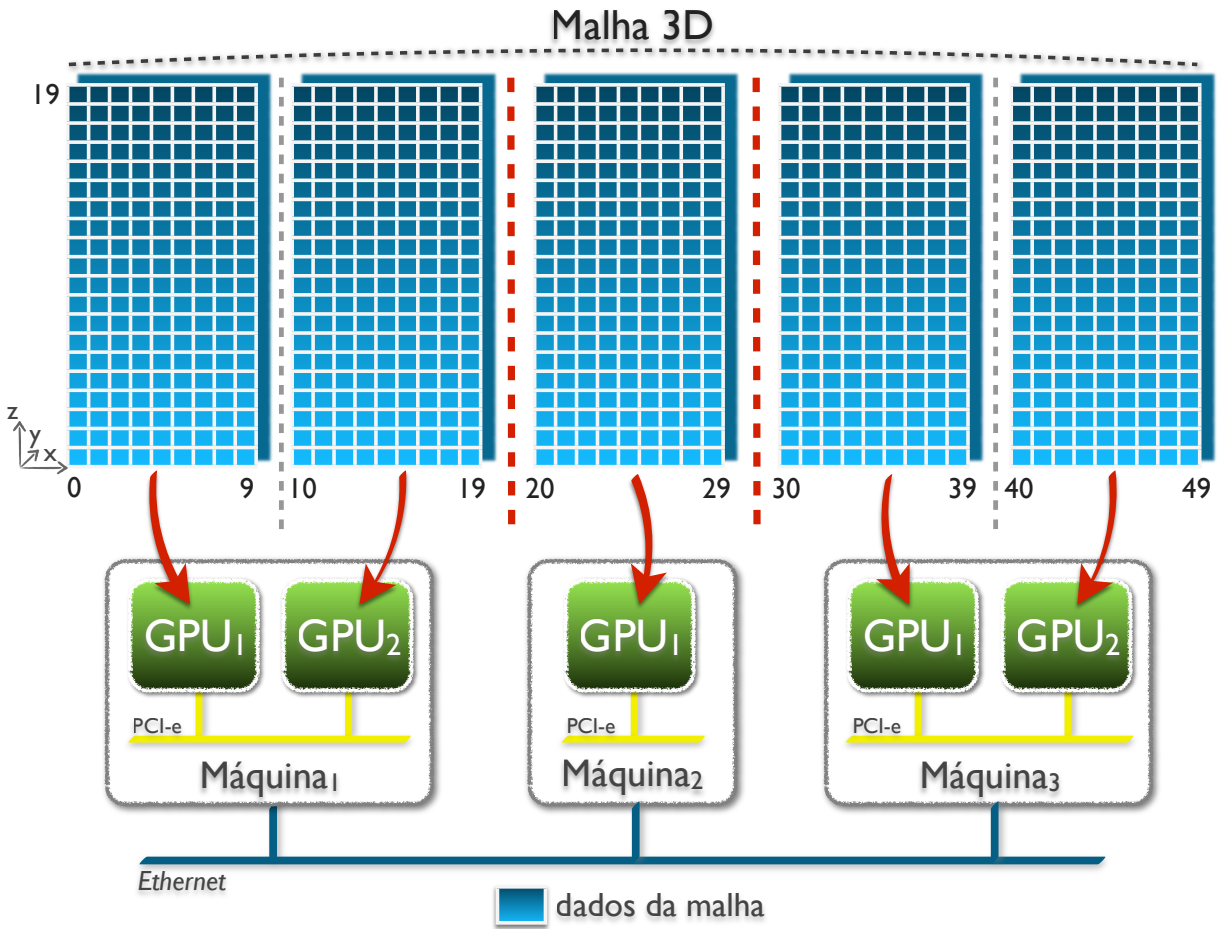


Figura 5.1: Exemplo de divisão de uma malha 3D entre 5 GPUs

Na arquitetura CUDA, sabe-se que a configuração de execução exerce grande influência no desempenho da aplicação. Baseado nos recursos disponíveis do *hardware* da GPU, como também na demanda de memória por cada *thread* CUDA, o número de *threads* por bloco foi fixado neste trabalho em 256. O cálculo da dimensão do *grid* de blocos é realizado de forma automática, levando em conta as dimensões da fatia da malha e número de *threads* por bloco.

Outro fator que impacta o desempenho de uma aplicação é a precisão numérica empregada nas computações: precisão simples ou precisão dupla. Como já observado no Capítulo 4, uma GPU possui oito vezes mais núcleos de processamento para lidar com cálculos de precisão simples que unidades de processamento para cálculos envolvendo precisão dupla. Logo, com o intuito de utilizar o maior número de núcleos possível, fato que contribui positivamente no desempenho da aplicação, foi utilizada a precisão simples em todos os cálculos do modelo. Contudo, tal escolha não resultou em prejuízos significativos que pusessem à prova a confiabilidade dos resultados numéricos.

Similar à versão do código para uma GPU [32], cada *thread* CUDA que executa o *kernel* representa um único ponto (x, y, z) no espaço discretizado do tecido tridimensional, sendo de sua responsabilidade o cálculo completo do conjunto de EDPs que representam as populações de células e moléculas simuladas no modelo, como neutrófilos, macrófagos, citocina pró-inflamatória, etc.

Durante o cálculo das EDPs, para que uma *thread* CUDA possa computar corretamente um ponto do tecido ela deve possuir acesso a dados produzidos pelas *threads* vizinhas. Com o auxílio de um *buffer*, uma *thread* que está realizando seus cálculos no tempo t consegue obter o acesso aos dados produzidos por seus vizinhos no tempo *t-1*. Assim, evita-se uma condição de corrida (ou *race condition*) entre as *threads*, visto que os dados novos produzidos por uma *thread* no tempo t somente são acessados por ela, pois seus vizinhos acessam os dados no tempo *t-1*. Essas duas entradas do *buffer* mudam seu papéis a cada passo de tempo para que uma cópia de dados seja evitada.

Devido ao esquema adotado de divisão da malha entre as GPUs, todos os dados vizinhos relacionados às dimensões y e z residem na própria GPU, restando apenas os dados relacionados à dimensão x a serem devidamente tratados. Seja uma *thread* que representa um ponto no espaço 3D do tecido definida pelas coordenadas (α, β, γ). Em relação à dimensão x, um vizinho dessa *thread* pode ser de coordenadas ($\alpha - 1$, β, γ) ou ($\alpha + 1$, β, γ) e seu dado pode estar localizado em uma dentre 3 regiões distintas: a) na mesma GPU em que a *thread* realiza o cálculo; b) em uma GPU vizinha que resida na mesma CPU; ou c) em uma GPU que esteja em uma outra máquina. Nos dois últimos casos, os dados são chamados de bordas, pois se encontram nos extremos do eixo x de uma fatia de tecido processada por uma GPU. A seguir, a Figura 5.2 ilustra os três tipos de localização definidos anteriormente para um dado de um vizinho no eixo x.

Por meio de um identificador global, cada GPU consegue identificar o local onde deverá buscar o dado vizinho, ou na sua própria memória ou em outra GPU. Caso seja classificado como o primeiro caso - especificado em a) - o vizinho compartilha com a *thread* o mesmo espaço de memória, logo a *thread* poderá obter o dado requisitado com um simples acesso à memória global de sua GPU. Para as situações definidas em b) ou c), foram utilizados vetores auxiliares que irão armazenar os dados de borda da malha de seus vizinhos que estejam localizados em outras GPUs.

Já que a cada passo de tempo os dados de borda são necessários em qualquer GPU para

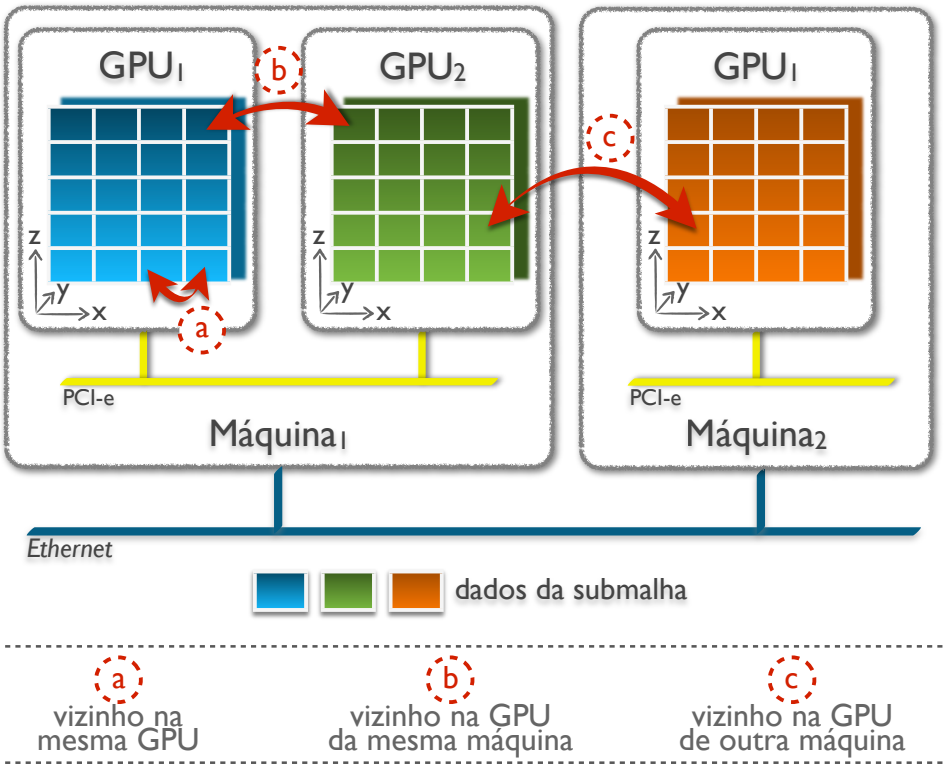


Figura 5.2: Possíveis localizações para um dado de um vizinho no eixo x

a correta computação dos pontos, tanto o envio quanto o recebimento de tais informações será realizado ao final de cada passo de tempo logo após o término da execução do *kernel*. Logo, o procedimento *enviarEReceberBordas* foi desenvolvido justamente para lidar com as trocas de bordas entre duas GPUs. Os dados de borda a serem enviados são aqueles calculados mais recentemente pela GPU e os dados recebidos serão utilizados pela GPU no cálculo das populações no próximo passo de tempo, portanto, a cada iteração uma GPU fará a troca de bordas ao chamar o procedimento *enviarEReceberBordas*.

A malha está organizada em um espaço contíguo na memória da GPU na forma de um vetor unidimensional e o acesso aos pontos no tecido é feito de forma linear. Além disso, o mapeamento no vetor para as populações de células e moléculas do modelo foi realizado com o objetivo de facilitar a transferência de dados entre as GPUs de uma mesma máquina e entre aquelas que se encontram em máquinas distintas e se comunicam por meio de primitivas MPI. Neste trabalho, a disposição dos dados do modelo na memória foi projetado de tal forma que as cópias de dados entre as GPUs seja realizada de maneira mais eficiente: para cada posição x no tecido, todas as populações do modelo no plano YZ_x são armazenadas de forma contígua na memória. Por exemplo, para o plano YZ_0 da malha com $x = 0$, todas as populações são armazenadas no início do vetor, seguidas logo após

pelo plano YZ_1 com $x = 1$ e assim por diante. Logo, somente uma sequência de valores contíguos será necessária na cópia das populações de um dado plano YZ_x do tecido entre duas GPUs. Do ponto de vista do desempenho, uma transferência de grande quantidade de dados incluindo todas as populações é muito melhor que muitas transferências pequenas para cada população.

Para lidar com os problemas deparados até o momento, como situações onde os dados de borda se encontram em endereços de memória de outras GPUs, e a necessidade de sincronização de todas as GPUs, de modo que as populações do modelo sejam calculadas corretamente a cada passo de tempo, três diferentes abordagens foram implementadas, e serão explicadas a seguir.

5.2.1 Versão 1

Nesta primeira versão da implementação, o conceito abordado é bem simples: os dados da borda são trocados entre duas GPUs que estejam ou não na mesma máquina, utilizando para este propósito a memória da CPU. Essa troca de dados, como já foi mencionada, só poderá ser realizada após o término da execução do *kernel* e a posterior cópia de dados entre GPU e CPU. O funcionamento do procedimento *enviarEReceberBordas*, responsável por todo o processo de envio e recebimento de uma borda entre duas GPUs passadas por parâmetro, é apresentado a seguir e baseia-se na localização dessas duas GPUs.

Quando presentes na mesma máquina, as GPUs podem trocar suas bordas em um processo de 3 etapas. Na primeira, através da função *cudaMemcpy*, os dados de uma GPU são copiados para um vetor auxiliar alocado na memória da CPU. Logo após, a GPU que irá receber a borda é selecionada com a primitiva *cudaSetDevice* e os dados são copiados para o vetor auxiliar de bordas na sua memória global. Ao final, a GPU que originou todo o processo é ativada novamente e o procedimento termina.

