Universidade Federal do Pampa

Matheus da Silva Serpa

Análise de Desempenho de Aplicações Paralelas em Arquiteturas *multi-core* e *many-core*

Alegrete

2015

Matheus da Silva Serpa

Análise de Desempenho de Aplicações Paralelas em Arquiteturas *multi-core* e *many-core*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Claudio Schepke

Alegrete
2015

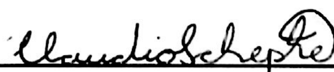
Matheus da Silva Serpa

Análise de Desempenho de Aplicações Paralelas em Arquiteturas *multi-core* e *many-core*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho de Conclusão de Curso defendido e aprovado em 30 de Novembro de 2015.

Banca examinadora:



Claudio Schepke
Orientador

Arthur Francisco Lorenzon
Universidade Federal do Rio Grande do Sul

Márcia Cristina Cera
Universidade Federal do Pampa

Dedico este trabalho à Deus, aos meus pais e amigos.

Success is the ability to move from
one failure to another without loss
of enthusiasm.
(Winston Churchill)

Resumo

Simulações numéricas auxiliam no avanço da Ciência e exigem cada vez mais recursos computacionais para sua previsão. Quanto maior a resolução de um modelo, mais preciso, exato e custoso serão suas previsões. Os supercomputadores atuais são sistemas heterogêneos compostos por arquiteturas *multi-core* e *many-core*. Com isso, pesquisadores podem utilizar ambos processadores e aceleradores para simulação de problemas com resoluções maiores. Todavia, desenvolver aplicações para esses sistemas é um desafio devido a complexidade da programação. Nesse contexto, esse trabalho analisou o desempenho de aplicações paralelas em sistemas heterogêneos, executando-as em processadores com *Open Multi-Processing* (OpenMP) e em aceleradores com *Computed Unifed Device Architecture* (CUDA). Para isso, desenvolvemos sete aplicações científicas em ambas bibliotecas. Nossos experimentos mostram que para aplicações com pouca demanda de comunicação, *Graphics Processing Units* (GPUs) tem desempenho até 8 vezes maior que *Central Processing Units* (CPUs) executando com 32 *threads*.

Palavras-chave: Aplicações Paralelas. Arquiteturas Heterogêneas. Computação de Alto Desempenho.

Abstract

Numerical simulations assists science advances and require ever more computing resources to their prediction. Higher resolution models will be more accurate and computationally expensive. Current supercomputers are formed by heterogeneous architectures that are divided in *multi-core* and *many-core*. Thus, developers can use both processors and accelerators for higher resolution simulations. However, developing applications for these systems is a challenge due hard programming. In this context, this study analyzed the performance of parallel applications on heterogeneous systems, by running them in processors with OpenMP and GPUs with CUDA. Therefore, seven scientific applications are developed in both libraries. Our experiments show that applications with low communication demand has better performance in GPUs, 8 times in comparasion with CPUs running 32 threads.

Key-words: Parallel Programming. Heterogeneous Computer Architectures. Multi-Core. Many-Core. High Performance Computing.

Lista de ilustrações

Figura 1 – Diagramas das classes *Single Instruction Multiple Data* (SIMD) (esquerda) e *Multiple Instruction Multiple Data* (MIMD) (direita). 24
Figura 2 – Projeto das arquiteturas *multi-core* (esquerda) e *many-core* (direita). . 25
Figura 3 – Arquitetura de uma GPU preparada para CUDA. 26
Figura 4 – Exemplo da soma de dois vetores utilizando OpenMP. 28
Figura 5 – Exemplo da soma de dois vetores utilizando CUDA. 30
Figura 6 – Método das Diferenças Finitas (MDF) aplicado ao problema da Transferência de Calor. 34
Figura 7 – Difusão do calor em OpenMP. 34
Figura 8 – Difusão do calor em CUDA. 35
Figura 9 – Jacobi em OpenMP. 36
Figura 10 – Jacobi em CUDA. 37
Figura 11 – Modelo de *Lattice* D3Q19. 38
Figura 12 – Laço principal do Método de Lattice Boltzmann (MLB). 38
Figura 13 – Lattice Boltzmann em OpenMP. 42
Figura 14 – Lattice Boltzmann em CUDA. 43
Figura 15 – Ordenação Par-Ímpar em OpenMP. 44
Figura 16 – Ordenação Par-Ímpar em CUDA. 45
Figura 17 – Quadratura de Gauss em OpenMP. 45
Figura 18 – Quadratura de Gauss em CUDA. 46
Figura 19 – Regra dos Trapézios em OpenMP. 46
Figura 20 – Regra dos Trapézios em CUDA. 47
Figura 21 – Speedup da Difusão de Calor em Processadores (OpenMP) 52
Figura 22 – Speedup do Método de Jacobi em Processadores (OpenMP) 52
Figura 23 – Speedup do Método de Lattice Boltzmann em Processadores (OpenMP) 53
Figura 24 – Speedup da Ordenação Par-Ímpar em Processadores (OpenMP) 53
Figura 25 – Speedup da Quadratura de Gauss em Processadores (OpenMP) 54
Figura 26 – Speedup da Regra dos Trapézios em Processadores (OpenMP) 54
Figura 27 – Speedup da Difusão de Calor em Placas Gráficas (CUDA) 55
Figura 28 – Speedup do Método de Jacobi em Placas Gráficas (CUDA) 55
Figura 29 – Speedup do Método de Lattice Boltzmann em Placas Gráficas (CUDA) 56
Figura 30 – Speedup da Ordenação Par-Ímpar em Placas Gráficas (CUDA) 56
Figura 31 – Speedup da Quadratura de Gauss em Placas Gráficas (CUDA) 57
Figura 32 – Speedup da Regra dos Trapézios em Placas Gráficas (CUDA) 57
Figura 33 – Comparação entre Processador e Placa Gráfica - Difusão de Calor . . . 58
Figura 34 – Speedup do Método de Jacobi 58

Figura 35 – Speedup do Método de Lattice Boltzmann 59
Figura 36 – Comparação entre Processador e Placa Gráfica - Ordenação Par-Ímpar 59
Figura 37 – Comparação entre Processador e Placa Gráfica - Quadratura de Gauss 60
Figura 38 – Comparação entre Processador e Placa Gráfica - Regra dos Trapézios . 60
Figura 39 – Speedup do conjunto de aplicações em Processador e Placa Gráfica . . 61

Lista de tabelas

Tabela 1 – Qualificadores de tipo de variável CUDA. 29
Tabela 2 – Ambiente de Execução. 40
Tabela 3 – Tipo e Tamanho de Entrada de cada Aplicação 49
Tabela 4 – Tipo de dado e Tempo de Execução Sequencial de cada Aplicação . . . 49

Lista de siglas

IPP Interface de Programação Paralela

CPU *Central Processing Unit*

CUDA *Computed Unifed Device Architecture*

DFC Dinâmica dos Fluidos Computacional

DRAM *Dynamic Random-Access Memory*

GB *Gigabyte*

GDDR *Graphics Double Data Rate*

GPU *Graphic Processing Unit*

CAD Computação de Alto Desempenho

MDF Método das Diferenças Finitas

MIMD *Multiple Instruction Multiple Data*

MLB Método de Lattice Boltzmann

OpenMP *Open Multi-Processing*

Xeon Phi Intel® Xeon Phi™

SIMD *Single Instruction Multiple Data*

SM *Streaming Multiprocessor*

SP *Streaming Processor*

TDP *Thermal Design Power*

ULA Unidade Lógica e Aritmética

Sumário

1 INTRODUÇÃO 21
1.1 Objetivos e Contribuições 22
1.2 Organização do documento 22

2 COMPUTAÇÃO DE ALTO DESEMPENHO 23
2.1 Arquiteturas Paralelas 23
2.1.1 Classificação de Flynn 23
2.1.2 Arquiteturas Heterogêneas 24
2.1.2.1 Unidades de processamento gráfico da NVIDIA 25
2.2 Interfaces de Programação Paralela 27
2.2.1 Programação em Memória Compartilhada 27
2.2.2 Programação Heterogênea 28
2.3 Trabalhos Relacionados 30

3 METODOLOGIA 33
3.1 Aplicações 33
3.1.1 Difusão de Calor 33
3.1.2 Jacobi 35
3.1.3 Lattice Boltzmann 36
3.1.4 Ordenação Par-Ímpar 37
3.1.5 Quadratura de Gauss 39
3.1.6 Regra dos Trapézios 39
3.2 Ambiente de Execução 40
3.3 Métricas Avaliadas 41

4 RESULTADOS 49
4.1 Análise de Desempenho do Processador 49
4.2 Análise de Desempenho da Placa Gráfica 50
4.3 Comparação de Desempenho das Arquiteturas 51
4.4 Considerações sobre os Experimentos 51

5 CONCLUSÕES 63

REFERÊNCIAS 65

Índice 69

1 Introdução

Simulações numéricas vêm sendo amplamente utilizadas com a intenção de prever o comportamento de diferentes tipos de fenômenos. Em sua maioria, tais fenômenos compreendem a previsão de ações da natureza, como por exemplo, furacões, climatologia, gerenciamento térmico, dentre outros (BATCHELOR, 2000). No entanto, a precisão e acurácia dos modelos numéricos estão associadas aos recursos computacionais disponíveis, uma vez que, quanto mais robusto o sistema computacional, maiores as chances do modelo prover uma solução correta em tempo hábil.

O uso de processadores de propósito geral em sistemas de Computação de Alto Desempenho (CAD) tem reduzido o tempo de execução destas simulações durante as últimas décadas e, com isso, permitido a resolução de problemas maiores em tempo aceitável. No entanto, o poder computacional destes processadores é limitado pelo *Thermal Design Power* (TDP), que em alguns casos pode chegar a 150 *Watts*, valor de TDP do Intel Core i7-3970X (INTEL, 2015). Diante disso, arquiteturas heterogêneas tem-se tornado uma alternativa para a execução deste tipo de aplicações (CLARKE et al., 2014; GOYAL; KAUR, 2015).

Arquiteturas heterogêneas são aquelas que integram processadores *multi-core* (CPU) e placas gráficas (*Graphic Processing Unit* (GPUs)) (KYRIAZIS, 2012). O objetivo dessas arquiteturas é o de reduzir a latência da execução de programas utilizando CPUs, GPUs e outros dispositivos de computação.

Enquanto as arquiteturas heterogêneas permitem executar aplicações usando diferentes conjuntos de processadores e aceleradores, o desenvolvimento destas aplicações se torna um desafio para os programadores. O programador deve mapear os trechos de códigos para serem executados na arquitetura que lhe fornece melhor desempenho. Isso é necessário, uma vez que o modelo de programação dessas arquiteturas é totalmente diferente, não sendo possível executar o mesmo código eficientemente em diferentes arquiteturas. Diversos aspectos da arquitetura como conjunto de instruções, hierarquia de *threads* e memória devem ser considerados.

Diversos fatores devem ser levados em consideração na fase de desenvolvimento de uma aplicação paralela. A criação de *threads*, o escalonamento das tarefas e a sincronização são essenciais. Interface de Programação Paralela (IPPs) buscam auxiliar o programador nesse processo, facilitando tarefas como comunicação, divisão de trabalho, exploração de paralelismo e redução de erros. As características das aplicações também são importantes, pois uma aplicação com uso intensivo de operações de ponto flutuante deve apresentar melhor desempenho em GPUs do que em CPUs (SERPA; SCHEPKE;

LIMA, 2015). Outras características como desbalanceamento de carga, dependências, laços encadeados e sincronizações podem dificultar a distribuição de trabalho e, consequentemente, o uso de arquiteturas heterogêneas.

Nesse sentido, como utilizar eficientemente arquiteturas heterogêneas para reduzir o tempo de execução de aplicações científicas?

1.1 Objetivos e Contribuições

O objetivo geral deste trabalho foi o de **desenvolver e analisar o desempenho de aplicações paralelas em arquiteturas heterogêneas**.

A principal contribuição deste trabalho é a otimização de diversas aplicações em arquiteturas heterogêneas compostas por processadores e aceleradores. As demais contribuições deste trabalho são:

- Apresentação de técnicas para o desenvolvimento de aplicações heterogêneas;
- Análise da escalabilidade das aplicações em arquiteturas heterogêneas.

1.2 Organização do documento

O restante deste documento está organizado da seguinte forma. O Capítulo 2 apresenta conceitos básicos sobre arquiteturas paralelas e modelos de programação. A metodologia utilizada neste trabalho, o ambiente de execução e as métricas são apresentadas no Capítulo 3. A análise dos resultados é apresentado no Capítulo 4. Por fim, o Capítulo 5 discute as conclusões deste trabalho.

2 Computação de Alto Desempenho

A Computação de Alto Desempenho (CAD) é um importante tema de pesquisa na Ciência da Computação, pois através do desenvolvimento e execução de algoritmos paralelos é possível solucionar problemas com resoluções maiores em tempo de execução aceitável (FOSTER, 1995; BELL; GRAY, 2002). O objetivo desse capítulo é apresentar conceitos básicos sobre arquiteturas paralelas, modelos de programação e desenvolvimento de aplicações paralelas.

Primeiramente, na seção 2.1 classificamos as arquiteturas paralelas e descrevemos as arquiteturas *multi-core* e *many-core*. Na seção 2.2 descrevemos modelos de programação paralela para computação heterogênea e apresentamos as Interfaces de Programação Paralela utilizadas nesse trabalho.

2.1 Arquiteturas Paralelas

Antes do uso de arquiteturas paralelas, o desempenho dos processadores era ligado diretamente a frequência de operação do processador (e *clock*). No geral, quanto maior a frequência, menor o tempo de *clock*, e maior era o desempenho. No entando, essa abordagem é limitada devido a questões de consumo de energia, dissipação de calor e dimensão do processador.

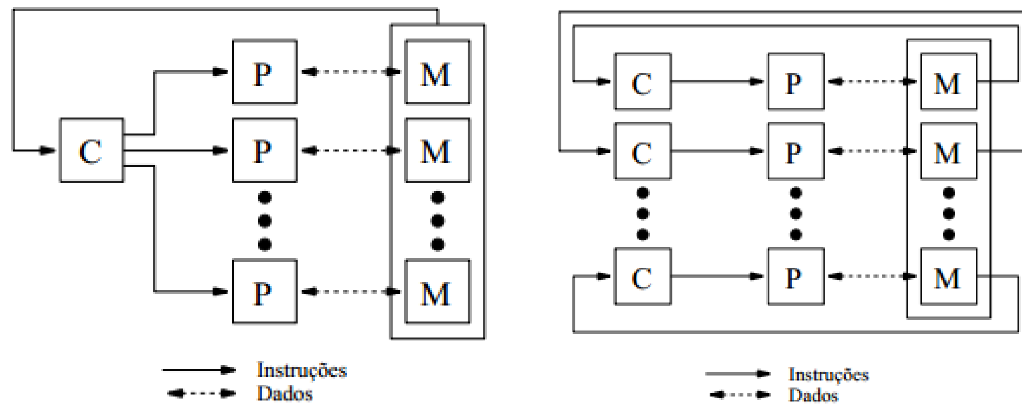
As arquiteturas paralelas *multi-core* e *many-core* têm sido adotadas em seu lugar. A primeira voltada a otimizar a execução de cada instrução (latência) e a segunda ao número de instruções executadas por unidade de tempo (*throughput*) (NAVARRO; HITSCHFELD-KAHLER; MATEU, 2014). O desenvolvimento de *software* foi afetado por essa mudança de paradigma e diversas aplicações sofreram reengenharias para tornar possível o aproveitamento dos recursos através da execução paralela.

2.1.1 Classificação de Flynn

A classificação de Flynn (1972) baseia-se no fato de um computador executar uma sequência de instruções sobre uma sequência de dados. As instruções e os dados são separados em um ou vários fluxos de instruções (*instruction stream*) e um ou vários fluxos de dados (*data stream*). As máquinas paralelas concentram-se em duas classes: *Single Instruction Multiple Data* (SIMD) e *Multiple Instruction Multiple Data* (MIMD) (ROSE; NAVAUX, 2002). A Figura 1 apresenta os diagramas das classes exemplificadas a seguir.

Em uma arquitetura SIMD, uma única instrução é executada ao mesmo tempo sobre múltiplos dados. Esse processamento é controlado por uma única unidade de controle

Figura 1 – Diagramas das classes SIMD (esquerda) e MIMD (direita).



Fonte: Rose e Navaux (2002, p. 19)

que é alimentada por um único fluxo de instruções. Cada instrução é enviada para todos processadores que executam as instruções em paralelo de forma síncrona sobre diferentes fluxos de dados. Essa arquitetura é encontrada nas unidades MMX/SSE dos processadores de uso geral e em aceleradores *many-core* como as *Graphics Processing Units* (GPUs) e os coprocessadores Intel® Xeon Phi™ (Xeon Phi). As máquinas SIMD possuem um único fluxo de instruções, o que significa que apenas um programa está em execução por vez.

Em arquiteturas MIMD, cada unidade de controle recebe um fluxo de instruções próprio e repassa-o para seu respectivo processador. Dessa forma, cada processador executa suas instruções em seus dados de forma assíncrona. O princípio dessa classe é bastante genérico, pois se um computador de um grupo de máquinas for analisado separadamente, este pode ser considerado uma máquina MIMD. Nessa classe encontram-se os processador *multi-core* e os servidores com múltiplos computadores.

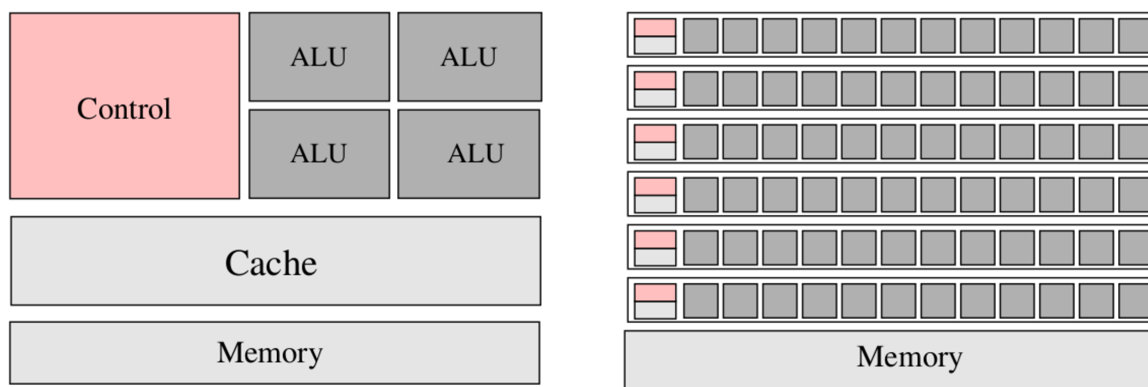
2.1.2 Arquiteturas Heterogêneas

Desde 2003, a indústria vem seguindo duas abordagens para o projeto de microprocessadores (KIRK; WEN-MEI, 2012). A abordagem *multi-core* é orientada à latência, onde instruções são executadas em poucos ciclos de *clock*. Por outro lado, a arquitetura *many-core* tem uma abordagem focada ao *throughput*, ou seja, um grande número de instruções são executadas por unidade de tempo.

O projeto das arquiteturas *multi-core* e *many-core*, ilustrado na Figura 2, é tão diferente que, dependendo da aplicação, o desempenho pode ser muito grande em uma arquitetura e muito pequeno na outra (COOK, 2012). A arquitetura *multi-core* utiliza uma lógica de controle sofisticada para permitir que instruções de uma única *thread* sejam

executadas em paralelo. Grandes memórias *cache* são fornecidas para reduzir latências de acesso às instruções e dados de aplicações que tem acesso à memória predominante. Por fim, as operações das Unidades Lógicas e Aritméticas ULAs também são projetadas visando otimizar a latência.

Figura 2 – Projeto das arquiteturas *multi-core* (esquerda) e *many-core* (direita).