Nas situações onde as GPUs se encontram localizadas em máquinas distintas, o processo de troca de bordas é dado pelo conjunto de passos a seguir, executados por ambas as CPUs onde estas GPUs se localizam. Os dados da GPU são copiados da sua memória global para um vetor auxiliar na memória do *host*. Então a primitiva não-bloqueante MPI_Isend é chamada para enviar uma mensagem MPI com seus dados de borda para a máquina que contém a outra GPU e, logo após, a função bloqueante *MPI_Recv* é chamada para receber a borda da outra GPU. O *host* permanece bloqueado até que os dados sejam

devidamente recebidos e então faz uma cópia de tais informações da CPU para o vetor auxiliar na memória global da GPU. Em seguida, o *host* aguarda por meio da primitiva *MPI_Wait* até que sua mensagem tenha sido enviada à outra GPU, para somente assim alcançar o final do procedimento.

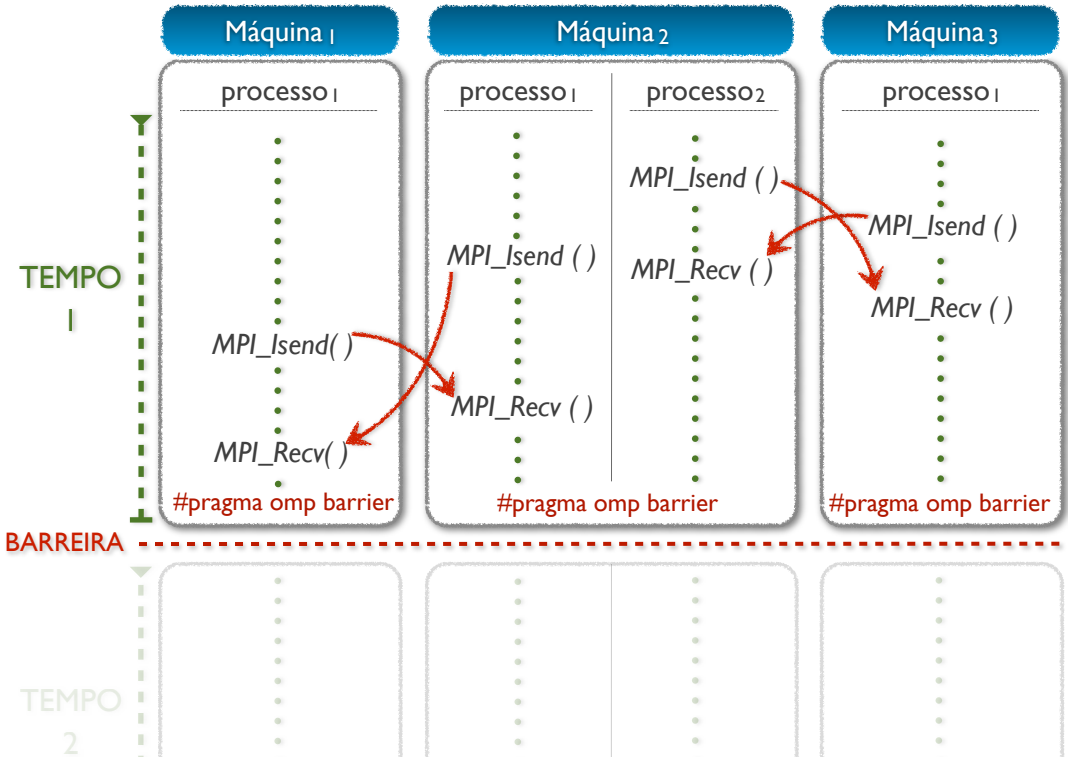


Figura 5.3: Processo de sincronização entre as GPUs

Além do obstáculo de troca de bordas entre GPUs de diferentes máquinas ter sido superado pela estratégia definida anteriormente, o problema de sincronização entre GPUs de diferentes máquinas também acaba sendo resolvido de forma implícita. Isso porque uma *thread* OpenMP que gerencie a primeira ou a última GPU de uma máquina terá de enviar sua respectiva borda e também receber a borda da máquina à sua direita ou esquerda a cada passo de tempo através da primitiva de comunicação *MPI_Recv*, que é de natureza bloqueante. Por conta desse fator e também da sincronização de todas as *threads* OpenMP da máquina com o uso da diretiva *#pragma omp barrier* ao final de cada iteração, todas as GPUs que participam da computação acabam ficando sincronizadas ao final de cada passo de tempo, fazendo com que as populações do modelo sejam calculadas corretamente. A Figura 5.3 ilustra o processo de sincronização entre todas as GPUs que participam da sincronização.

Algoritmo 4 Funcionamento geral da Versão 1

```
principal
    ... declaração de variáveis ...
    ... inicialização dos processos MPI ...
    ... criação de processos OpenMP para gerenciar cada GPU presente na máquina ...
    ... obtenção do número total de GPUs participantes da computação ...

    ... definição da submalha a ser calculada por cada GPU ...
    ... preenchimento da submalha com as condições iniciais das equações ...
    para t de t_i até t_f faça
        ... geração dos arquivos de saída das populações (caso habilitado) ...
        ... kernel para o cálculo dos pontos da submalha ...

        ... enviarEReceberBordas para troca de bordas e sincronização entre as GPUs ...
        ... sincronização das GPUs de uma mesma máquina ...
    fim-para
fim-principal
```

O Algoritmo 4 exibe um pseudocódigo do funcionamento geral da versão 1 do código de múltiplas GPUs, onde t é a variável que representa o tempo atual no laço temporal e t_i e t_f são os tempos inicial e final do laço.

Embora seja uma versão simples, a cópia de dados entre GPU e CPU, e vice-versa, prejudica muito o desempenho da aplicação. Isso sem contar o caso onde a GPU esteja em outra máquina, sendo necessário o uso de mais duas transferências de dados, agora envolvendo as duas CPUs com as primitivas MPI para envio e recebimento de mensagens. Logo, com o intuito de reduzir as operações de cópia de dados, uma segunda versão do código foi desenvolvida, utilizando o acesso direto a dados provido pelo UVA.

5.2.2 Versão 2

A segunda versão implementada faz o uso de um recurso especial oferecido por GPUs da classe Fermi ou em modelos mais atuais, o espaço virtual de endereçamento (UVA [36]). Este recurso permite que as GPUs de uma máquina compartilhem seus endereços de memória, de modo que cada uma possa acessar os dados da outra de forma direta, sem a necessidade de qualquer cópia de memória entre CPU e GPU.

Para que seja possível utilizar tal funcionalidade, é necessário que cada GPU de um par

faça uma chamada à função *cudaEnablePeerAccess*, isso porque o acesso direto garantido pela chamada de tal função é unidirecional e somente permitido entre pares de GPUs. No momento em que ambas as GPUs tenham chamado a primitiva *cudaEnablePeerAccess*, um *peer* de comunicação entre elas é estabelecido e, a partir daí, o acesso às posições de memória de cada uma pode ser diretamente acessado por qualquer uma das duas.

A estratégia para a troca de bordas entre GPUs de diferentes máquinas e entre aquelas de mesma máquina, mas que não ofereçam suporte ao UVA, é a mesma definida na versão 1 do código de múltiplas GPUs, sem alteração alguma.

Embora nesta versão o acesso aos dados de duas GPUs localizadas em uma mesma máquina que ofereçam suporte ao UVA seja feito de forma direta, o processo de envio das bordas entre duas GPUs ainda é realizado de forma simples, pois o *kernel* tem que calcular todos os pontos da submalha para somente depois realizar a troca dos dados de borda. Visando minorar o impacto no desempenho decorrente deste problema, uma última versão foi implementada na tentativa de sobrepor as transferências das bordas com computação.

5.2.3 Versão 3

Visto que os processos participantes da computação devem trocar uma pequena porção de seus dados computados a todo momento, a última versão do código tenta sobrepor essas trocas de bordas (ou seja, comunicação) com computação. Como a quantidade de pontos de borda é menor que o número de pontos mais internos da submalha de cada GPU, é provável que o cálculo dos pontos de borda termine em pouco tempo, possibilitando que seus dados sejam enviados a outro processo enquanto os pontos mais internos estejam sendo processados de forma simultânea, havendo uma certa sobreposição da comunicação com computação.

Para alcançar esse objetivo, foram criados dois *kernels* CUDA: um irá calcular os pontos de borda que serão utilizados na troca entre os processos e o outro será responsável por computar os pontos restantes, ou mais internos. A idéia é que por meio do uso de diferentes *streams* CUDA tal abordagem possa ser implementada, associando cada *kernel* a um diferente *stream*.

Algoritmo 5 Funcionamento geral da Versão 3

```
principal
    ... declaração de variáveis ...
    ... inicialização dos processos MPI ...
    ... criação de processos OpenMP para gerenciar cada GPU presente na máquina ...
    ... obtenção do número total de GPUs participantes da computação ...
    ... definição da submalha a ser calculada por cada GPU ...
    ... preenchimento da submalha com as condições iniciais das equações ...
    ... criação de duas streams para gerenciar operações nos pontos de borda e mais internos ...
    para t de t_i até t_f faça
        ... geração dos arquivos de saída das populações (caso habilitado) ...
        ... kernel para o cálculo dos pontos de borda da submalha ...
        ... kernel para o cálculo dos pontos mais internos da submalha ...
        ... enviarEReceberBordas para troca de bordas e sincronização entre as GPUs ...
        ... sincronização das GPUs de uma mesma máquina ...
    fim-para
fim-principal
```

O Algoritmo 5 representa um pseudocódigo do funcionamento geral da versão mais elaborada do código de múltiplas GPUs (versão 3), onde t é a variável que representa o tempo atual no laço temporal e t_i e t_f são os tempos inicial e final do laço.

Com o uso de duas *streams* CUDA, a tarefa de calcular o *kernel* para os pontos mais internos pode ser executada simultaneamente com a cópia dos pontos de borda para a memória da CPU, de modo que essa transferência de dados possa ser iniciada imediatamente após o cálculo da borda ter sido realizado. A Figura 5.4 ilustra o processo descrito anteriormente: primeiro o *kernel* para o cálculo dos pontos de borda é invocado pela *stream* denominada de "pontos de borda" enquanto a outra, chamada "pontos mais internos" fica suspensa. A partir do momento em que a computação do *kernel* para os pontos de borda termina, a operação de cópia assíncrona de dados para a CPU é executada concorrentemente com a operação de cálculo do *kernel* para os pontos mais internos (*stream* pontos mais internos). A limitação imposta à execução sequencial de mais de um *kernel* está presente em GPUs com versão do *compute capability* abaixo de 2.0. O mesmo não acontece com placas da classe Fermi (*compute capability* 2.x) e

modelos mais atuais, porém, dependendo do modelo do *chip*, algumas exceções podem ser encontradas.

Utilizando a função *cudaMemcpyAsync*, após o término de execução da *stream* responsável pelos pontos de borda, os respectivos dados poderão começar a ser copiados para o *host* utilizando a mesma *stream*, e assim este pode realizar a troca dos resultados com os outros processos participantes da computação. Enquanto isso, os pontos restantes podem ser simultaneamente computados pela outra *stream* (Figura 5.4).

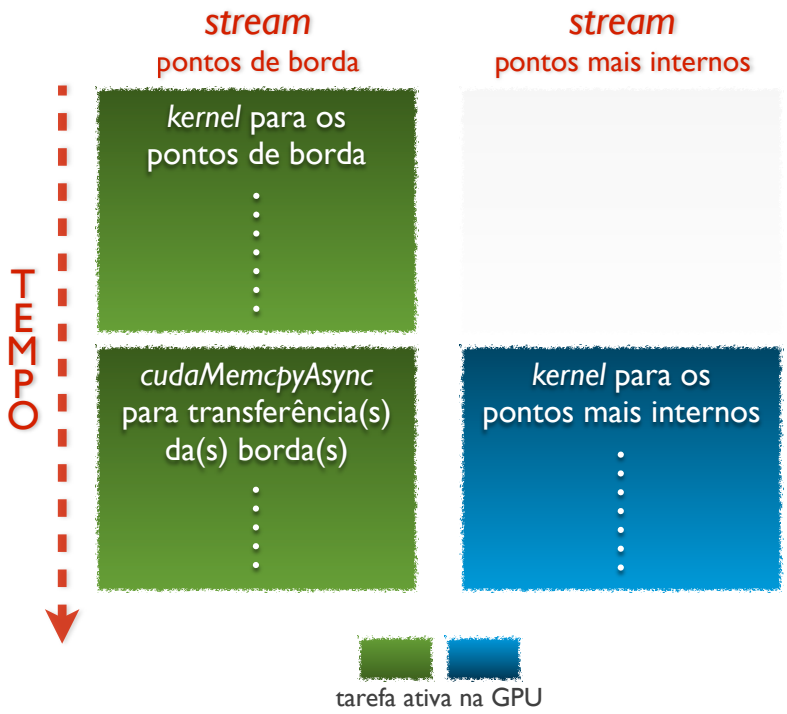


Figura 5.4: Concorrência de operações em *streams* CUDA independentes

Ao utilizar tal estratégia, é esperado que se consiga sobrepor parte da comunicação com o processamento dos pontos mais internos da submalha, de modo a reduzir ainda mais o tempo de execução do modelo.

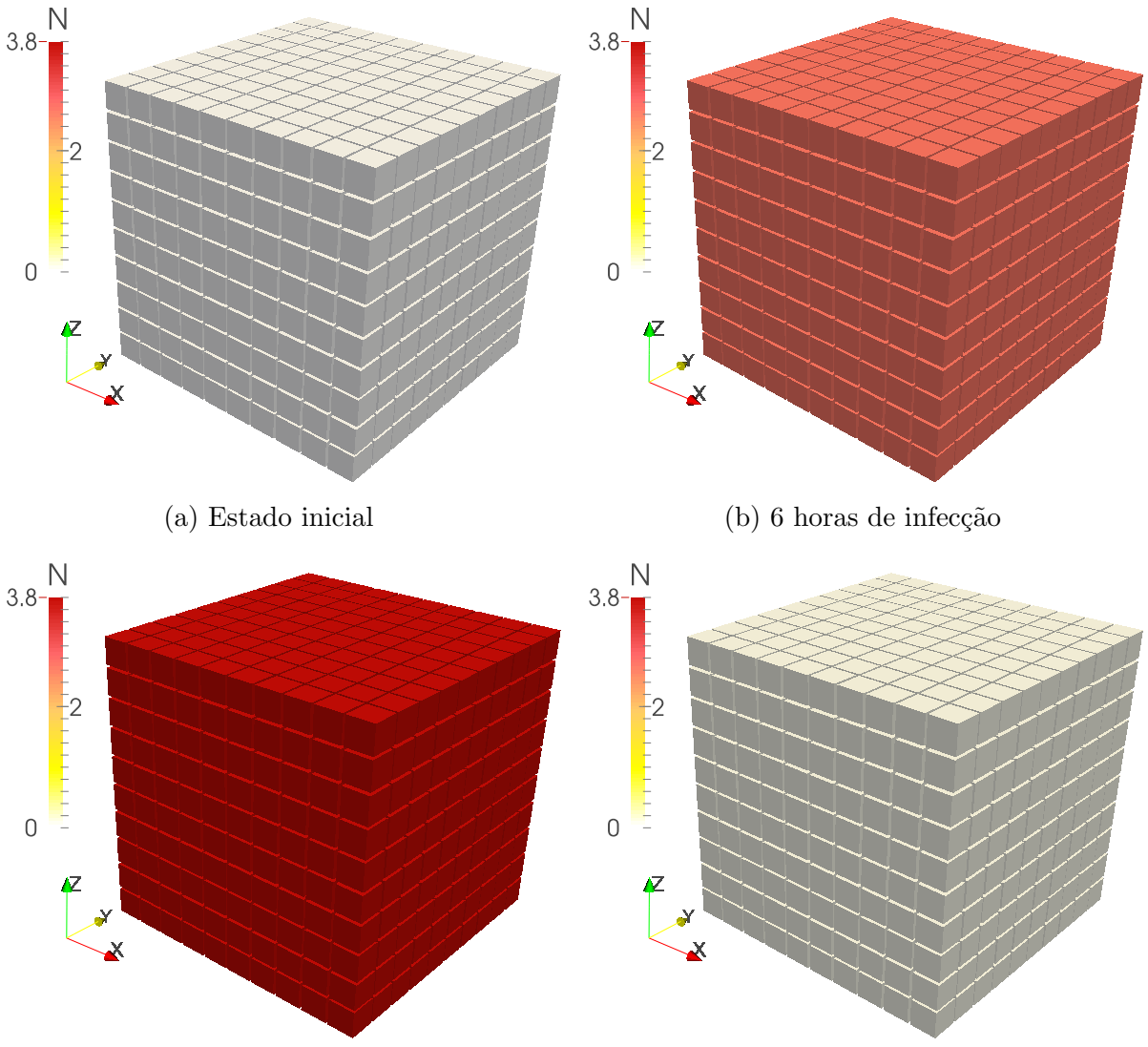
6 RESULTADOS

No presente capítulo são apresentados alguns resultados biológicos obtidos pela avaliação do novo modelo 3D do SIH (Sistema Imunológico Humano) inato desenvolvido neste trabalho para observar a contribuição de cada um dos componentes simulados na resposta imune inata ao LPS (Lipopolissacarídeo). Também são apresentados os tempos de execução alcançados pela versão sequencial (de uma CPU - *Central Processing Unit*) do modelo, bem como de cada uma das três implementações do algoritmo paralelo que utilizam múltiplas GPUs (*Graphics Processing Unit*), avaliados em cenários com diferentes configurações de tamanho de malha e de número de GPUs envolvidas na computação. Por fim, são apresentadas as contribuições e os ganhos obtidos em desempenho por cada versão paralela implementada do modelo.

6.1 Resultados Biológicos

A atual seção apresenta os resultados biológicos obtidos pela simulação de 5 dias de infecção causada pela exposição de uma seção de tecido 3D humano com 1 mm^3 de volume à endotoxina LPS. De acordo com a discretização espacial utilizada, já definida na Tabela 5.1, este tecido é representado por uma malha de $10 \times 10 \times 10$ pontos. As figuras a seguir ilustram a evolução temporal das populações de neutrófilos, citocina anti-inflamatória, LPS, citocina pró-inflamatória, macrófagos ativados e grânulos proteicos no tecido, em seus estados iniciais e após 6 horas, 1 dia e 5 dias de infecção.

A Figura 6.1 exibe a concentração da população de neutrófilos no tecido infectado pelo LPS. Pode-se notar que no estado incial, ou seja, no instante em que o tecido está para ser invadido pelo LPS, a quantidade de neutrófilos no local é nula (Figura 6.1a), porém, a partir do momento em que o LPS é detectado no organismo, a população de neutrófilos começa a aumentar, de modo a combater tal ameaça (Figura 6.1b). Em aproximadamente 24 horas após o início da infecção (Figura 6.1c), a população de neutrófilos no tecido alcança sua concentração máxima e, a partir daí, começa a diminuir, já que o invasor foi devidamente combatido. No quinto dia, a população de neutrófilos no local infectado é quase nula (Figura 6.1d).



(a) Estado inicial

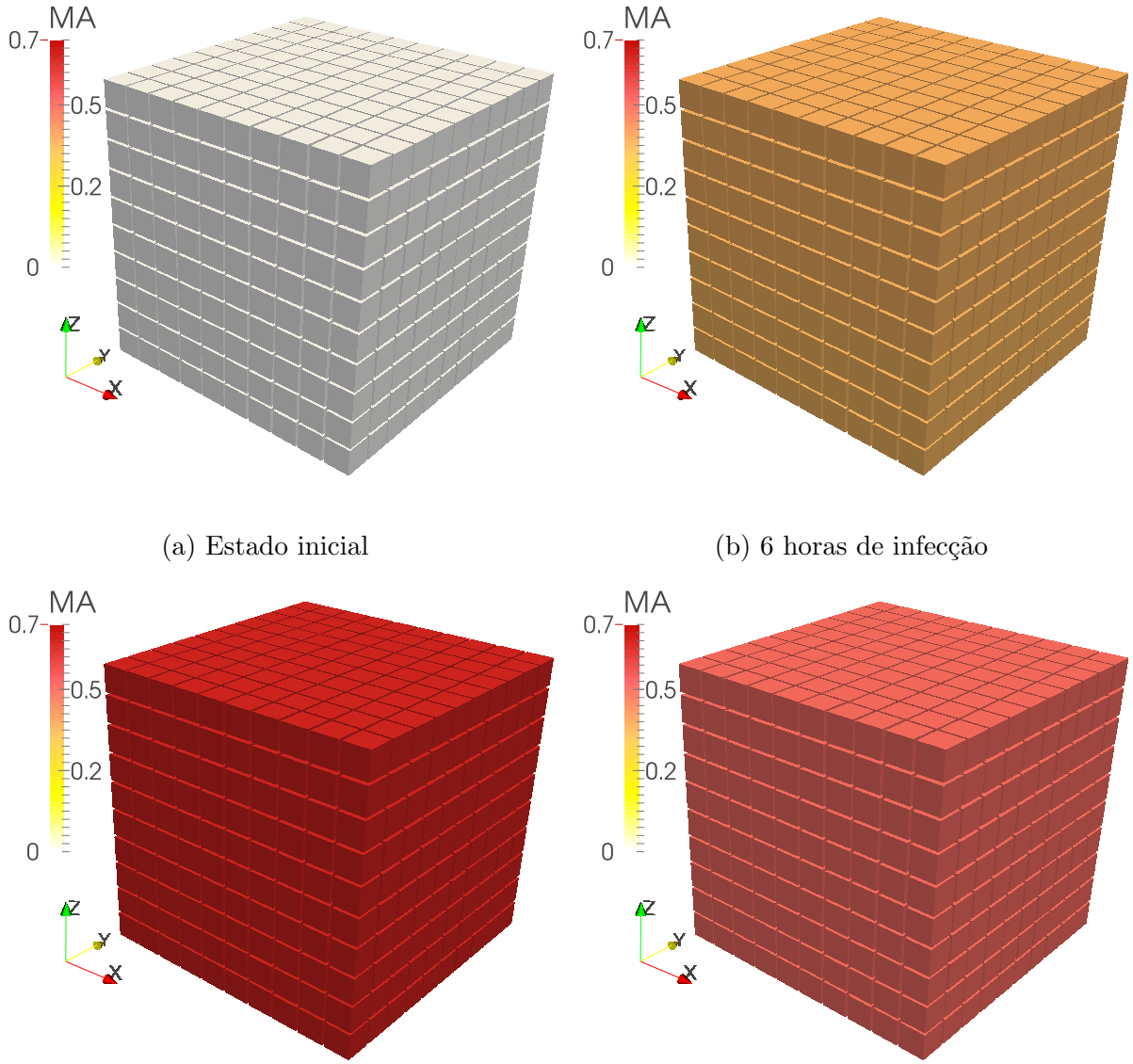
(b) 6 horas de infecção

(c) 1 dia de infecção

(d) 5 dias de infecção

Figura 6.1: Evolução da população de neutrófilos no tecido

Além do crescimento da população de neutrófilos, há também um aumento na concentração de macrófagos ativados no tecido infectado (Figura 6.2). Tais crescimentos populacionais ocorrem devido à ação de substâncias como as citocinas pró-inflamatórias (Figura 6.3) e os grânulos proteicos (Figura 6.4), ambos com produção iniciada a partir do momento em que o LPS é detectado no organismo. A medida que este vai sendo eliminado (Figura 6.5), a concentração de citocinas anti-inflamatórias vai aumentando (Figura 6.6). Isso traz como consequência o bloqueio da ativação de mais macrófagos em repouso, como também a inibição da produção de citocinas pró-inflamatórias, acarretando em um decaimento dessas populações.