Fonte: Lima (2014, p. 34)

A arquitetura *many-core* tira proveito de um grande número de *threads* de execução. Pequenas memórias *cache* são fornecidas para evitar que múltiplas *threads*, acessando os mesmos dados, precisem ir até a memória principal. Com isso, a maior parte do *chip* é dedicada a unidades de ponto flutuante. Arquiteturas desse tipo são projetadas como mecanismos de cálculo de ponto flutuante e não para operações convencionais, que são realizadas por arquiteturas *multi-core*. A maioria das aplicações usará tanto *multi-core* quanto *many-core* em conjunto, sendo cada arquitetura melhor para um tipo de operação.

2.1.2.1 Unidades de processamento gráfico da NVIDIA

A pesquisa em torno dessa arquitetura cresceu pela demanda do mercado por gráficos de alta qualidade e aplicações de tempo real. Em jogos eletrônicos cenas complexas são renderizadas em uma resolução cada vez maior, a uma taxa de 60 quadros por segundo (KIRK; WEN-MEI, 2012). As GPUs evoluíram tanto que ficavam cada vez mais semelhantes aos computadores paralelos de alto desempenho. Essa evolução fez com que pesquisadores começassem a explorar o uso dessa arquitetura para resolver problemas científicos e de engenharia que requeriam um uso intensivo de operações matemáticas.

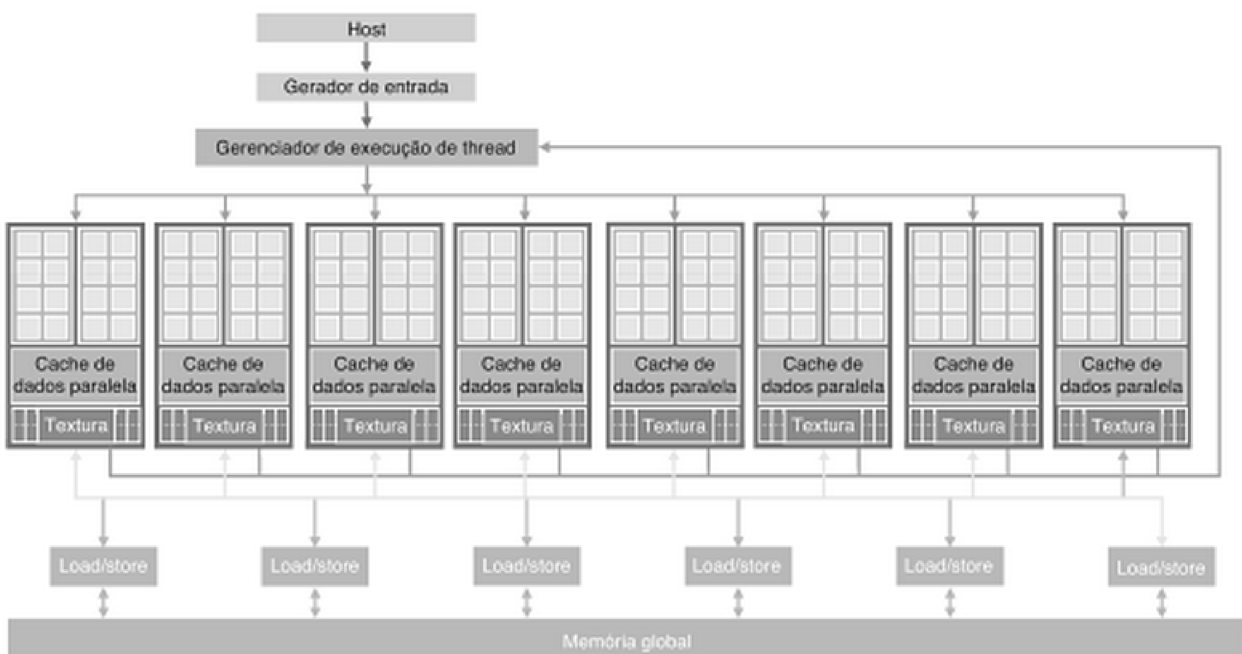
O problema dessa abordagem era a necessidade de transformar um problema científico em um problema tratável por IPPs gráficas como OpenGL ou Direct3D. Tudo mudou em 2007 com o lançamento do *framework* CUDA (NVIDIA, 2007). Isso representou uma mudança no *software* e *hardware* adicional que foi acrescentado em seus chips. Na GPU G80 e suas sucessoras para computação paralela, programas são desenvolvidos

utilizando a IPP de CUDA e não passam mais pela interface gráfica, na qual era necessária a programação gráfica em *shaders*.

A Figura 3 mostra a arquitetura de uma GPU preparada para CUDA. Na figura, cada bloco é formado por dois *Streaming Multiprocessors* (SMs) e cada SM é formado por oito *Streaming Processors* (SPs) que compartilham a lógica de controle e a *cache* de instruções. O número de SMs por bloco e SPs por SM pode variar de uma geração de GPUs CUDA para outra geração. Cada GPU vem com uma memória global DRAM GDDR de até 24 *Gigabytes* (GBs).

Por exemplo, a GPU Tesla C2075 (Fermi) (NVIDIA, 2011) possui 14 SMs, cada um com 32 SPs, totalizando 448 CUDA Cores. Cada SM tem uma *cache* L1 privada e pode executar até 1536 *threads*. Os SMs tem uma *cache* L2 compartilhada de 768 KB, além da memória global de 6 GB. Cada SP tem uma unidade de multiplicação-adição escalar (MAD) e uma unidade de multiplicação para ponto flutuante. Com os 448 SPs, têm-se um total de mais de 515 Gigaflops com precisão dupla e 1.03 Teraflops utilizando precisão simples.

Figura 3 – Arquitetura de uma GPU preparada para CUDA.



Fonte: Kirk e Wen-mei (2012, p. 8)

2.2 Interfaces de Programação Paralela

Essa seção apresenta IPPs para arquiteturas heterogêneas compostas por processadores *multi-core* e aceleradores *many-core*.

2.2.1 Programação em Memória Compartilhada

Uma das formas básicas de explorar o paralelismo é fazer uso da memória compartilhada. Nesse tipo de arquitetura todos processadores podem acessar a memória e comunicar-se através de variáveis compartilhadas. Nesta subseção descrevemos a IPP OpenMP utilizada em memória compartilhada (DIAZ; MUNOZ-CARO; NINO, 2012).

OpenMP consiste em um padrão de programação paralela para arquiteturas de memória compartilhada (CHAPMAN; JOST; PAS, 2008). OpenMP utiliza a diretiva *#pragma*, definida no padrão da linguagem C/C++. O construtor paralelo *#pragma omp parallel* cria uma região paralela mas isso não significa que o trabalho será executado em paralelo. Para isso são necessários construtores como:

- `#pragma omp for`: desencadeia a execução paralela das iterações de laços `for`;
- `#pragma omp sections`: possibilita o compartilhamento de regiões não iterativas. Regiões independentes limitadas pela diretiva `#pragma omp section` são aninhadas dentro de uma região limitada pela diretiva `sections`;
- `#pragma omp task`: define um bloco que será executado em paralelo em forma de paralelismo de tarefas.

A Figura 4 exemplifica o uso da diretiva composta `#pragma omp parallel for` que cria uma região paralela e divide o trabalho entre um número definido de *threads*. OpenMP também especifica cláusulas para definir escopo de variáveis.

A palavra `shared` define quais variáveis são compartilhadas entre as *threads* dentro da região paralelas. Esse compartilhamento implica no mesmo valor da variável e mesmo endereço de memória. No caso de variáveis privadas, utiliza-se a palavra `private` que indica que cada *thread* tem acesso exclusivo a uma cópia dessas variáveis.

Outro operador importante é o de `reduction(operation:var)`. Essa operação cria uma cópia de cada variável para cada *thread*. Ao final da região paralela, a lista de variáveis original é atualizada com os valores da cópia privada de cada *threads* usando um operador especificado. Os operadores podem ser: $+$, $*$, $-$, $\wedge$, &, &&, | e ||.

Diretivas de sincronização garantem que o acesso ou atualização de uma variável compartilhada ocorra no momento certo. Condições de corrida ocorrem quando duas ou mais *threads* tentam atualizar, ao mesmo tempo, uma mesma variável ou quando uma

Figura 4 – Exemplo da soma de dois vetores utilizando OpenMP.

```
1 void vetSoma (int *A, int *B, int *C, int size){
2         int i;
3         #pragma omp parallel for private(i) shared(A, B, C, size)
4         for (i = 0; i < size; i++){
5           C[i] = A[i] + B[i];
6         }
7 }
8
9 int main(){
10     ...
11     vetSoma(A, B, C, 1024*100);
12     ...
13     return 0;
14 }
```

thread atualiza uma variável e outra acessa o valor dessa variável ao mesmo tempo. Dessa forma, essas diretivas garantem que o acesso ou atualização de uma determinada variável aconteça no momento certo. As principais diretivas de sincronização OpenMP são:

- `#pragma omp critical`: restringe a execução de uma determinada tarefa a uma *thread* (do mesmo grupo) por vez.

- `#pragma omp atomic`: impede que várias *threads* acessem essa variável ao mesmo tempo. Esse bloqueio é aplicado a todas as *threads* do programa e não apenas as *threads* do mesmo grupo.

- `#pragma omp barrier`: utilizado para sincronizar todas as *threads* em determinado ponto do código. Alguns construtores como: `parallel`, `for`, `sections` e `single` tem uma barreira implícita ao final da delimitação.

2.2.2 Programação Heterogênea

A programação para aceleradores pode ser feita através de duas abordagens. No modo *offload* o programa executado pelo *host* (processador) chama funções que são executadas no acelerador por um intervalo de tempo (em que o processador pode ficar ocioso ou executar outras funções). Esse é o modelo de execução das GPUs, e pode ser utilizado com coprocessadores. O modo nativo pode ser utilizado apenas por processadores e coprocessadores Xeon Phi. Nesse modo, o programa é executado nativamente pelo acelerador sem influência direta do *host*.

Nessa subseção apresentamos o modelo de programação CUDA, utilizado para programação em arquiteturas heterogêneas.

O modelo de programação CUDA consiste em um conjunto de extensões para C, C++ e FORTRAN que permite a execução em GPUs NVIDIA (SANDERS; KANDROT, 2010). Este modelo de programação é ideal para aplicações com alto nível de paralelismo de dados e para aplicações que não possuem dependência entre tarefas. No entanto, as limitações do modelo incluem o não controle da coerência dos dados utilizados. Assim, os ganhos de desempenho do modelo CUDA dependem de um bom conhecimento por parte do desenvolvedor sobre a arquitetura e o modelo de programação.

Um *kernel* CUDA é executado por um *grid* no qual todas *threads* executam o mesmo código. Essas *threads* são organizadas em uma hierarquia de dois níveis usando coordenadas exclusivas da seguinte forma:

- Um *grid* tem **gridDim** *blocks*. A variável **BlockIdx** varia de 0 a **gridDim - 1**;
- Cada bloco, por sua vez, consiste de **blockDim** *threads*, cada uma com um valor de **threadId** que varia de 0 a **BlockDim - 1**;
- Sendo que cada *grid* tem um total de **gridDim * blockDim** *threads*;
- Uma forma de identificar cada *thread* de um *grid* é determinando seu identificador por: **tid = blockIdx * blockDim + threadIdx**.

A Figura 5 exemplifica a soma de dois vetores utilizando o modelo de programação CUDA. Outro ponto importante é que programador é responsável pela transferência de dados da CPU para GPU e vice-versa. CUDA admite vários tipos de memória que podem ser usados pelo programador. A Tabela 1 mostra os qualificadores de tipo de variável, o tipo de memória, escopo e tempo de vida.

A memória global é a mais lenta mas é a maneira pela qual todas *threads* de um *grid* se comunicam. A memória constante admite acesso de leitura com baixa latência e alta largura de banda quando todas as *threads* acessam simultaneamente o mesmo local. Os registradores e a memória compartilhada são memórias no chip. As variáveis que

Tabela 1 – Qualificadores de tipo de variável CUDA.

Declaração da variável	**Memória**	**Escopo**	**Tempo de vida**
Variáveis automáticas que não sejam matrizes	Registrador	*Thread*	*Kernel*
Variáveis automáticas de matriz	Local	*Thread*	*Kernel*
___device___, ___shared___, int SharedVar;	Compartilhada	*Block*	*Kernel*
___device___, int GlobalVar;	Global	*Grid*	Aplicação
___device___, ___constant___, int ConstVar;	Constante	*Grid*	Aplicação

Fonte: Kirk e Wen-mei (2012, p. 65)

Figura 5 – Exemplo da soma de dois vetores utilizando CUDA.

```
1  __global__ void vetSoma (int *A, int *B, int *C, int size){
2         int i = blockIdx.x * blockDim.x + threadIdx.x;
3         if (i < size){
4           C[i] = A[i] + B[i];
5         }
6  }
7
8
9  int main(){
10     ...
11     cudaMemcpy(d_A, h_A, size * sizeof(int), cudaMemcpyHostToDevice);
12     cudaMemcpy(d_B, h_B, size * sizeof(int), cudaMemcpyHostToDevice);
13     ...
14     vetSoma<<<100, 1024>>>(d_A, d_B, d_C, 1024*100);
15     ...
16     cudaMemcpy(h_C, d_C, size * sizeof(int), cudaMemcpyDeviceToHost);
17     ...
18     return 0;
19 }
```

residem nesses tipos de memória podem ser acessadas em velocidade muito alta de uma maneira altamente paralela. Os registradores são alocados à *threads* individuais e cada *thread* só pode acessar seus próprios registradores. A memória compartilhada é alocada em blocos de *threads* e todas as *threads* em um bloco podem acessar variáveis nos locais de memória compartilhada.

2.3 Trabalhos Relacionados

Este trabalho tem como objetivo realizar um estudo sobre o desenvolvimento de aplicações paralelas em arquiteturas heterogêneas. O escalonamento é um ponto importante no desenvolvimento de aplicações paralelas e vem sendo estudado por diversos trabalhos. Kaleem et al. (2014) apresentam duas técnicas para escalonamento de tarefas em aplicações heterogêneas. Para isso, foram utilizadas dezesseis aplicações com características diferentes como: um *kernel* ou muitos *kernels*, uma chamada ou muitas chamadas, regular ou irregular. Os resultados mostraram que a execução heterogênea melhora em até duas vezes o melhor desempenho atingido por processadores e aceleradores sozinhos.

O número de processadores ativos de uma GPU é outro ponto importante relacionado ao desempenho de aplicações heterogêneas. Hong e Kim (2010) apresentam um modelo de desempenho e consumo para arquiteturas do tipo GPU. Os resultados mostra-

ram que utilizando o número ótimo de unidades de execução, é possível reduzir o consumo energético da GPU em até 22.09%.

O paralelismo de tarefas é o modelo de programação mais adequado para paralelismo explícito. Sua associação com dependências de dados oferece uma visão de memória independente da arquitetura. Além disso, ocorre uma redução de sincronizações que é um aspecto essencial para explorar o paralelismo e melhorar a escalabilidade em arquiteturas com aceleradores. Gautier et al. (2013) investigam os desafios no uso de paralelismo de tarefas com dependências de dados em arquiteturas heterogêneas. As conclusões são que um modelo de programação com dependências de dados pode ser eficiente em aceleradores e que o suporte a diferentes estratégias de escalonamento é essencial para o desenvolvimento de aplicações paralelas.

O desenvolvimento de algoritmos também vem sendo estudado por diversos trabalhos. Lashuk et al. (2012) apresentam um novo algoritmo escalável para a técnica *fast multipole method* utilizada para calcular as forças de longo alcançe do problema *n-body*. Assim sendo, na próxima seção encerramos esse capítulo com algumas considerações finais sobre CAD.

3 Metodologia

Este capítulo apresenta a metologia utilizada neste trabalho. Na seção 3.1 são descritas as aplicações, suas características e implementação. O ambiente de execução e as métricas analisadas neste trabalho são mostrados nas seções 3.2 e 3.3, respectivamente.

3.1 Aplicações

Embora existam *benchmarks* paralelos tais como NAS (BAILEY et al., 1991), PARSEC (BIENIA et al., 2008), nenhum deles disponibiliza implementações híbridas. Adicionalmente, eles não exploram as APIs usadas neste trabalho. Desta maneira, foram implementadas sete aplicações paralelas utilizando as APIs CUDA e OpenMP para verificar seu desempenho e escalabilidade em arquiteturas *multi-core* e *many-core*.

3.1.1 Difusão de Calor

A equação do calor é um modelo matemático para difusão de calor em sólidos. Existem diversas variações na literatura (BROWN; CHURCHILL, 2012; HABERMAN, 2013). Na sua forma mais conhecida ela modela a condução de calor em um sólido homogêneo que não possua fontes internas de calor. Esse problema pode ser resolvido através de aplicação de métodos numéricos como o Método das Diferenças Finitas (MDF) (AMES, 2014).

O primeiro passo da discretização é a construção de uma grade com os pontos que estamos interessados em resolver (KOTSIANTIS; KANELLOPOULOS, 2006). A Figura 6 exemplifica o método numérico que será utilizado para resolver a equação do Calor. A malha é discretizada, pontos de contorno são adicionados e após a aplicação do MDF cada ponto u na iteração $n+1$ é calculado por:

$$u_{i,j}^{n+1} = u_{i,j}^{n} + (u_{i-1,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n} + u_{i+1,j}^{n} - 4 * u_{i,j}^{n})$$

Exemplificando na Figura 6, o ponto rosa será calculado. Para isso, os pontos em laranja (ao redor) são usados. Os pontos em amarelo fazem parte da malha e os azuis foram adicionados para facilitar o cálculo dos pontos que se encontram em cantos. Os pontos em cinza não são utilizados. Por fim, o algoritmo retorna uma matriz da iteração n que representa a temperatura em cada ponto discretizado da malha.

A Figura 7 apresenta a implementação paralela com OpenMP. A diretiva `omp parallel for` foi inserida nas linhas **2** e **8** para indicar a divisão de trabalho e computação

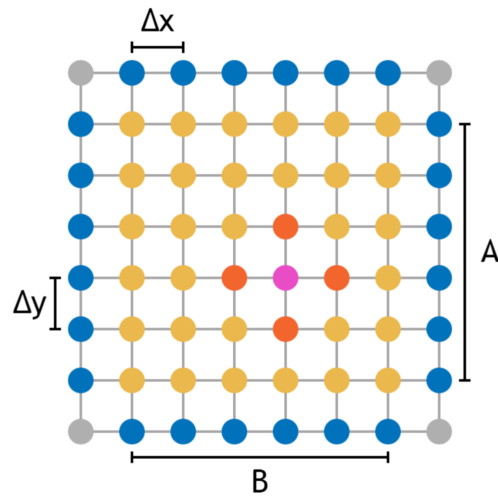


Figura 6 – MDF aplicado ao problema da Transferência de Calor.

de um laço entre as *threads*. A versão em CUDA, mostrada na Figura 8, executa na GPU essas duas funções, sendo que cada *thread* CUDA calcula uma posição da malha.

Figura 7 – Difusão do calor em OpenMP.

```
1 void copy(...){
2         #pragma omp parallel for private(...)  shared(...)
3         for(int offset = 0; offset < size * size; offset++)
4                 ...
5 }
6
7 void blend(...){
8         #pragma omp parallel for private(...)  shared(...)
9         for(int offset = 0; offset < size * size; offset++)
10                ...
11 }
12
13 int main(){
14        ...
15        for (int step = 1; step <= iterations; ++step){
16                copy(...);
17                blend(...);
18        }
19        ...
20        return 0;
21 }
```

Figura 8 – Difusão do calor em CUDA.

```
1  __global__ void copy(...){
2         int offset = blockIdx.x * blockDim.x + threadIdx.x;
3         ...
4  }
5
6  __global__ void blend(...){
7         int offset = blockIdx.x * blockDim.x + threadIdx.x;
8         ...
9  }
10
11 int main(int argc, char **argv[]){
12        ...
13        int threads_per_block = atoi(argv[2]); // Entrada do programa
14        int blocks = (size * size + threads_per_block - 1) /
             threads_per_block;
15        ...
16        for (int step = 1; step <= iterations; ++step){
17               copy<<<threads_per_block, blocks>>>(...);
18               blend<<<threads_per_block, blocks>>>(...);
19        }
20        ...
21        return 0;
22 }
```

3.1.2 Jacobi

O método de Jacobi é um método numérico iterativo utilizado para obter a solução de um sistema de equações lineares $A(\vec{x}) = (\vec{b})$ de forma aproximada (CHAPRA; CANALE, 2012). Essa abordagem consiste em dado um vetor inicial contendo qualquer resposta, seguir um método sistemático para obter uma estimativa melhor a cada iteração.