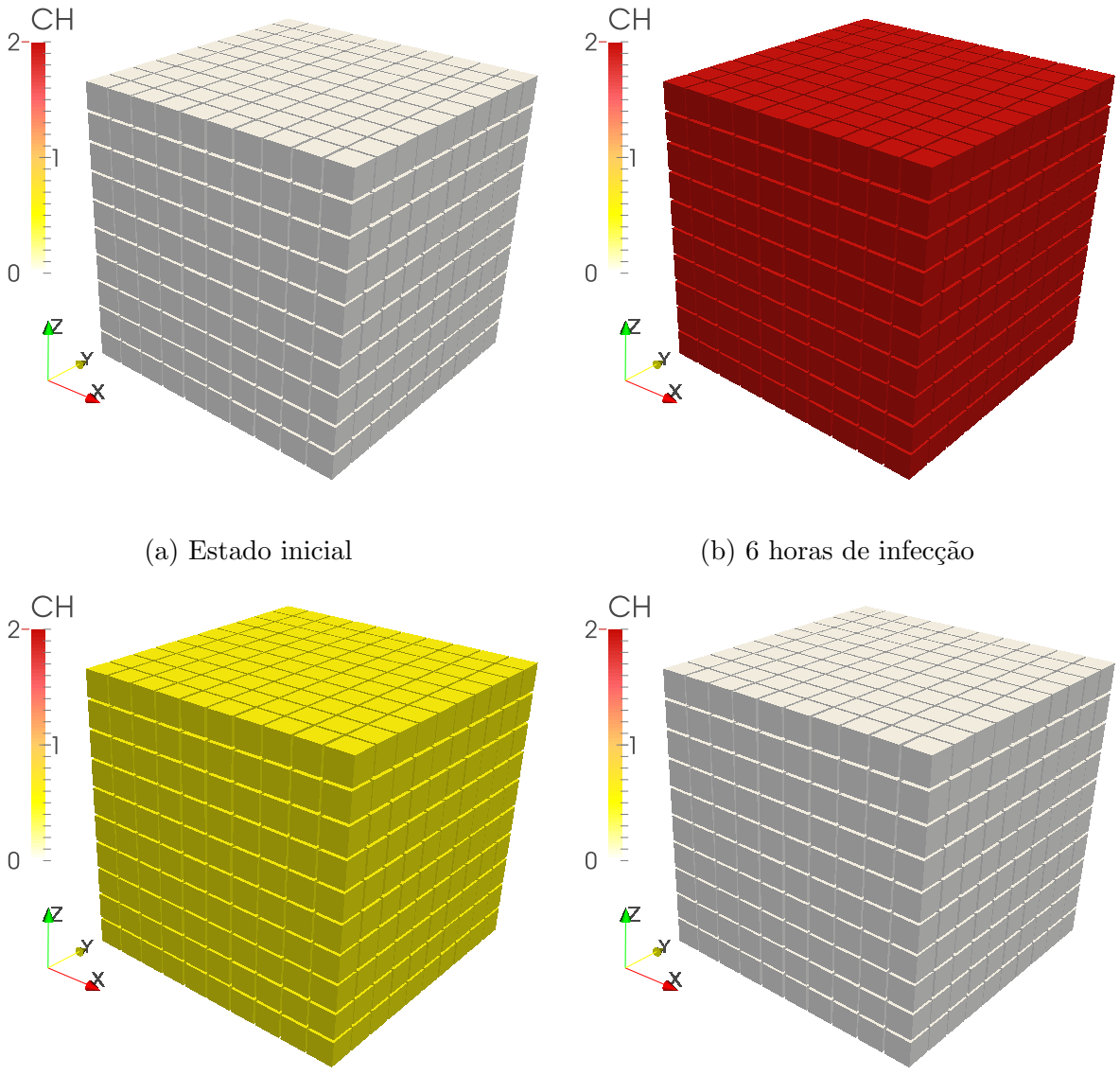
(a) Estado inicial

(b) 6 horas de infecção

(c) 1 dia de infecção

(d) 5 dias de infecção

Figura 6.2: Evolução da população de macrófagos ativados no tecido



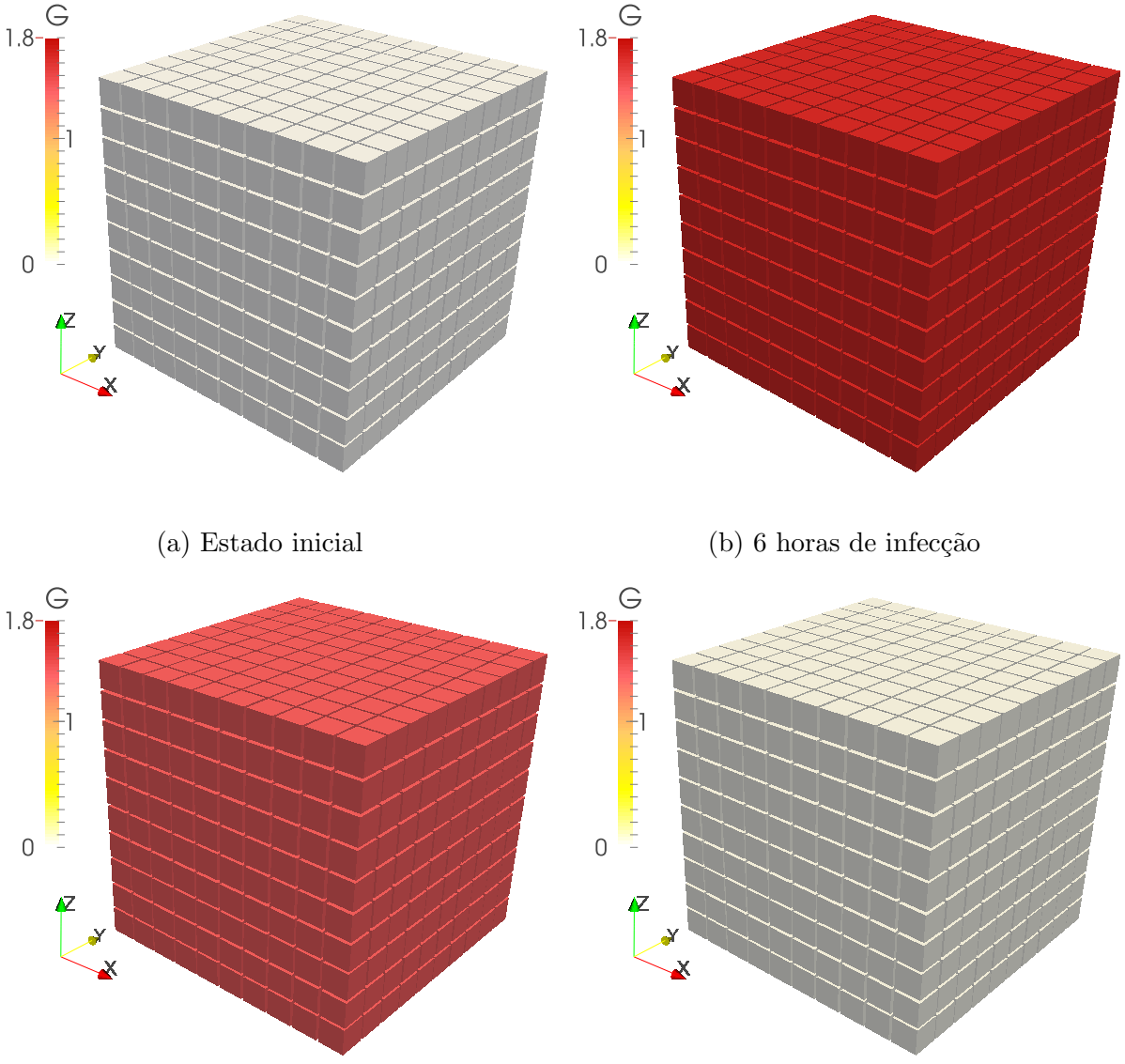
(a) Estado inicial

(b) 6 horas de infecção

(c) 1 dia de infecção

(d) 5 dias de infecção

Figura 6.3: Evolução da população de citocina pró-inflamatória no tecido



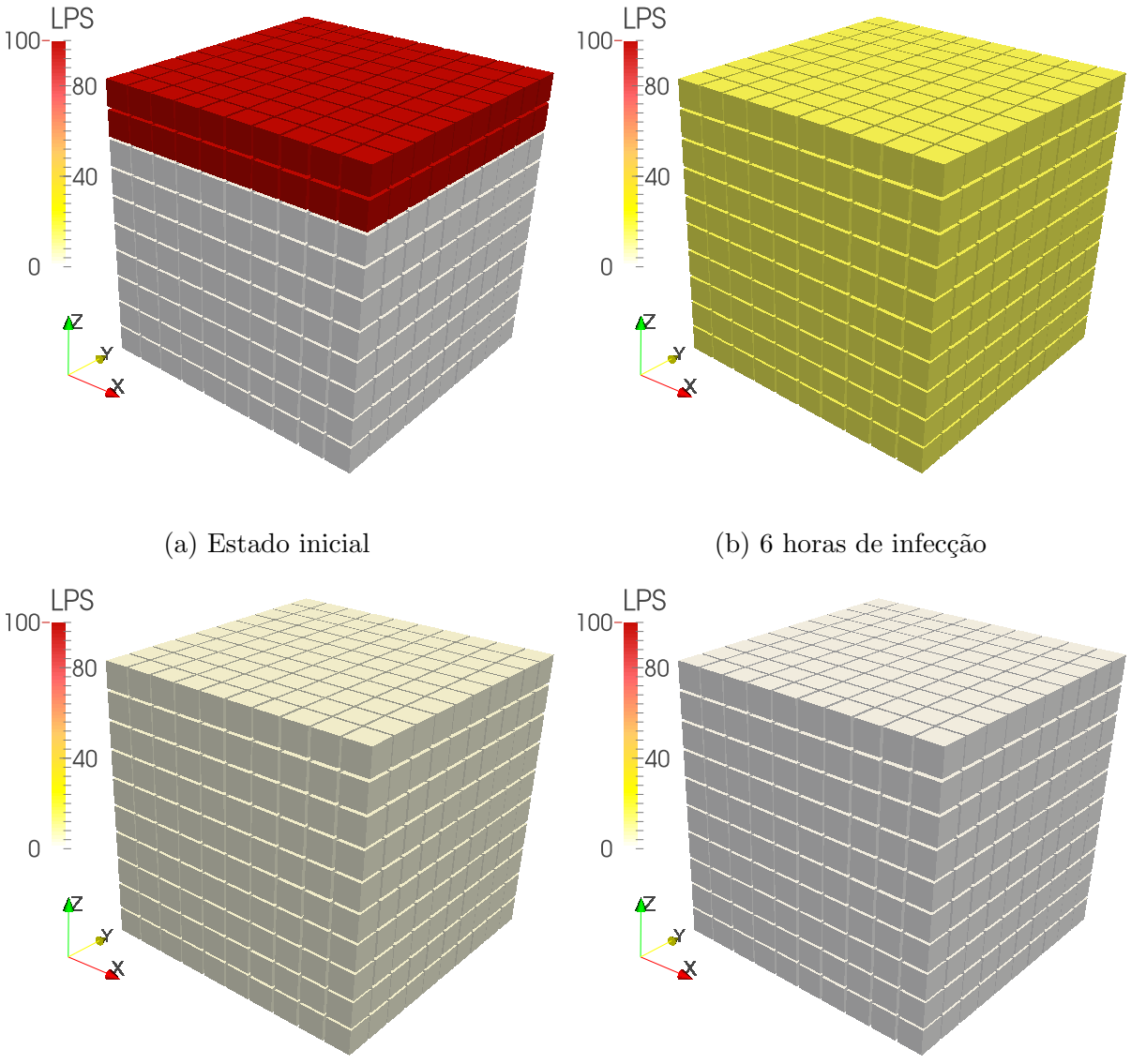
(a) Estado inicial

(b) 6 horas de infecção

(c) 1 dia de infecção

(d) 5 dias de infecção

Figura 6.4: Evolução da população de grânulos proteicos no tecido



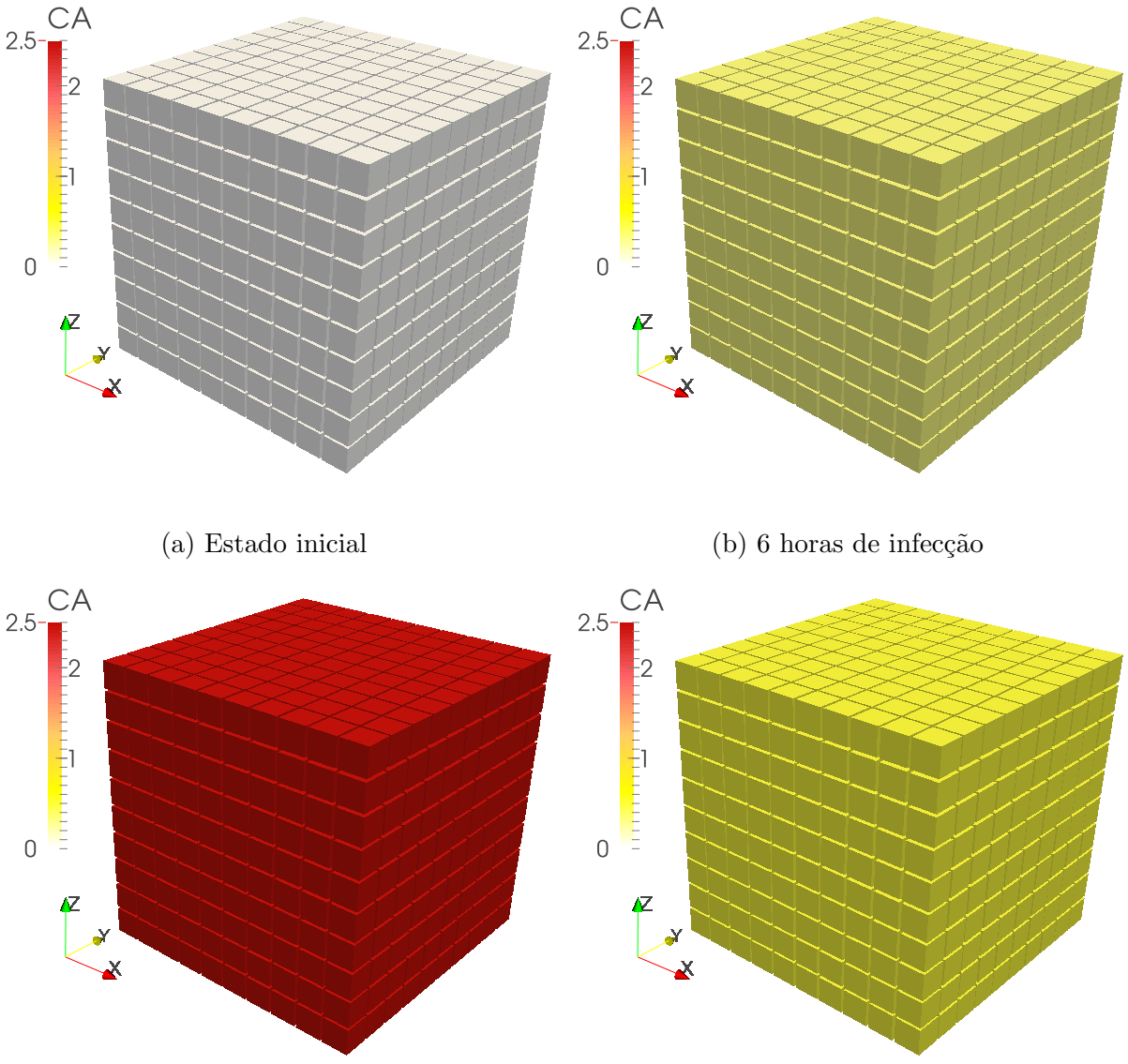
(a) Estado inicial

(b) 6 horas de infecção

(c) 1 dia de infecção

(d) 5 dias de infecção

Figura 6.5: Evolução da população da endotoxina LPS no tecido



(a) Estado inicial

(b) 6 horas de infecção

(c) 1 dia de infecção

(d) 5 dias de infecção

Figura 6.6: Evolução da população de citocina anti-inflamatória no tecido

6.2 Parâmetros do Modelo

Com o intuito de avaliar os ganhos de desempenho obtidos pelas versões de múltiplas GPUs implementadas neste trabalho, o modelo foi submetido a testes em diferentes cenários previamente selecionados.

Para cada experimento foram simulados diferentes tamanhos de tecido, com volumes de 125 mm^3, 1 cm^3, 3,375 cm^3, 8 cm^3 e 15,625 cm^3, representados pelas malhas de $50 \times 50 \times 50$, $100 \times 100 \times 100$, $150 \times 150 \times 150$, $200 \times 200 \times 200$ e $250 \times 250 \times 250$ pontos, respectivamente. Tais valores são casos válidos, uma vez que a simulação possa ser de um pequeno pedaço de tecido a um órgão inteiro. De acordo com a Tabela A.1 presente no Apêndice, a discretização da malha gera cerca de 1.000 pontos para cada 1 mm^3 de tecido.

Uma outra variável do modelo é o tempo de simulação da infecção em número de dias. Definido na Tabela 5.1 no capítulo Implementação, cada dia de infecção é representado por 1.000.000 iterações. Devido ao alto tempo gasto na simulação das malhas para a versão sequencial do código, apenas 1 % do período equivalente a um dia de infecção (10.000 iterações) será simulado em todas as versões da aplicação e em todos os cenários. Isso pode ser feito pois todo o processo realizado a cada iteração da simulação é dado sempre pelo mesmo conjunto de instruções, cujo tempo de computação é muito próximo para qualquer valor usado como dado nas computações, logo qualquer número de iterações pode ser tomado como uma amostra da simulação. Além disso, o maior foco deste trabalho consiste na avaliação das técnicas e abordagens empregadas para a redução no tempo total gasto em uma simulação do novo modelo criado e não em seus resultados biológicos propriamente ditos. Estima-se que a simulação de um dia de infecção em uma malha de $250 \times 250 \times 250$ pontos na versão sequencial leve cerca de 240 dias para ser completada.

6.3 Ambiente de Execução

As versões paralelas para múltiplas GPUs foram avaliadas em dois ambientes de execução diferentes, de modo que cada versão implementada irá apresentar um determinado ganho de acordo com o ambiente em que foi avaliada. Isso foi feito porque as máquinas de um ambiente possuem GPUs diferentes das encontradas nas máquinas do outro ambiente. Entre as diferenças encontradas estão o número de núcleos de processamento, quantidade

de memória global disponível e suporte ao recurso UVA (*Unified Virtual Addressing*). Já a versão sequencial do código utiliza somente a CPU para realizar as computações, e como as configurações de ambos os ambientes são exatamente as mesmas em relação à quantidade de memória e modelo da CPU, o tempo gasto para a versão sequencial é praticamente o mesmo nos dois ambientes. Logo qualquer um dos dois ambientes poderá ser considerado na avaliação do código sequencial.

Cada uma das máquinas que formam os dois ambientes possuem processador Intel Xeon E5620 2.4 GHz, com 12 GB de memória RAM (*Random Access Memory*) e sistema operacional Linux Rocks 5.4 64-*bits*, com *kernel* versão 2.6.18. As máquinas são interligadas através do cabo RJ45 com arquitetura de conexão *Gigabit Ethernet*.

No processo de compilação de um programa, existe a possibilidade de fazer com que o compilador tente realizar algumas otimizações na aplicação de modo a tornar a sua execução mais rápida. Através do uso de *flags* o usuário poderá definir e habilitar algumas das otimizações disponíveis. Como exemplo de *flags* do compilador *nvcc* temos *-Xptxas* e *-Xopencc*, já para o compilador *g++*, podem ser usados *-O1*, *-O2* e *-O3* para otimizar um programa. Para compilar todas as versões de múltiplas GPUs implementadas neste trabalho foi utilizado o compilador *nvcc* versão 4.2 sem quaisquer opções de otimização. A versão sequencial foi compilada pelo compilador *g++* versão 4.1.2 também sem quaisquer opções de otimização.

Em todas as versões para múltiplas GPUs implementadas neste trabalho, o número de *threads* por bloco utilizado é igual a 256 e o tamanho do *grid* é calculado automaticamente com base nas dimensões da malha, no número total de GPUs participantes da computação e no tamanho do bloco de *threads*.

Cada configuração de dimensões da malha foi executada 5 vezes para garantir a confiabilidade dos tempos colhidos: o desvio padrão ficou abaixo de 2 % para todos os cenários testados. A cada execução, a aplicação *time* do Linux foi utilizada para medir o tempo gasto na execução do programa.

O fator de aceleração, ou *speedup*, foi utilizado para mensurar os ganhos obtidos pelos códigos paralelos em relação ao código sequencial. Logo o *speedup* poderá ser calculado empregando a seguinte Equação 6.1:

$$S(p) = \frac{t_s}{t_p}, \tag{6.1}$$

onde t_s é o tempo de execução sequencial e t_p é o tempo de execução da versão paralela com p processadores.

A seguir, os dois ambientes computacionais utilizados nos experimentos deste trabalho são devidamente especificados.

6.3.1 Ambiente 1

O primeiro ambiente de execução conta com um conjunto de 4 máquinas, cada uma com duas GPUs NVIDIA Tesla C1060 com 30 SMs (*Streaming Multiprocessor*) e 8 *cores* por SM, totalizando 240 CUDA *cores* em cada uma. A GPU Tesla C1060 possui 4 GB GDDR3 de memória global, 65 KB de memória para constantes, versão *compute capability* 1.3, versão CUDA 4.2 e não oferece suporte ao recurso UVA.

6.3.2 Ambiente 2

No segundo ambiente de execução existem apenas 2 máquinas, cada uma com duas GPUs NVIDIA Tesla M2050 com 14 SMs e 32 *cores* por SM, totalizando 448 CUDA *cores* em cada uma. A GPU Tesla M2050 possui uma memória global de aproximadamente 2,5 GB GDDR3, uma memória de 65 KB para constantes, versão *compute capability* 2.0, versão CUDA 3.2 e oferece suporte ao recurso UVA.

As próximas seções irão apresentar os *speedups* alcançados por cada versão paralela desenvolvida neste trabalho.

6.4 Resultados para a Versão 1

Esta seção apresenta os ganhos obtidos com a primeira versão para múltiplas GPUs em relação à versão sequencial da aplicação que realiza a computação somente na CPU. A versão 1 do código é a mais simples: os dados de borda são trocados entre duas GPUs, que residam ou não na mesma máquina, utilizando para este propósito a memória da CPU.

Os tempos de execução foram colhidos nos dois ambientes descritos anteriormente para diferentes configurações de malha e, no caso da versão para múltiplas GPUs, variando-se a quantidade destas. As Tabelas 6.1 e 6.2 apresentam, respectivamente, os resultados obtidos com a primeira versão do código para múltiplas GPUs no primeiro e segundo ambientes de execução. Todos os tempos foram medidos em segundos e a versão 1 é

representada na tabela pela quantidade de GPUs empregadas nas simulações. Os maiores *speedups* alcançados para cada configuração de malha estão marcados em negrito.

Pode-se notar ao observar os dados presentes na Tabela 6.1 que os maiores *speedups* obtidos para malhas pequenas ($50 \times 50 \times 50$ e $100 \times 100 \times 100$) são alcançados com o uso de apenas duas GPUs e não com quatro ou oito, como esperado. Isso acontece porque com um número maior de GPUs executando a aplicação, o tamanho da fatia da malha que será processada por cada uma é muito pequena, acarretando na subutilização dos recursos oferecidos pela GPU. O aumento na quantidade de GPUs utilizadas na simulação também implica em um maior custo na comunicação entre os processos para o envio das bordas, elevando o tempo total da execução.

A medida que as dimensões da malha vão aumentando e o número de GPUs utilizadas permanece o mesmo, a submalha que cada GPU precisa processar também aumenta, e com isso surge a necessidade de que mais GPUs sejam incorporadas à computação para que o tamanho de cada fatia não seja tão grande para seu cálculo. Para a malha de $150 \times 150 \times 150$ pontos, as acelerações obtidas pelas configurações de quatro e oito GPUs ultrapassam aquela alcançada com duas GPUs, já que agora cada GPU está sendo totalmente utilizada na computação das suas respectivas submalhas, contribuindo para uma redução no tempo de execução.

A partir da dimensão de malha $200 \times 200 \times 200$, o *speedup* alcançado pela versão com oito GPUs é o melhor dentre as três configurações.

Os resultados obtidos nos experimentos realizados no ambiente 2 foram os melhores (Tabela 6.2). Devido ao maior número de núcleos CUDA presentes nas GPUs NVIDIA Tesla M2050 do segundo ambiente de execução, os tempos de simulação reduziram consideravelmente quando comparados àqueles coletados no ambiente 1, alcançando melhores *speedups* em todos os cenários simulados. Pode-se verificar que em alguns casos a redução nos tempos de execução, em relação aos tempos mensurandos no primeiro ambiente, foi muito significativa, chegando a mais de 50 %.

Analisando os resultados obtidos no ambiente 2, somente na configuração de malha de $250 \times 250 \times 250$ pontos é que a versão com quatro GPUs conseguiu superar a de duas GPUs. Em tal configuração, as dimensões da fatia que é designada a cada GPU possui um número considerável de pontos no tecido a serem computados, chegando a valores da ordem de milhões de pontos.

Tabela 6.1: *Speedups* obtidos no ambiente de execução 1 pela primeira versão com múltiplas GPUs em relação ao código sequencial

Malha	**Versão**	**Tempo médio (s)**	**Desvio padrão**	***Speedup***
50x50x50	sequencial	2.016,5	0,1 %	-
	1 GPU	14,6	0%	137,8
	2 GPUs	9,6	1,4 %	**209,1**
	4 GPUs	15,8	1,2 %	127,9
	8 GPUs	21,7	0,9 %	92,8
100x100x100	sequencial	16.680,1	0,6 %	-
	1 GPU	111,4	0 %	149,7
	2 GPUs	65,4	0,8 %	**255,0**
	4 GPUs	67,6	1,2 %	246,7
	8 GPUs	85,3	0,8 %	195,7
150x150x150	sequencial	49.638,6	1,1 %	-
	1 GPU	384,7	0 %	129,0
	2 GPUs	219,7	0,5 %	226,0
	4 GPUs	186,9	1,5 %	**265,6**
	8 GPUs	202,7	0,6 %	244,9
200x200x200	sequencial	110.675,4	0,7 %	-
	1 GPU	894,4	0 %	123,7
	2 GPUs	506,9	0,1 %	218,3
	4 GPUs	383,1	0,9 %	288,9
	8 GPUs	382,4	0,2 %	**289,4**
250x250x250	sequencial	208.005,9	0,9 %	-
	1 GPU	1.794,5	0 %	115,9
	2 GPUs	1.011,9	0,2 %	205,6
	4 GPUs	705,6	0,1 %	294,8
	8 GPUs	646,6	1,2 %	**321,7**

Tabela 6.2: *Speedups* obtidos no ambiente de execução 2 pela primeira versão com múltiplas GPUs em relação ao código sequencial

Malha	**Versão**	**Tempo médio (s)**	**Desvio padrão**	***Speedup***
50x50x50	sequencial	2.016,5	0,1 %	-
	1 GPU	6,4	0,1 %	315,0
	2 GPUs	5,4	0,9 %	**370,2**
	4 GPUs	13,8	1,7 %	146,1
100x100x100	sequencial	16.680,1	0,6 %	-
	1 GPU	49,7	0,9 %	335,7
	2 GPUs	31,7	0,4 %	**526,6**
	4 GPUs	50,8	1,6 %	328,1
150x150x150	sequencial	49.638,6	1,1 %	-
	1 GPU	167,5	0 %	296,3
	2 GPUs	100,8	0,7 %	**492,3**
	4 GPUs	127,2	1,9 %	390,3
200x200x200	sequencial	110.675,4	0,7 %	-
	1 GPU	392,1	0 %	282,3
	2 GPUs	229,1	0,7 %	**483,0**
	4 GPUs	247,0	0,8 %	448,1
250x250x250	sequencial	208.005,9	0,9 %	-
	1 GPU	778,0	0 %	267,4
	2 GPUs	450,0	0,4 %	462,2
	4 GPUs	425,1	1,7 %	**489,3**

6.5 Resultados para a Versão 2

A atual seção apresenta os ganhos obtidos ao avaliar a segunda versão do código no segundo ambiente de execução. Em relação à primeira versão implementada, agora não é mais necessária a cópia de dados entre as GPUs de uma máquina que ofereça suporte ao UVA. O UVA permite uma comunicação direta entre duas GPUs, provendo recursos de acesso e cópia direta de dados, o que torna a computação mais eficiente sem o *overhead* causado pela cópia dos dados de borda. Como as máquinas que compõem o primeiro ambiente de execução não possuem suporte ao UVA, não foram utilizadas nesta avaliação. A configuração com o uso de apenas 1 GPU também não será considerada, uma vez que não haverá troca de dados entre GPUs.

A Tabela 6.3 apresenta os resultados para a segunda versão com diferentes tamanhos de malha e quantidades de GPUs. Para efeitos de comparação, são apresentados também os resultados obtidos pela primeira versão do código. A última coluna da Tabela 6.3 mostra os ganhos percentuais obtidos pela segunda versão em relação a primeira. Em todas as simulações realizadas, a segunda versão mostrou-se mais rápida que a primeira e, para a malha de $50 \times 50 \times 50$ pontos, apresentou os maiores ganhos de desempenho, da ordem de 35 %.