Esse método é utilizado principalmente para sistemas lineares compostos de matrizes esparsas, as quais, envolvem uma grande porcentagem de coeficiente nulos. A saída do método é um resultado aproximado para x, calculado através da convergência dos vetores (PRESS, 2007).

A versão OpenMP é ilustrada na Figura 9, sendo que duas funções foram paralelizadas. Em ambas funções, existem dois laços que percorrem as matrizes de entrada do sistema linear. Nas linhas **2** e **9** foram inseridas diretivas OpenMP para paralelização de laço. Na Figura 10, mostramos a implementação em CUDA, sendo que nessa, cada *thread* opera sobre um elemento da matriz.

Figura 9 – Jacobi em OpenMP.

```
1  void copy(...){
2          #pragma omp parallel for private(...)  shared(...)
3          for(int j = 1; j < linhas - 1; ++j)
4                  for(int i = 1; i < colunas - 1; ++i)
5                          ...
6  }
7
8  void jacobi(...){
9          #pragma omp parallel for private(...)  shared(...)
10         for(int j = 1; j < linhas - 1; ++j)
11                 for(int i = 1; i < colunas - 1; ++i)
12                         ...
13 }
14
15 int main(){
16         ...
17         for (int step = 1; step <= iterations; ++step){
18                 copy(...);
19                 jacobi(...);
20         }
21         ...
22         return 0;
23 }
```

3.1.3 Lattice Boltzmann

A Dinâmica dos Fluidos Computacional (DFC) pode ser descrita como a simulação numérica de processos físicos como: furacões, previsão de tempo, aerodinâmica, aeroacústica e gerenciamento térmico. A evolução dos sistemas computacionais tornou possível resolver esses problemas de forma eficiente através de novas técnicas de simulação. Para tanto, alguns métodos e algoritmos foram desenvolvidos.

O Método de Lattice Boltzmann (MLB) é considerado uma representação discreta da Equação de Boltzmann, sendo esta, a base da teoria cinética dos gases. O comportamento das partículas que constituem um fluido é representado por meio de uma malha que consiste em um modelo discreto representado por um conjunto de células ou pontos (CHEN; DOOLEN, 1998).

Na literatura, pode-se encontrar diversos modelos de malha (BIFERALE et al., 2011). Neste trabalho foi utilizado o modelo tridimensional conhecido por D3Q19, onde cada partícula pode estar em movimento para 19 direções, como está representado na Figura 11. Esse modelo foi utilizado, pois aproxima-se mais de um modelo real.

Figura 10 – Jacobi em CUDA.

```
1 __global__ void copy(...){
2       int offset = threadIdx.x + blockIdx.x * blockDim.x;
3       int j = offset / linhas + 1;
4       int i = offset % colunas + 1;
5       ...
6 }
7
8 __global__ void jacobi(...){
9       int offset = threadIdx.x + blockIdx.x * blockDim.x;
10      int j = offset / linhas + 1;
11      int i = offset % colunas + 1;
12      ...
13 }
14
15 int main(int argc, char **argv[]){
16      ...
17      int threads_per_block = atoi(argv[2]); // Entrada do programa
18      int blocks = (linhas * colunas + threads_per_block - 1) /
            threads_per_block;
19      ...
20      for (int step = 1; step <= iterations; ++step){
21          copy<<<threads_per_block, blocks>>>(...);
22          blend<<<threads_per_block, blocks>>>(...);
23      }
24      ...
25      return 0;
26 }
```

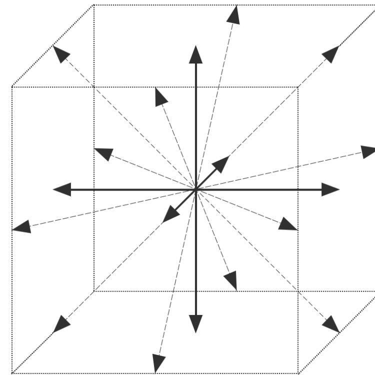
O laço principal do MLB pode ser dividido de acordo com a Figura 12. Em cada iteração os valores da função de equilíbrio são calculados e passados para o cálculo da função de relaxação. A função de equilíbrio é utilizada para propagar a distribuição das partículas para todas as células vizinhas. Após, a distribuição local dos pontos é modificada para satisfazer as condições de contorno.

O MLB foi paralelizado em OpenMP, como mostra a Figura 13. Nesta aplicação, paralelizamos quatro funções e quatro laços, foram incluídas diretivas de compilação nas linhas **2**, **8**, **14**, **22**. Nossa implementação em CUDA (Figura 14) inclui a execução dessas mesmas quatro funções na GPU.

3.1.4 Ordenação Par-Ímpar

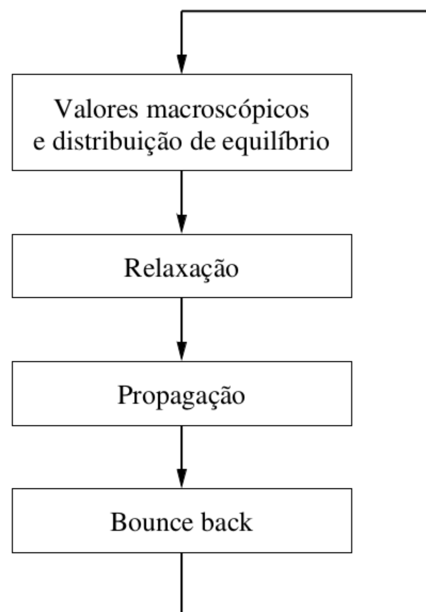
A ordenação Par-Ímpar consiste de um algoritmo de ordenação por comparação baseado no *bubble-sort*. Os elementos são comparados em duas fases, sendo que na pri-

Figura 11 – Modelo de *Lattice* D3Q19.



Adaptado de: Schepke (2007, p. 29)

Figura 12 – Laço principal do MLB.



Fonte: Schepke (2007, p. 37)

meira fase os elementos pares são comparados com os ímpares e trocados se necessário. Na segunda etapa, os ímpares são comparados com os pares. Estas duas etapas são repetidas durante N/2 iterações, onde N é o número de elementos do vetor.

Nas Figuras 15 e 16 mostramos nossas versões OpenMP e CUDA, respectivamente. A versão OpenMP paraleliza dois laços nas linhas **5** e **8**. Para a versão CUDA, foi necessário criar duas funções: uma para ordenar os pares e outra para ordenar os ímpares. Isso foi necessário, devido a sincronização que deve ser feita ao fim do laço da linha **19**.

3.1.5 Quadratura de Gauss

A integração pela Quadratura de Gauss é outro método numérico que aproxima a integral de uma função. Uma quadratura de **n** pontos produz um resultado exato para polinômios de grau **2n - 1**. Em questões de convergência, o método da quadratura de gauss é o considerado o mais rápido e eficiente (TREFETHEN, 2008).

A Figura 17 apresenta a implementação em OpenMP da Quadratura de Gauss. A diretiva incluída na linha **3** indica o paralelismo de um laço, sendo que `reduce(+: total)`, faz com que, ao final da região paralela, a soma dos valores de cada *thread* seja feita. A versão CUDA (Figura 18), implementa uma redução em GPUs. Para isso, usa-se a memória compartilhada a fim de aumentar o desempenho dos acessos a memória.

3.1.6 Regra dos Trapézios

A regra dos trapézios é a primeira regra fechada para integração de Newton-Cotes. Ela baseia-se na estratégia de substituir uma função complicada por uma aproximação polinomial de primeira ordem (CHAPRA; CANALE, 2012).

$$I = \int_a^b f(x)dx = \int_a^b f(a) + \frac{f(b) - f(a)}{b - a}(x - a)$$

A área sob esta linha reta é aproximadamente:

$$I = \int_a^b f(x)dx = (b - a)\frac{f(a) + f(b)}{2}$$

Uma forma de melhorar a aproximação da regra dos trapézios é dividir o intervalo de **a** a **b** em um número de segmentos (**n**) e aplicar o método em cada segmento. As áreas de cada segmento podem ser somadas para concluir-se a área total.

Existiram **n + 1** pontos $(x_0, x_1, ..., x_n)$, os quais resultaram em **n** segmentos de tamanho **h**.

$$h = \frac{b - a}{n}$$

Substituindo cada segmento na regra dos trapézios temos:

$$I = h\frac{f(x_0) + f(x_1)}{2} + h\frac{f(x_1) + f(x_2)}{2} + ... + h\frac{f(x_{n-1}) + f(x_n)}{2}$$

Agrupando os termos, podemos calcular a integral total por:

$$I = \int_a^b f(x)dx = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right]$$

O resultado deste método é um número real que representa o valor da área da função $f(x)$ entre os pontos a e b.

As Figuras 19 e 20 apresentam as versões paralelas da Regra dos Trapézios. Em geral, as implementações são iguais as da Quadratura de Gauss, com a diferença, das operações de dentro dos laços.

3.2 Ambiente de Execução

Nesta seção iremos apresentar o ambiente de execução utilizado neste trabalho. O ambiente é uma *workstation* composta por dois processadores Intel Xeon E5, uma GPU Tesla C2075. As principais características do ambiente de execução são apresentadas na Tabela 2.

Tabela 2 – Ambiente de Execução.

	Tesla C2075	**2 x Xeon E5-2650**
Frequência	1.15 GHz	2.0 GHz
Tecnologia	40 nm	32 nm
# de Núcleos	14	2 x 8
# de Threads	448	2 x 16
Cache L1	48 KB	32 KB
Cache L2	768 KB	256 MB
Cache L3	—-	20 MB
Memória RAM	6 GB	128 GB

O ambiente de execução possui dois processadores e um acelerador. O processador Intel Xeon E5-2650 possui 8 *cores* operando em 2.0 GHz de frequência com suporte a *Hyper-Threading*. O sistema de memória do processador é formado por três níveis de *cache* e a memória principal:

- Cache L1 privada de 32 KB para dados e 32 KB para instruções;
- Cache L2 privada de 256 KB unificada para dados e instruções;
- Cache L3 compartilhada de 20 MB unificada para dados e instruções;
- Memória principal de 128 GB.