Logo a abordagem empregada nesta segunda versão do código se mostrou muito eficiente e contribuiu positivamente para a aceleração alcançada em relação à primeira versão do código. A Figura 6.7 exibe um gráfico com os *speedups* em relação ao código sequencial alcançados pelas versões 1 e 2 no segundo ambiente de execução. Para a segunda versão do código, a malha $100 \times 100 \times 100$ com 2 GPUs conseguiu a maior aceleração dentre todas as outras configurações, sendo até 652 vezes mais rápida que a versão sequencial.

Tabela 6.3: Ganhos obtidos no ambiente de execução 2 pela segunda versão em relação à primeira

Malha	**Versão**		**Tempo médio (s)**	**Desvio padrão**	**Ganho**
50x50x50	Versão 1	2 GPUs	5,4	0,9 %	-
		4 GPUs	13,8	1,7 %	-
	Versão 2	2 GPUs	3,5	0,8 %	35,2 %
		4 GPUs	12,7	0,5 %	7,9 %
100x100x100	Versão 1	2 GPUs	31,7	0,4 %	-
		4 GPUs	50,8	1,6 %	-
	Versão 2	2 GPUs	25,6	0,2 %	19,3 %
		4 GPUs	48,0	1,2 %	5,5 %
150x150x150	Versão 1	2 GPUs	100,8	0,7 %	-
		4 GPUs	127,2	1,9 %	-
	Versão 2	2 GPUs	85,1	0,1 %	15,6 %
		4 GPUs	120,0	0,5 %	5,7 %
200x200x200	Versão 1	2 GPUs	229,1	0,7 %	-
		4 GPUs	247,0	0,8 %	-
	Versão 2	2 GPUs	196,9	0,0 %	14,1 %
		4 GPUs	235,0	1,1 %	4,9 %
250x250x250	Versão 1	2 GPUs	450,0	0,4 %	-
		4 GPUs	425,1	1,7 %	-
	Versão 2	2 GPUs	391,6	0,0 %	13,0 %
		4 GPUs	412,1	0,7 %	3,1 %

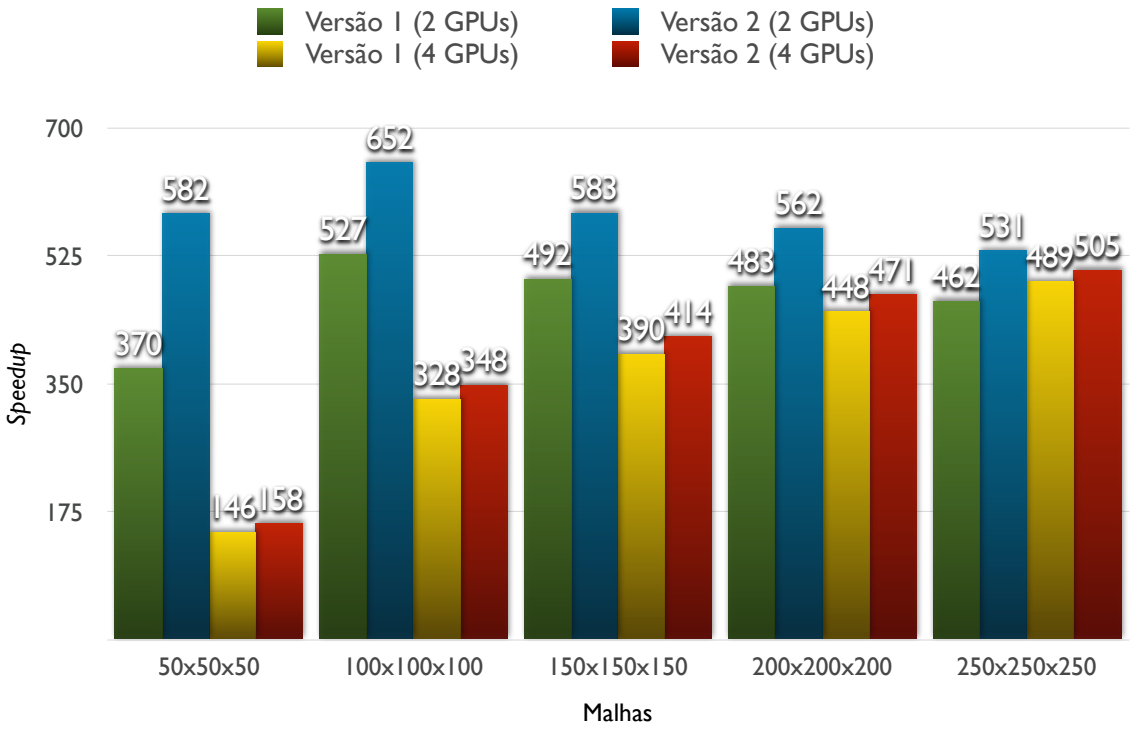


Figura 6.7: *Speedups* alcançados pelas versões 1 e 2 em relação à versão sequencial. As duas primeiras barras representam o *speedup* para duas GPUs, sendo a primeira para a versão 1 e a segunda para a versão 2. As duas barras seguintes representam o *speedup* para quatro GPUs, sendo a terceira barra para a versão 1 e a quarta para a versão 2 do código.

Embora os resultados tenham se mostrado muito bons para as primeiras versões implementadas, o processo de envio das bordas entre duas GPUs ainda pode ser melhorado, visto que o *kernel* tem que primeiro calcular todos os pontos da submalha para somente depois realizar a troca dos dados de borda com outras GPUs à sua esquerda e/ou direita. A última versão do código tenta sobrepor as transferências das bordas com computação, e seus resultados são apresentados na próxima seção.

6.6 Resultados para a Versão 3

Na presente seção os resultados obtidos pela implementação da terceira e última versão do código para múltiplas GPUs são apresentados. A versão 3 foi criada com o intuito de reduzir o custo de comunicação imposto pelo uso de múltiplas GPUs, realizando a computação dos pontos mais internos da malha enquanto a transferência de dados de

borda é efetuada. No caso do segundo ambiente de execução, faz-se também uso do recurso UVA para que o acesso aos dados entre duas GPUs localizadas na mesma máquina possa ser realizado diretamente. Os resultados para essa última versão em relação ao código sequencial foram colhidos nos primeiro e segundo ambientes de execução e são apresentados nas Tabelas 6.4 e 6.5, respectivamente. Os maiores *speedups* alcançados para cada configuração de malha da última versão do código estão marcados em negrito.

Novamente, em malhas de dimensões pequenas, os maiores *speedups* são alcançados com o uso de duas ou quatro GPUs, visto que o tamanho da fatia a ser processada por cada uma é pequena. A medida que o tamanho da malha simulada vai aumentando, as fatias a serem computadas por cada GPU também acabam aumentando e o uso de mais poder de processamento (ou seja, GPUs) se faz necessário. Logo a partir da dimensão de malha $200 \times 200 \times 200$, o uso de oito GPUs oferece um melhor resultado comparado às configurações com menor número e acarreta em uma execução da aplicação da ordem de 500 vezes mais rápida que a versão sequencial.

Como pode ser observado em ambas as tabelas apresentadas, todos os tempos de execução são menores que aqueles obtidos pela versão 1 (Tabelas 6.1 e 6.2), porém a versão 2 (Tabela 6.3) apresenta resultados melhores que os encontrados aqui para a configuração com duas GPUs. A explicação é que como as duas GPUs estão localizadas na mesma máquina e o acesso aos dados de cada uma é realizado diretamente (UVA), a execução de dois *kernels* em separado não faz mais sentido, visto que não existirão dados de borda a serem enviados a nenhum processo. Logo a chamada de dois *kernels* (versão 3) contra apenas um (versão 2) se torna desnecessária e representa um custo adicional na computação, tornando-a mais lenta. Porém, o mesmo não acontece na configuração com quatro GPUs, onde a versão 3 se mostra mais rápida que a versão 2 para todas as dimensões de malha, exceto para a de $50 \times 50 \times 50$ pontos. A melhoria foi justamente pelo fato de que agora, para quatro GPUs, existem dados de borda a serem enviados a outros processos, e a computação poderá sobrepor-se com a comunicação utilizando a técnica empregada na versão 3 do código.

A malha de $50 \times 50 \times 50$ pontos não tirou proveito com tal abordagem pois a quantidade de pontos a ser calculada por cada GPU é muito pequena, fazendo com que a utilização de dois *kernels* represente um custo adicional na execução da aplicação.

Tabela 6.4: *Speedups* obtidos no ambiente de execução 1 pela terceira versão com múltiplas GPUs em relação ao código sequencial

Malha	**Versão**	**Tempo médio (s)**	**Desvio padrão**	***Speedup***
50x50x50	sequencial	2.016,5	0,1 %	-
	1 GPU	16,4	0 %	123,0
	2 GPUs	9,2	0,1 %	**220,3**
	4 GPUs	12,5	2,0 %	161,2
	8 GPUs	21,3	1,6 %	94,7
100x100x100	sequencial	16.680,1	0,6 %	-
	1 GPU	122,7	0 %	135,9
	2 GPUs	62,7	0 %	265,9
	4 GPUs	38,7	1,0 %	**430,5**
	8 GPUs	70,1	0,5 %	238,1
150x150x150	sequencial	49.638,6	1,1 %	-
	1 GPU	420,6	0 %	118,0
	2 GPUs	215,1	0 %	230,7
	4 GPUs	110,0	0,9 %	**451,3**
	8 GPUs	148,8	0,4 %	333,6
200x200x200	sequencial	110.675,4	0,7 %	-
	1 GPU	975,8	0 %	113,4
	2 GPUs	495,2	0 %	223,5
	4 GPUs	247,3	0 %	**447,5**
	8 GPUs	258,4	0,6 %	428,3
250x250x250	sequencial	208.005,9	0,9 %	-
	1 GPU	1.964,6	0 %	105,9
	2 GPUs	995,2	0 %	209,0
	4 GPUs	503,7	0,2 %	412,9
	8 GPUs	400,7	0,6 %	**519,1**

Tabela 6.5: *Speedups* obtidos no ambiente de execução 2 pela terceira versão com múltiplas GPUs em relação ao código sequencial

Malha	**Versão**	**Tempo médio (s)**	**Desvio padrão**	***Speedup***
50x50x50	sequencial	2.016,5	0,1 %	-
	1 GPU	7,1	0,4 %	284,0
	2 GPUs	4,8	0,2 %	**420,5**
	4 GPUs	13,0	0,4 %	154,5
100x100x100	sequencial	16.680,1	0,6 %	-
	1 GPU	53,5	0 %	311,8
	2 GPUs	29,2	0,1 %	**570,5**
	4 GPUs	46,8	0,1 %	356,3
150x150x150	sequencial	49.638,6	1,1 %	-
	1 GPU	178,1	0 %	278,7
	2 GPUs	95,6	0 %	**519,4**
	4 GPUs	116,7	0,3 %	425,4
200x200x200	sequencial	110.675,4	0,7 %	-
	1 GPU	416,6	0 %	265,7
	2 GPUs	221,1	0 %	**500,5**
	4 GPUs	229,3	0,1 %	482,7
250x250x250	sequencial	208.005,9	0,9 %	-
	1 GPU	830,6	0 %	250,4
	2 GPUs	438,1	0 %	474,8
	4 GPUs	404,8	0,2 %	**513,9**

A Tabela 6.6 resume os ganhos obtidos pela versão 3 em relação às versões anteriores 1 e 2, considerando na comparação as configurações que obtiveram os melhores *speedups*.

Tabela 6.6: Ganhos obtidos pela versão 3 no ambiente de execução 2 em relação aos maiores *speedups* alcançados pelas primeira e segunda versões

Malha	**Versão**	**Número de GPUs**	**Ganho da Versão 3**
50x50x50	Versão 1	2 GPUs	11,1 %
	Versão 2	2 GPUs	-37,1 %
100x100x100	Versão 1	2 GPUs	7,9 %
	Versão 2	2 GPUs	-14,1 %
150x150x150	Versão 1	2 GPUs	5,2 %
	Versão 2	2 GPUs	-12,3 %
200x200x200	Versão 1	2 GPUs	3,5 %
	Versão 2	2 GPUs	-12,3 %
250x250x250	Versão 1	4 GPUs	5,7 %
	Versão 2	2 GPUs	-2,3 %

Pode-se observar, na Tabela 6.6, que a perda de desempenho da versão 3 em relação à versão 2 reduz de modo significativo à medida que o tamanho da malha aumenta, indicando uma tendência de que, para malhas maiores, pudesse ocorrer uma completa inversão nos resultados, ou seja, que a versão 3 apresentasse ganhos de desempenho em relação à versão 2. Para confirmar ou refutar tal hipótese, foram realizados testes adicionais com malhas ainda maiores. Os resultados obtidos revelam que neste cenário a versão 3 de fato consegue obter melhores resultados que a versão 2, conforme pode ser observado na Tabela 6.7.

Os resultados apresentados na Tabela 6.7 para a versão 2 foram obtidos no ambiente de execução 2, enquanto que os da terceira versão foram avaliados em ambos os ambientes 1 e 2, sendo referidos na Tabela 6.7 respectivamente por A_1 e A_2.

Com uma configuração de malha de $300 \times 300 \times 300$, o maior *speedup* obtido pela segunda versão do código passa agora a ser alcançado com o uso de quatro GPUs e não

Tabela 6.7: Ganhos obtidos pela versão 3 em relação à versão 2 em um novo cenário

Malha	**Versão**		**Tempo médio (s)**	**Ganho da Versão 3**
300x300x300	Versão 2	2 GPUs	677,0	-
		4 GPUs	652,5	-
	Versão 3	8 GPUs (A_1)	569,3	12,8 %
		4 GPUs (A_2)	638,9	2,1 %
400x400x400	Versão 2	2 GPUs	-	-
		4 GPUs	1.364,4	-
	Versão 3	8 GPUs (A_1)	1.010,0	26,0 %
		4 GPUs (A_2)	1.345,9	1,4 %

de duas, fato que não ocorria para as malhas utilizadas nos experimentos anteriores, onde a configuração com duas GPUs era sempre a que apresentava os maiores ganhos. Na simulação da malha de $300 \times 300 \times 300$ pontos, a versão 2 com quatro GPUs terminou sua execução após 652,5 segundos, já a versão 3 com oito GPUs do primeiro ambiente gastou apenas 569,3 segundos para realizar a mesma simulação, alcançando um ganho de 12,8 % sobre sua versão anterior. Conforme já destacado, a versão 3 conseguiu melhores resultados que a versão 2 para as duas configurações de malhas, quando ambas as versões fazem o uso de quatro GPUs do segundo ambiente. Tal fato comprova que a versão 3 é a melhor dentre todas para lidar com grandes dimensões de malha.

Deve-se ainda ressaltar que a configuração de malha $400 \times 400 \times 400$, para o caso da versão 2 com duas GPUs, não pode ser executada pois não existiam recursos suficientes, como memória e/ou número de *threads* CUDA, para lidar com tamanha quantidade de pontos a serem processados. O maior ganho da versão 3 sobre a versão 2 foi alcançada justamente com o uso de tal tamanho de malha, chegando a expressivos 26,0 % de ganho.

Os dados apresentados na Tabela 6.7 reforçam ainda mais a idéia já discutida anteriormente de que para maiores tamanhos de malha, o uso de mais GPUs é justificado, já que a quantidade de dados a serem processados por cada uma (ou seja, o tempo de computação) acaba compensando o custo da comunicação envolvida.

6.7 Tempo Teórico Mínimo

Embora as acelerações apresentadas até aqui tenham se mostrado expressivas, os ganhos obtidos pelo uso de múltiplas GPUs poderiam, teoricamente, ser ainda maiores caso não fosse o *overhead* causado principalmente pela transferência de dados entre as GPUs participantes da computação, apesar de outros aspectos, como a sincronização entre processos, a sincronização entre a CPU e a GPU, dentre outros, também serem responsáveis pela redução do desempenho final obtido.

Para determinar o tempo mínimo de execução que a aplicação necessitaria para ser executada nas múltiplas GPUs, a terceira versão implementada foi comparada com uma outra versão desenvolvida neste trabalho, que faz o uso de uma única GPU para o cálculo das simulações do modelo do SIH inato. Nesta versão da aplicação não existe necessidade alguma de enviar dados a outra GPU ou realizar a computação dos pontos de borda de forma separada daqueles mais internos, pois o cálculo de toda a malha será de sua responsabilidade, logo tal versão não faz uso do recurso UVA e utiliza somente um *kernel* para processar todos os pontos da malha.

Como já abordado neste trabalho, o esquema de divisão do espaço discretizado entre N GPUs que participam da computação é realizado de tal maneira que cada uma receba uma fração do tecido de tamanho igual a $((X + N - 1)/N,\ Y,\ Z)$. Portanto, ao se executar uma simulação com o número de GPUs igual a oito e dimensões de malha de $240 \times 240 \times 240$ pontos por exemplo, seria teoricamente esperado que o tempo total da simulação fosse equivalente àquele obtido em uma execução da versão do código que utiliza apenas uma GPU e processa uma malha de $30 \times 240 \times 240$ pontos. Porém, devido aos *overheads*, como o alto custo de comunicação entre os processos, esse tempo ideal de execução dificilmente será alcançado.

Com o intuito de definir a eficiência com que a aplicação desenvolvida neste trabalho executa o modelo do SIH inato, foram realizadas algumas simulações para diferentes tamanhos de malhas e número de GPUs utilizadas. Valores predeterminados para as dimensões da malha foram escolhidos de modo que a fração da mesma seja realizada igualitariamente entre todas as N GPUs, tornando a comparação mais justa, pois para valores de dimensão X não múltiplos de N, as fatias que cada GPU computa seriam de tamanhos distintos.

As Tabelas 6.8 e 6.9 apresentam os tempos obtidos nos dois ambientes de execução

pela terceira versão, que utiliza múltiplas GPUs, como também pela versão com uma única GPU, que será usada para se calcular o tempo mínimo de computação para cada configuração de malha. A coluna chamada eficiência explicita a relação entre a versão 3 e aquela que faz uso de apenas uma GPU. Seu cálculo é apresentado a seguir pela Equação 6.2:

$$E(p) = \frac{t_e}{t_o}, \tag{6.2}$$

onde t_e é o tempo de execução esperado, ou seja, aquele obtido com o uso de apenas uma GPU, e t_o é o tempo de execução obtido com o uso de múltiplas GPUs. O valor final indica o quão rápida a terceira versão está em relação ao que poderia ser, sendo que o valor ideal (100 %) seria alcançado quando não existissem *overheads* para a execução das versões que empregam múltiplas GPUs, como comunicação e sincronização. Cada configuração de dimensões da malha e número de GPUs foi executada três vezes para garantir a confiabilidade dos tempos colhidos: o desvio padrão ficou abaixo de 2,2 % para todos os cenários testados.

Ao fazer uma análise sobre os resultados obtidos pelos experimentos no ambiente de execução 1 (Tabela 6.8), pode-se notar que a versão 3 obtém em alguns cenários uma boa eficiência. Em todas as configurações de malha, os melhores resultados são obtidos com o uso de duas ou quatro GPUs apenas, visto que para duas GPUs não haverá comunicação MPI e com quatro GPUs parte da comunicação existente, somente entre duas máquinas, foi sobreposta com computação pelo emprego de duas *streams* CUDA.

Novamente, para malhas pequenas e número maior de GPUs a eficiência alcançada é baixa, pois o tamanho do espaço a ser calculado por cada uma é pequeno e a comunicação entre os processos se torna mais predominante. Porém, a medida que as dimensões do problema vão aumentando, a eficiência vai se tornando cada vez maior.

A mesma conclusão discutida pode ser aplicada aos resultados apresentados na Tabela 6.9, contudo, os ganhos obtidos nas simulações avaliadas aqui (segundo ambiente de execução) são menores em relação àqueles alcançados no primeiro ambiente. Isso se dá por efeito da arquitetura das GPUs NVIDIA Tesla M2050 do segundo ambiente, que possuem mais unidades de processamento (CUDA *cores*) em relação àquelas encontradas no primeiro (448 contra 240). Logo a computação de seus dados é realizada mais rapidamente. Contudo, a transferência dos mesmos é realizada no mesmo tempo, de modo que proporcionalmente a comunicação acaba ocupando uma maior porção do tempo total

Tabela 6.8: Comparação entre os tempos colhidos e os teoricamente esperados pelo uso de múltiplas GPUs no ambiente 1

Malha	**Número de GPUs**	**Tempo obtido (s)**	**Tempo esperado (s)**	**Eficiência**
80x80x80	2 GPUs	31,9	23,9	74,9 %
	4 GPUs	26,2	12,1	46,2 %
	8 GPUs	47,5	6,5	13,6 %
160x160x160	2 GPUs	253,3	187,9	74,2 %
	4 GPUs	127,0	95,0	74,8 %
	8 GPUs	168,3	47,6	28,3 %
240x240x240	2 GPUs	847,2	615,8	72,7 %
	4 GPUs	423,4	308,1	72,8 %
	8 GPUs	367,1	154,2	42,0 %
320x320x320	2 GPUs	1.996,4	1.456,0	72,9 %
	4 GPUs	997,5	721,2	72,3 %
	8 GPUs	653,4	364,7	55,8 %

Tabela 6.9: Comparação entre os tempos colhidos e os teoricamente esperados pelo uso de múltiplas GPUs no ambiente 2

Malha	**Número de GPUs**	**Tempo obtido (s)**	**Tempo esperado (s)**	**Eficiência**
80x80x80	2 GPUs	15,4	8,8	57,1 %
	4 GPUs	29,9	4,7	15,6 %
160x160x160	2 GPUs	116,2	72,1	62,1 %
	4 GPUs	135,4	36,4	26,9 %
240x240x240	2 GPUs	382,6	239,6	62,6 %
	4 GPUs	366,6	120,0	32,7 %
320x320x320	2 GPUs	904,3	552,4	61,1 %
	4 GPUs	767,9	275,5	35,9 %

de execução.

Por fim, mesmo com os obstáculos impostos pela comunicação e sincronização entre as GPUs, a abordagem utilizada neste trabalho produziu resultados bastante significativos, podendo-se dizer que nos melhores cenários obteve-se 75 % de eficiência máxima quando a versão com múltiplas GPUs foi executada no primeiro ambiente (Tabela 6.8), e cerca de 63 % de eficiência quando executada no segundo ambiente (Tabela 6.9).

7 CONCLUSÃO

Neste trabalho foi desenvolvido um novo modelo matemático-computacional que simula a dinâmica de alguns dos principais componentes do sistema imunológico humano (SIH) inato como neutrófilos, macrófagos ativados, macrófagos *resting*, neutrófilos apoptóticos, citocina pró-inflamatória, citocina anti-inflamatória e grânulos proteicos durante uma resposta imune inata ao lipopolissacarídeo (LPS) em uma seção de tecido em três dimensões. Através de contribuições anteriores (Pigozzo [2] e Rocha [3]), o modelo criado neste trabalho oferece uma visão mais detalhada e completa do SIH inato, auxiliando no melhor entendimento da relação existente entre todas as suas células e moléculas simuladas neste.

Pôde-se notar no novo modelo criado, que a ação dos grânulos proteicos no local contribuem para o recrutamento de mais monócitos ao tecido infectado, e a concentração de citocinas pró-inflamatórias influencia na permeabilidade endotelial, permitindo a chegada de mais neutrófilos e monócitos ao local, intensificando a resposta imune inata. As citocinas anti-inflamatórias também desempenham funções importantes: à medida que o LPS vai sendo eliminado, tais substâncias impedem a ativação de macrófagos que estejam no estado de repouso, como também inibem a produção de citocinas pró-inflamatórias realizada pelos neutrófilos e macrófagos ativados. A existência de neutrófilos apoptóticos no tecido exerce grande influência na produção de citocinas anti-inflamatórias realizada pelos macrófagos, pois quanto maior é a sua concentração no local, maiores são as evidências de que o LPS está sendo destruído. Logo, a ação das citocinas anti-inflamatórias é crucial na regulação da resposta inflamatória, pois evita que o processo de inflamação persista mesmo após a completa eliminação do LPS no organismo.

O uso de múltiplas GPUs (*Graphical Processing Units*) em um ambiente de agregados computacionais contribuiu significativamente na redução no tempo necessário para a execução de tal modelo. Devido ao alto custo de comunicação imposto pelo uso de múltiplos dispositivos, o emprego de técnicas alternativas às transferências de dados entre eles foi realizado neste trabalho. No total, três abordagens distintas foram criadas para lidar com este obstáculo: a primeira delas realiza as transferências de dados entre distintas GPUs usando a memória principal como espaço intermediário de armazenamento; já a

segunda versão utiliza o recurso UVA (*Unified Virtual Addressing*) para que as GPUs acessem seus dados diretamente e, por fim, a versão mais elaborada tenta sobrepor as transferências de dados com computação, empregando *streams* CUDA (*Compute Unified Device Architecture*) para este propósito.

Apesar dos ótimos ganhos em aceleração obtidos por cada versão implementada neste trabalho, a comunicação ainda é considerada uma das principais responsáveis pela aplicação não ter alcançado acelerações ainda melhores.

É neste contexto desafiador que em trabalhos futuros outras técnicas de paralelismo podem ser empregadas na tentativa de reduzir o custo da comunicações entre as GPUs. Também torna-se viável o uso de outras estratégias de divisão de trabalho que diferem daquela utilizada neste trabalho, como por exemplo, o balanceamentos de carga entre CPU (*Central Processing Unit*) e GPU: as submalhas a serem processadas também podem ser calculadas pelos processadores disponíveis na CPU, e não apenas pelas GPUs, como no esquema adotado neste trabalho. O balanceamento de carga entre GPUs de diferentes configurações também é possível, logo, para uma divisão mais justa do tecido entre as GPUs, deverão ser levadas em consideração as características desses dispositivos, em especial a quantidade de núcleos CUDA disponíveis, acesso direto aos dados (UVA) e quantidade de memória. Com isso, as fatias de tecido designados à diferentes GPUs não terão o mesmo tamanho, logo as GPUs com maior poder de processamento serão responsáveis pelo cálculo de submalhas com dimensões maiores que aquelas designadas às GPUs com menor poder computacional.

Uma outra alternativa seria utilizar uma infraestrutura de rede de comunicação mais moderna que ofereça uma latência mais baixa na comunicação entre os processos, como por exemplo a rede *InfiniBand*.