A GPU Nvidia Tesla C2075 possui 32 SPs em cada um dos seus 14 SMs, sendo que cada SP trabalha em 1.15 GHz. Seu sistema de memória consiste de dois níveis de *cache* e a memória global do acelerador:

- Cache L1 de 48 KB por SMs;
- Cache L2 de 768 KB por SMs;
- Memória global de 6 GB.

3.3 Métricas Avaliadas

A análise de desempenho será feita a partir de um conjunto de métricas. Para a obtenção destes, será utilizada a ferramenta Nvprof (NVIDIA, 2011) e a biblioteca `time.h` da Linguagem de Programação C. A ferramenta de *profiling* Nvprof permite que dados das GPUs sejam coletados e visualizados, auxiliando na otimização de desempenho.

As métricas que serão utilizadas neste trabalho são:

Tempo de Execução: O tempo total de execução de cada aplicação foi obtido através da biblioteca `time.h`.

Speedup: O *Speedup* indica a relação de tempo entre o programa paralelo e sua versão sequencial (ROSE; NAVAUX, 2002). Cada aplicação tem a sua curva de *speedup*, dependendo do trabalho e da incidência de complicadores. Toda aplicação tem um número de unidades ativas ideal para a obtenção do melhor desempenho (*sweetspot*) não sendo verdade que quanto mais unidades ativas melhor.

Escalabilidade: Consiste em analisar o comportamento das IPPs e das arquiteturas em termos de desempenho quando o número de unidades de trabalho é aumentado. A **escalabilidade forte** analisa a solução quando o número de trabalhadores varia e o tamanho do problema é fixado. A **escalabilidade fraca** analisa a solução quando o trabalho por trabalhador é fixado e o número de trabalhadores é aumentado (LASHUK et al., 2012).

Figura 13 – Lattice Boltzmann em OpenMP.

```
1  void redistribute(...){
2          #pragma omp parallel for private(...)  shared(...)
3          for (int y = 0; y < lattice->ly; ++y)
4                  ...
5  }
6
7  void propagate(...){
8          #pragma omp parallel for private(...)  shared(...)
9          for (int x = 0; x < lattice->lx; ++x)
10                 for (int y = 0; y < lattice->ly; ++y)
11                         ...
12 }
13
14 void bounceback(...){
15         #pragma omp parallel for private(...)  shared(...)
16         for (int x = 0; x < lattice->lx; ++x)
17                 for (int y = 0; y < lattice->ly; ++y)
18                         ...
19 }
20
21 void relaxation(...){
22         #pragma omp parallel for private(...)  shared(...)
23         for (int x = 0; x < lattice->lx; ++x)
24                 for (int y = 0; y < lattice->ly; ++y)
25                         ...
26 }
27
28 int main(){
29         ...
30         for (int step = 1; step <= iterations; ++step){
31                 redistribute(...);
32                 propagate(...);
33                 bounceback(...);
34                 relaxation(...);
35         }
36         ...
37         return 0;
38 }
```

Figura 14 – Lattice Boltzmann em CUDA.

```
1  __global__ void redistribute(...){
2          int y = blockIdx.x * blockDim.x + threadIdx.x;
3          ...
4  }
5
6  __global__ void propagate(...){
7          int offset = blockIdx.x * blockDim.x + threadIdx.x;
8          int x = offset / ly;
9          int y = offset % ly;
10         ...
11 }
12
13 __global__ void bounceback(...){
14         int offset = blockIdx.x * blockDim.x + threadIdx.x;
15         int x = offset / ly;
16         int y = offset % ly;
17         ...
18 }
19
20 __global__ void relaxation(...){
21         int offset = blockIdx.x * blockDim.x + threadIdx.x;
22         int x = offset / ly;
23         int y = offset % ly;
24         ...
25 }
26
27
28 int main(int argc, char **argv[]){
29         ...
30         int threads_per_block = atoi(argv[2]); // Entrada do programa
31         int blocks_xy = (lattice.lx * lattice.ly + threads_per_block -
              1) / threads_per_block;
32         int blocks_y = (lattice.ly + threads_per_block - 1) /
              threads_per_block;
33         ...
34         for (int step = 1; step <= iterations; ++step){
35                 redistribute<<<threads_per_block, blocks_y>>>(...);
36             propagate<<<threads_per_block, blocks_xy>>>(...);
37             bounceback<<<threads_per_block, blocks_xy>>>(...);
38             relaxation<<<threads_per_block, blocks_xy>>>(...);
39         }
40         ...
41         return 0;
42 }
```

Figura 15 – Ordenação Par-Ímpar em OpenMP.

```
1 void oddEvenSort(...){
2         #pragma omp parallel private(...)  shared(...)
3         {
4                 for(int j = 0; j < size / 2; j++){
5                         #pragma omp for
6                         for(i = 0; i < size - 1; i += 2)
7                                 ...
8                         #pragma omp for
9                         for(i = 1; i < size - 1; i += 2)
10                                ...
11                }
12        }
13 }
14
15 int main(){
16        ...
17        oddEvenSort(...);
18        ...
19        return 0;
20 }
```

Figura 16 – Ordenação Par-Ímpar em CUDA.

```
1
2 __global__ void odd(...){
3         int tid = threadIdx.x + blockIdx.x * blockDim.x;
4         int i = 2 * tid; // pares
5         ...
6 }
7
8 __global__ void even(...){
9         int tid = threadIdx.x + blockIdx.x * blockDim.x;
10        int i = 2 * tid + 1; // mpares
11        ...
12 }
13
14 int main(int argc, char **argv[]){
15        ...
16        int threads_per_block = atoi(argv[2]); // Entrada do programa
17        int blocks = (size + threads_per_block - 1) / threads_per_block
18        ...
19        for(int j = 0; j < size / 2; j++){
20               odd<<<threads_per_block, blocks>>>(...);
21               even<<<threads_per_block, blocks>>>(...);
22        }
23        ...
24        return 0;
25 }
```

Figura 17 – Quadratura de Gauss em OpenMP.

```
1 int main(){
2         ...
3         #pragma omp parallel for private(...)  shared(...)
             reduction(+:  total)
4         for(i = 0; i < divisoes; ++i)
5                ...
6         return 0;
7 }
```

Figura 18 – Quadratura de Gauss em CUDA.

```
1 __global__ void sum_reduction(...){
2         __shared__ double cache[threadsPerBlock];
3         long long int offset = threadIdx.x + blockIdx.x * blockDim.x;
4         int cacheIdx = threadIdx.x;
5         ...
6 }
7
8 int main(int argc, char **argv[]){
9         ...
10        int threads_per_block = atoi(argv[2]); // Entrada do programa
11        int blocks = (divisoes + threads_per_block - 1) /
             threads_per_block;
12        ...
13        sum_reduction<<<threads_per_block, blocks>>>(...);
14        ...
15        return 0;
16 }
```

Figura 19 – Regra dos Trapézios em OpenMP.

```
1 int main(){
2         ...
3         #pragma omp parallel for private(...)  shared(...)
             reduction(+:  total)
4         for(i = 0; i < divisoes; ++i)
5                 ...
6         return 0;
7 }
```

Figura 20 – Regra dos Trapézios em CUDA.

```
1  __global__ void sum_reduction(...){
2          __shared__ float cache[threadsPerBlock];
3          long long int offset = threadIdx.x + blockIdx.x * blockDim.x;
4          int cacheIdx = threadIdx.x;
5          ...
6  }
7
8  int main(int argc, char **argv[]){
9          ...
10         int threads_per_block = atoi(argv[2]); // Entrada do programa
11         int blocks = (divisoes + threads_per_block - 1) /
               threads_per_block;
12         ...
13         sum_reduction<<<threads_per_block, blocks>>>(...);
14         ...
15         return 0;
16 }
```

4 Resultados

Este capítulo apresenta resultados de desempenho das aplicações utilizando as IPPs OpenMP e CUDA sobre uma arquitetura composta por dois processadores Intel Xeon e uma GPU Tesla. Os resultados apresentados correspondem ao Tempo de Execução e *Speedup* das implementações desenvolvidas sobre diferentes tamanhos do problema e número de *threads*. O compilador usado em todas implementações foi o GCC 4.8.2 sem *flags* de otimização. Os tempos de execução são a média de 30 execuções, onde o desvio foi menor do que 1%.

As aplicações foram descritas na seção 3.1, sendo que o tipo e o tamanho de entrada são descritos na Tabela 3 e o tempo de execução sequencial na Tabela 4. Nos processadores, os programas foram executados em 2, 4, 8, 16 e 32 *threads*, devido a limitação do processador. A versão do OpenMP foi a 4.0, enquanto que a versão do CUDA, utilizada para GPU foi a 7.5.

Tabela 3 – Tipo e Tamanho de Entrada de cada Aplicação

Aplicações	**Tipo**	**Entrada**			
Difusão de Calor	Matriz N x N	2000	3000	4000	50000
Jacobi	Matriz N x N	1024	2048	4096	8192
Lattice Boltzmann	Matriz N x N	256	512	1024	1536
Ordenação Par-Ímpar	Vetor (Milhares)	250	300	350	400
Quadratura de Gauss	Iterações (Bilhões)	1	5	10	20
Regra dos Trapézios	Iterações (Bilhões)	1	5	10	20

Tabela 4 – Tipo de dado e Tempo de Execução Sequencial de cada Aplicação

Aplicações	**Tipo de dado**	**Tempo (seg)**			
Difusão de Calor	*float*	89.8	195.5	339.5	524.8
Jacobi	*float*	12.3	43.7	166.4	646.9
Lattice Boltzmann	*double*	22.2	87.9	334.3	714.9
Ordenação Par-Ímpar	*int*	189.6	272.7	371.1	484.4
Quadratura de Gauss	*double*	42.6	206.9	413.7	827.2
Regra dos Trapézios	*float*	28.1	138.6	276.9	553.9

4.1 Análise de Desempenho do Processador

As aplicações foram executadas no processador Intel Xeon utilizando a IPP OpenMP, variando-se o número de *threads* e as entradas. Os fatores que influenciam no desempenho são: criação de *threads*, divisão da carga de trabalho e comunicação entre *threads* via memória compartilhada. As Figuras 21, 22, 23, 24, 25, 26 apresentam os resultados de *speedup* no processador. O eixo x de cada gráfico corresponde a variações no tamanho do problema (descrito na Tabela 3) e no número de *threads*.