REFERÊNCIAS

[1] SHUI, W., "Proteomics Study Yields Clues as to How Tuberculosis Might be Thwarting the Immune System", Disponível em: http://newscenter.lbl.gov/news-releases/2008/11/05/proteomics-study-yields-clues-as-to-how-tuberculosis-might-be-thwarting-the-immune-system/. Acesso em: 02 de julho de 2013, Novembro 2008.

[2] PIGOZZO, A. B., *Implementação Computacional de um Modelo Matemático do Sistema Imune Inato*, Master's Thesis, Universidade Federal de Juiz de Fora, 2011.

[3] ROCHA, P. A. F., *Emprego de GPGPUs para Acelerar Simulações do Sistema Humano Inato*, Master's Thesis, Universidade Federal de Juiz de Fora, 2012.

[4] SOMPAYRAC, L. M., *How the Immune System Works*. 3rd ed. Wiley, John & Sons, Incorporated, 2008.

[5] MURPHY, K. M., TRAVERS, P., WALPORT, M., *Imunobiologia*. 7th ed. Garland Science, Novembro 2010.

[6] KINDT, T. J., OSBORNE, B. A., GOLDSBY, R. A., *Kuby Immunology*. W. H. Freeman, Julho 2006.

[7] MAYER, G., "Immunology - Chapter One: Innate (non-specific) Immunity", *Microbiology and Immunology*, 2006.

[8] ENG, R. H., SMITH, S. M., FAN-HAVARD, P., OGBARA, T., "Effect of antibiotics on endotoxin release from gram-negative bacteria", *Diagnostic Microbiology and Infectious Disease*, v. 16, n. 3, pp. 185–189, 1993.

[9] MALLICK, A., ISHIZAKA, A., STEPHENS, K., HATHERILL, J., TAZELAAR, H., RAFFIN, T., "Multiple organ damage caused by tumor necrosis factor and prevented by prior neutrophil depletion", *Chest*, v. 95, 1989.

[10] HORGAN, M. J., PALACE, G. P., EVERITT, J. E., MALIK, A. B., "TNF-alpha release in endotoxemia contributes to neutrophil-dependent pulmonary edema",

American Journal of Physiology - Heart and Circulatory Physiology, v. 264, n. 4, pp. H1161–H1165, Abril 1993.

[11] CHERTOV, O., UEDA, H., XU, L., TANI, K., MURPHY, W., WANG, J., HOWARD, O., SAYERS, T., OPPENHEIM, J., "Identification of human neutrophil-derived cathepsin G and azurocidin/CAP37 as chemoattractants for mononuclear cells and neutrophils." *J Exp Med*, v. 186, n. 5, pp. 739–747, Agosto 1997.

[12] CHERTOV, O., MICHIEL, D., XU, L., WANG, J., TANI, K., MURPHY, W., LONGO, D., TAUB, D., OPPENHEIM, J., "Identification of defensin-1, defensin-2, and CAP37/azurocidin as T-cell chemoattractant proteins released from interleukin-8-stimulated neutrophils", *The Journal of biological chemistry*, v. 271, 1996.

[13] YANG, D., CHEN, Q., SCHMIDT, A. P., ANDERSON, G. M., WANG, J. M., WOOTERS, J., OPPENHEIM, J. J., CHERTOV, O., "Ll-37, the Neutrophil Granule- and Epithelial Cell-Derived Cathelicidin, Utilizes Formyl Peptide Receptor-Like 1 (Fprl1) as a Receptor to Chemoattract Human Peripheral Blood Neutrophils, Monocytes, and T Cells", *The Journal of Experimental Medicine*, v. 192, n. 7, pp. 1069–1074, Outubro 2000.

[14] SOEHNLEIN, O., WEBER, C., LINDBOM, L., "Neutrophil granule proteins tune monocytic cell function", *Trends in Immunology*, v. 30, n. 11, pp. 538 – 546, 2009.

[15] SOEHNLEIN, O., ZERNECKE, A., WEBER, C., "Neutrophils launch monocyte extravasation by release of granule proteins", *Thrombosis and Haemostasis*, 2009.

[16] OLD, L. J., "Tumor necrosis factor", *Scientific American*, v. 258, pp. 59–75, Maio 1988.

[17] SAKLATVALA, J., DAVIS, W., GUESDON, F., KARIN, M., MARSHALL, C. J., "Interleukin 1 (IL1) and Tumor Necrosis Factor (TNF) Signal Transduction", *Philosophical Transactions: Biological Sciences*, v. 351, n. 1336, pp. 151–157, Fevereiro 1996.

[18] BAZZONI, F., CASSATELLA, M. A., ROSSI, F., CESKA, M., DEWALD, B., BAGGIOLINI, M., "Phagocytosing neutrophils produce and release high amounts of the neutrophil-activating peptide 1/interleukin 8." *The Journal of Experimental Medicine*, v. 173, n. 3, pp. 771–774, Março 1991.

[19] STRIETER., R. M., KASAHARA., K., ALLEN., R. M., STANDIFORD., T. J., ROLFE., M. W., BECKER., F. S., CHENSUE., S. W., KUNKEL, S. L., "Cytokine-induced neutrophil-derived interleukin-8", *Am J Pathol*, v. 141, pp. 397–407, 1992.

[20] FIORENTINO, D., ZLOTNIK, A., MOSMANN, T., HOWARD, M., O'GARRA, A., "IL-10 inhibits cytokine production by activated macrophages", *The Journal of Immunology*, v. 147, n. 11, pp. 3815–3822, Dezembro 1991.

[21] MOORE, K. W., DE WAAL MALEFYT, R., COFFMAN, R. L., O'GARRA, A., "Interleukin-10 and the interleukin-10 receptor", *Annual Review of Immunology*, v. 19, n. 1, pp. 683–765, 2001.

[22] BAGGIOLINI, M., DEWALD, B., MOSER, B., "Human chemokines: an update", *Annual Review Immunology*, v. 15, pp. 675–705, 1997.

[23] MULLIGAN, M., JONES, M., BOLANOWSKI, M., BAGANOFF, M., DEPPELER, C., MEYERS, D., RYAN, U., WARD, P., "Inhibition of lung inflammatory reactions in rats by an anti-human IL-8 antibody", *The Journal of Immunology*, v. 150, n. 12, pp. 5585–5595, Junho 1993.

[24] SEKIDO, N., MUKAIDA, N., HARADA, A., NAKANISHI, I., WATANABE, Y., MATSUSHIMA, K., "Prevention of lung reperfusion injury in rabbits by a monoclonal antibody against interleukin-8", *Nature*, v. 365, n. 6447, pp. 654–657, Outubro 1993.

[25] BROADDUS, V., BOYLAN, A., HOEFFEL, J., KIM, K., SADICK, M., CHUNTHARAPAI, A., HEBERT, C., "Neutralization of IL-8 inhibits neutrophil influx in a rabbit model of endotoxin-induced pleurisy", *The Journal of Immunology*, v. 152, n. 6, pp. 2960–2967, Março 1994.

[26] MCCORMICK, T. S., STEVENS, S. R., KANG, K., "Macrophages and cutaneous inflammation", *Nat Biotech*, v. 18, n. 1, pp. 25–26, Janeiro 2000.

[27] FUJIWARA, N., KOBAYASHI, K., "Macrophages in inflammation", *Current Drug Targets - Inflammation & Allergy*, v. 4, n. 3, pp. 281–286, Junho 2005.

[28] SU, B., ZHOU, W., DORMAN, K. S., JONES, D. E., "Mathematical modelling of immune response in tissues", *Computational and Mathematical Methods in Medicine: An Interdisciplinary Journal of Mathematical, Theoretical and Clinical Aspects of Medicine*, v. 10, pp. 1748–6718, 2009.

[29] BORREGAARD, N., COWLAND, J. B., "Granules of the Human Neutrophilic Polymorphonuclear Leukocyte", *Blood*, v. 10, pp. 3503–3521, 1997.

[30] GOUTELLE, S., MAURIN, M., ROUGIER, F., BARBAUT, X., BOURGUIGNON, L., DUCHER, M., MAIRE, P., "The Hill equation: a review of its capabilities in pharmacological modelling", *Fundamental & clinical pharmacology*, v. 22, n. 6, pp. 633–648, Dezembro 2008.

[31] PIGOZZO, A. B., MACEDO, G. C., DOS SANTOS, R. W., LOBOSCO, M., "On the Computational Modeling of the Innate Immune System", *BMC BioInformatics*, v. 14, pp. S7, Abril 2013.

[32] ROCHA, P. A. F., XAVIER, M. P., PIGOZZO, A. B., QUINTELA, B. M., MACEDO, G. C., DOS SANTOS, R. W., LOBOSCO, M., "A Three-Dimensional Computational Model of the Innate Immune System", In: *Computational Science and Its Applications ICCSA 2012*, v. 7333, pp. 691–706, *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2012.

[33] FLYNN, M. J., "Some Computer Organizations and Their effectiveness", *IEEE Transactions on Computers*, v. 21, pp. 948–960, Setembro 1972.

[34] KIRK, D. B., MEI W. HWU, W., *Programming Massively Parallel Processors: A Hands-on Approach*. 1st ed. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2010.

[35] NVIDIA, *CUDA C Best Practices Guide*. Technical Report, NVIDIA Corporation, Outubro 2012.

[36] COOK, S., *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. 1st ed. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2012.

[37] CHANDRA, R., DAGUM, L., KOHR, D., MAYDAN, D., MACDONALD, J., MENON, R., *Parallel Programming in OpenMP*. 1st ed. Morgan Kaufmann Publishers, 2001.

[38] PACHECO, P. S., *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1996.

[39] "MPICH - High-Performance Portable MPI", Disponível em: http://www.mpich.org/. Acesso em: 16 de julho de 2013, 2013.

[40] LEVEQUE, R. J., *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics, 2007.

[41] FELDER, S., KAM, Z., "Human neutrophil motility: Time-dependent three-dimensional shape and granule diffusion", *Cell Motility and the Cytoskeleton*, v. 28, n. 4, pp. 285–302, 1994.

[42] CHETTIBI, S., LAWRENCE, A., YOUNG, J., LAWRENCE, P., STEVENSON, R., "Dispersive locomotion of human neutrophils in response to a steroid-induced factor from monocytes", *J Cell Sci*, v. 107, n. 11, pp. 3173–3181, Novembro 1994.

APÊNDICE A - PARÂMETROS UTILIZADOS NO MODELO

Tabela A.1: Condições Iniciais do Modelo

Parâmetro	**Valor**	**Unidade**	**Referência**
LPS_0	$100 : 0.8 * Z < z < Z$	$10^4 moléculas/mm^3$	[31, 32]
LPS_0	$0 : 0 \leq z \leq 0.8 * Z$	$10^4 moléculas/mm^3$	[31, 32]
MR_0	1	$10^4 células/mm^3$	[31]
MA_0	0	$10^4 células/mm^3$	[31]
N_0	0	$10^4 células/mm^3$	[31]
ND_0	0	$10^4 células/mm^3$	[31]
CH_0	0	$10^4 moléculas/mm^3$	[31]
G_0	0	$10^4 moléculas/mm^3$	[31]
CA_0	0	$10^4 moléculas/mm^3$	[31]

Tabela A.2: Parâmetros do Modelo

Parâmetro	**Valor**	**Unidade**	**Referência**
θ_{CA}	1	$1/(células/mm^3)$	[2]
$\phi_{MR\|LPS}$	0.1	$1/(células/mm^3)$.dia	[31]
$\lambda_{N\|LPS}$ e $\lambda_{LPS\|N}$	0.55	$1/(células/mm^3)$.dia	[28]
$\lambda_{MA\|LPS}$ e $\lambda_{ND\|MA}$	0.8 e 2.6	$1/(células/mm^3)$.dia	[28]
$\alpha_{CA\|MA}$	1.5	adimensional	[31]
$\beta_{CH\|MA}$ e $\beta_{CH\|N}$	0.8 e 1	$1/(células/mm^3)$.dia	[2]
$\beta_{MR\|ND}$	1.5	$1/(células/mm^3)$.dia	[31]
$\alpha_{G\|N}$	0.6	adimensional	[31]
μ_{LPS}	0.005	1/dia	[28]
μ_{MR} e μ_{MA}	0.033 e 0.07	1/dia	[28]
μ_N e μ_{CH}	3.43 e 7	1/dia	[2]
μ_G e μ_{CA}	5 e 4	1/dia	[2]
P_N^{max}	11.4	1/dia	[31]
P_N^{min}	0.0001	1/dia	[2]
P_{MR}^{min} e P_{MR}^{max}	0.01 e 0.1	1/dia	[2]
$P_{MR\_g}^{min}$ e $P_{MR\_g}^{max}$	0 e 0.5	1/dia	[2]
M^{max} e N^{max}	6 e 8	$células/mm^3$	[2]
D_{LPS}	2000	μm^2/dia	[31]
D_{MA} e D_{MR}	3000 e 4320	μm^2/dia	[31]
D_N	12096	μm^2/dia	[41]
D_{ND}	0.144	μm^2/dia	[28]
D_{CH}	9216	μm^2/dia	[28]
D_G e D_{CA}	9216	μm^2/dia	[2]
$keqch$ e $keqg$	1	$células/mm^3$	[2]
$caInf$ e $chInf$	3.6	$células/mm^3$	[31]
$gInf$	3.1	$células/mm^3$	[2]
χ_N	14400	μm^2/dia	[42]
χ_{MR}	3600	μm^2/dia	[2]
χ_{MA}	4320	μm^2/dia	[2]