O *speedup* das aplicações esteve em sua maioria próximo ao ideal (2 para 2 *threads*, 4 para 4 *threads*). O desempenho torna-se não linear quando executadas com 32 *threads*, devido, ao uso da tecnologia *Hyper-Threading*, que permite que duas *threads* executem no mesmo *core* aproveitando-se das bolhas do *pipeline*. Segundo a Intel, essa tecnologia oferece um aumento de desempenho de até 30% (JR et al., 2009), sendo, que para desempenho linear, ao dobrar o número de *threads*, o ganho deve dobrar (ganho de 100%).

A fim de analisar a escalabilidade, variamos o número de *threads* e o tamanho do problema. Em geral nossas implementações OpenMP escalaram em ambos os casos. No primeiro, fixamos o tamanho do problema e variamos o número de *threads* de 2 à 32 (múltiplos de base 2). No segundo, definimos quatro tamanhos de problema (de 15 segundos, a 15 minutos).

Cabe ressaltar que, dentre as aplicações, a ordenação *Par-Ímpar* não escalou com 32 *threads*, devido ao número de regiões paralelas criadas. Neste caso, houve um *overhead* causado pelo gerenciador do OpenMP. Em cada iteração é necessário fazer sincronizações implicitamente, implicando em maior tempo das *threads* em espera.

4.2 Análise de Desempenho da Placa Gráfica

Executamos nossas aplicações na Placa Gráfica Tescla C2075 utilizando a IPP CUDA. Nesse caso, variamos o número de *threads* por bloco e o tamanho das entradas. Na GPU, os fatores que influenciam no desempenho são: número total de *threads*, acessos a memória global e fluxo das *threads*. As Figuras 27, 28, 29, 30, 31, 32 apresentam os resultados de *speedup* na placa Gráfica. O eixo x de cada gráfico corresponde a variações no tamanho do problema (descrito na Tabela 3) e no número de *threads* por bloco.

De forma diferente dos processadores, nas placas gráficas não existe um consenso de qual o *speedup* ideal para cada arquitetura de GPU. Isso deve-se as características dessa arquitetura. Por exemplo, uma GPU com 448 *CUDA Cores* não terá um desempenho de 448 vezes, pois, esses *cores* são simples comparados a um *core* de processador.

Com o propósito de melhorar o desempenho de nossas implementações, variamos o número de *threads* por bloco de 128 à 1024. Essa variação não implica na mudança do número total de *threads*, mas, no número de *threads* por bloco, sendo que cada bloco é escalonado aos multiprocessadores. Nossos experimentos mostraram que em média, blocos de 128 *threads* otimizam o desempenho dessa GPU. Devido a limitação do número de registradores por bloco, os Métodos de Lattice Boltzmann e Jacobi limitaram-se à executar 512 *threads*.

Em relação a escalabilidade, que foi avaliada variando o tamanho do problema, o desempenho da GPU manteve-se constante ou aumentou. O desempenho variou de

acordo com o tipo de aplicação. Os Métodos de Lattice Boltzmann e Jacobi que baseiam-se em operações em matrizes bi e tridimensionais tiveram *speedup* de 26 vezes em média. O algoritmo de ordenação Par-Ímpar apresentou *speedup* de 26, sendo que esse, baseia-se em operações de comparação e acessos a memória global. A difusão de calor apresentou um desempenho intermediário, tendo em média 66 de *speedup*. As integrações pela quadratura de Gauss e pela regra dos trapézios obtiveram os melhores resultados. Em média, o *speedup* foi de 187 para Gauss e 224 para trapézios. Essa diferença deve-se da primeira utilizar o tipo `double` e a segunda o tipo `float`.

4.3 Comparação de Desempenho das Arquiteturas

Como pode ser observado nas Figuras 33, 34, 35, 36, 37, 38, a placa gráfica Tesla C2075 executou o conjunto de aplicações em menos tempo em relação ao processador Intel Xeon E5. Isso ocorreu devido as características das aplicações. As placas gráficas são indicadas para aplicações que fazem uso intenso de operações matemáticas e comparações. Outra característica das aplicações que favorecem a GPU é a não existência de dependência entre as iterações.

Ao comparar o desempenho da melhor versão OpenMP com a melhor versão CUDA de cada aplicação, temos o ganho da placa gráfica para cada aplicação é de:

- Difusão de Calor - 6.38x;
- Jacobi - 3.23x;
- Lattice Boltzmann - 1.64x;
- Ordenação Par-Ímpar - 2.54x;
- Quadratura de Gauss - 7.55x;
- Regra dos Trapézios - 8.56x.

4.4 Considerações sobre os Experimentos

Este capítulo apresentou os resultados obtidos na execução do conjunto de aplicações no processador Intel Xeon utilizado OpenMP, na placa gráfica Tesla com CUDA e a comparação destes resultados. O objetivo dessa análise foi mostrar a escalabilidade das implementações e o elevado desempenho do *framework* CUDA para placas gráficas.

A Figura 39 mostra o melhor resultado de cada uma das sete aplicações nas arquiteturas Xeon e Tesla. Nossas análises mostraram que as características das aplicações impactam diretamente na diferença de desempenho entre as arquiteturas.

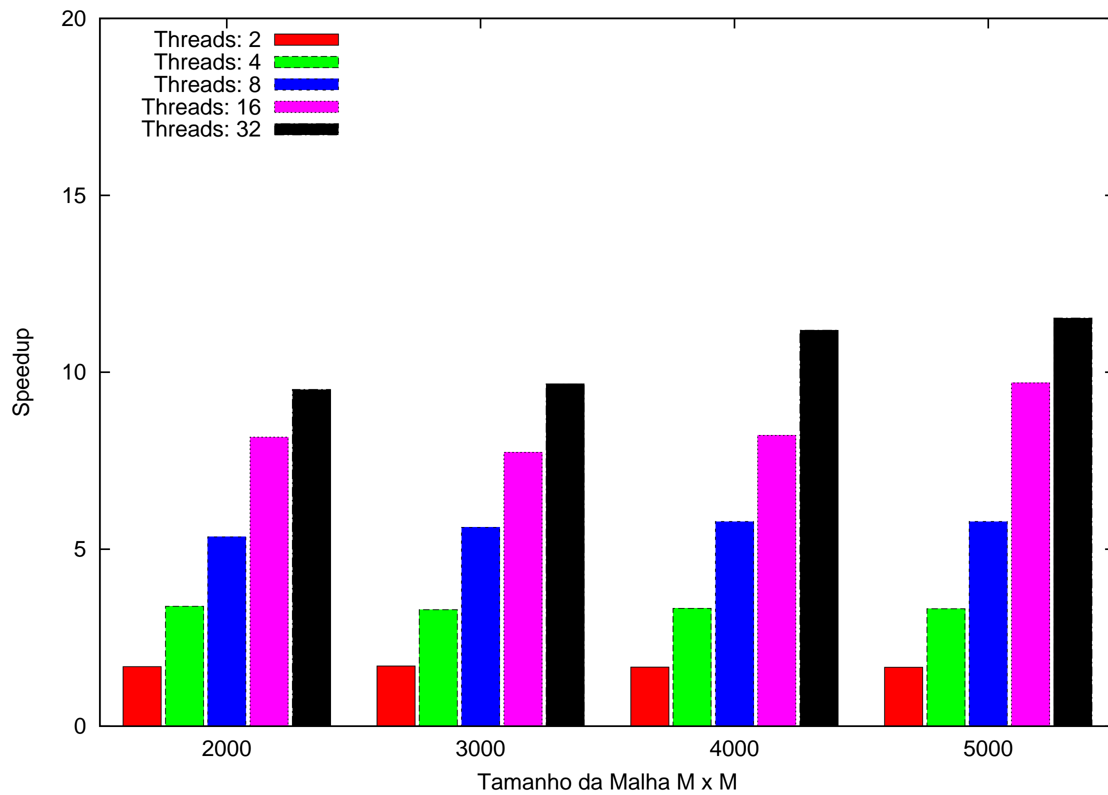


Figura 21 – Speedup da Difusão de Calor em Processadores (OpenMP)

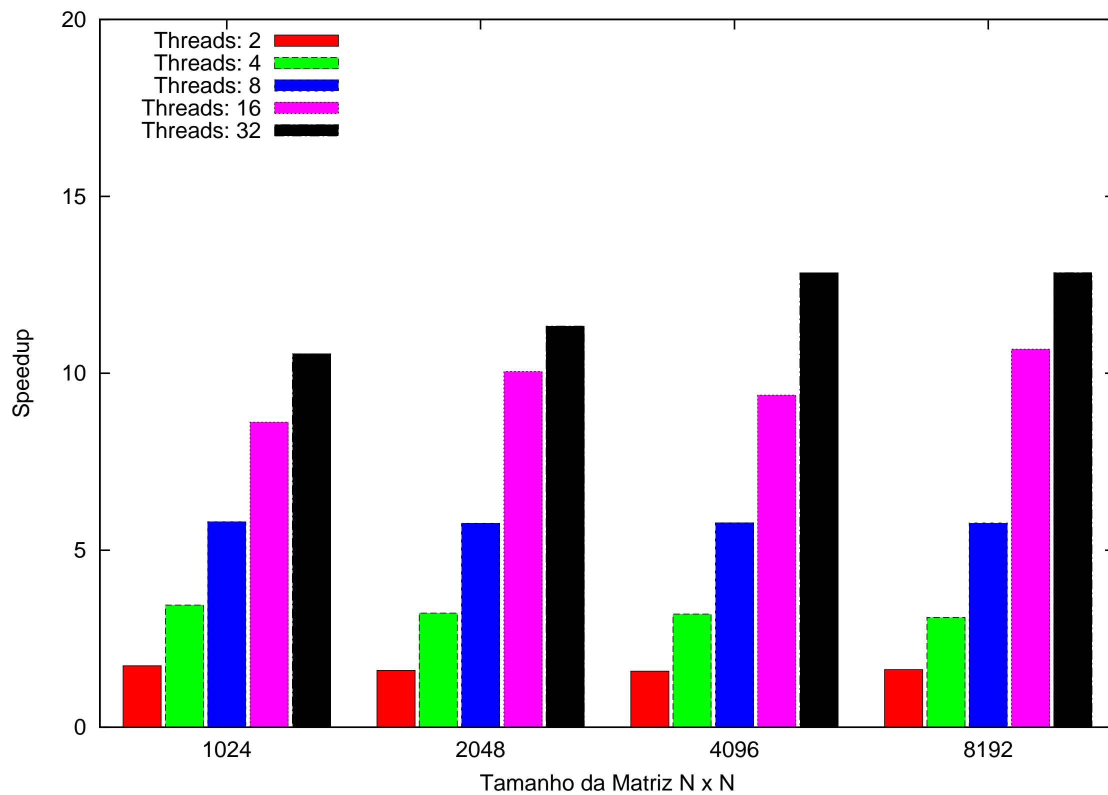


Figura 22 – Speedup do Método de Jacobi em Processadores (OpenMP)

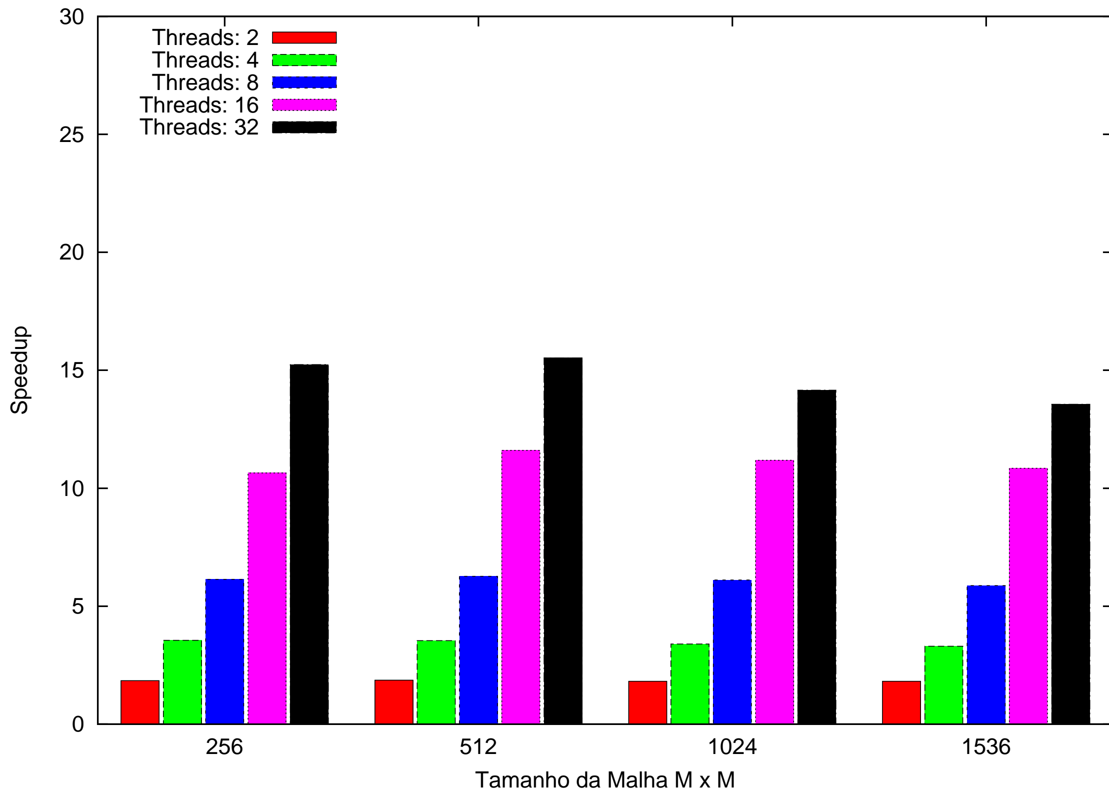


Figura 23 – Speedup do Método de Lattice Boltzmann em Processadores (OpenMP)

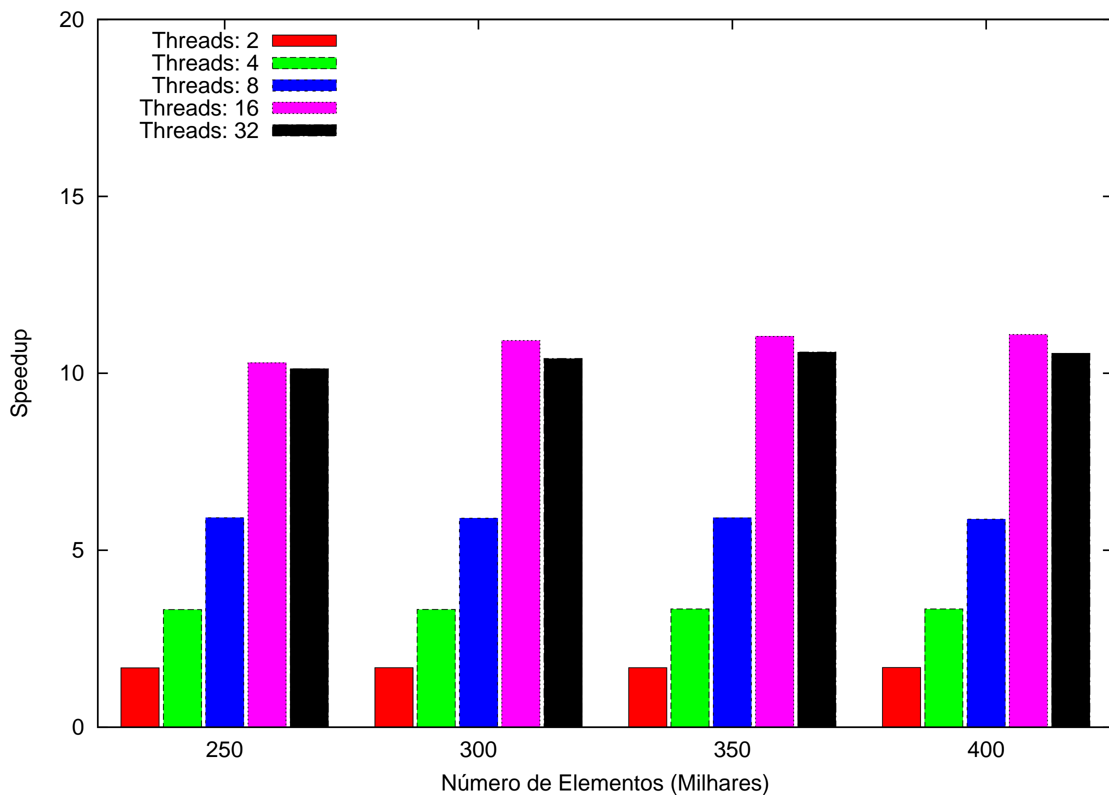


Figura 24 – Speedup da Ordenação Par-Ímpar em Processadores (OpenMP)

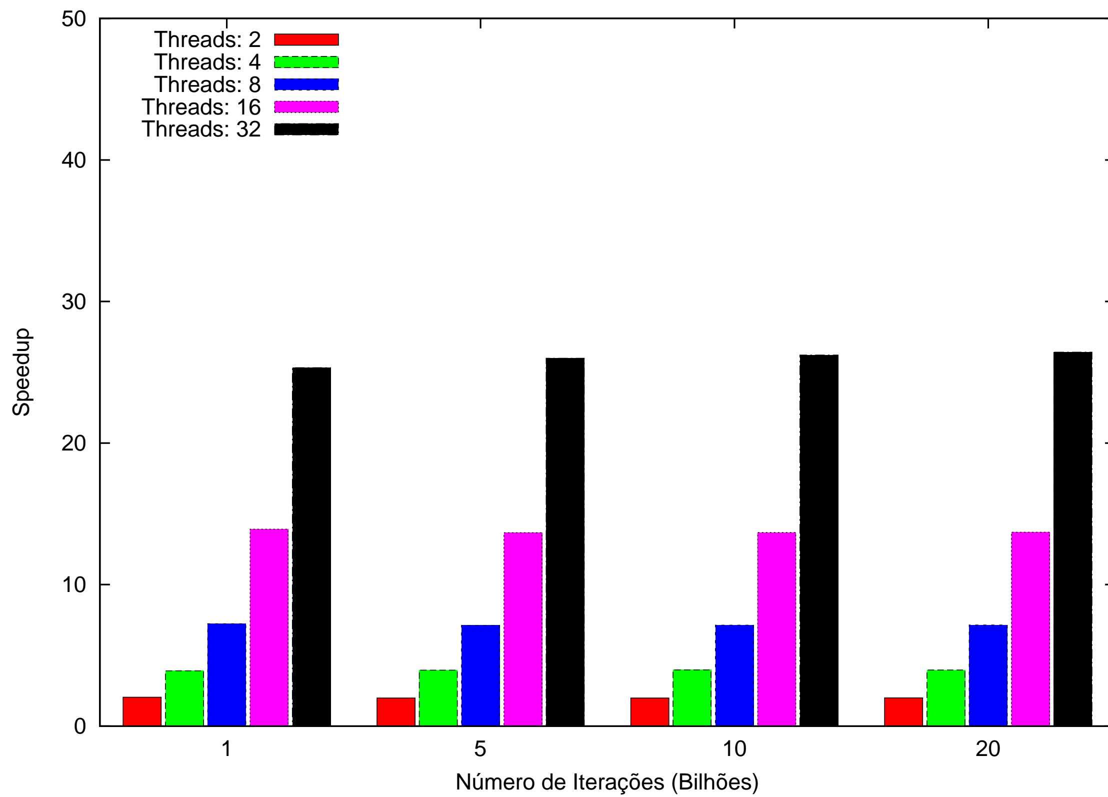


Figura 25 – Speedup da Quadratura de Gauss em Processadores (OpenMP)

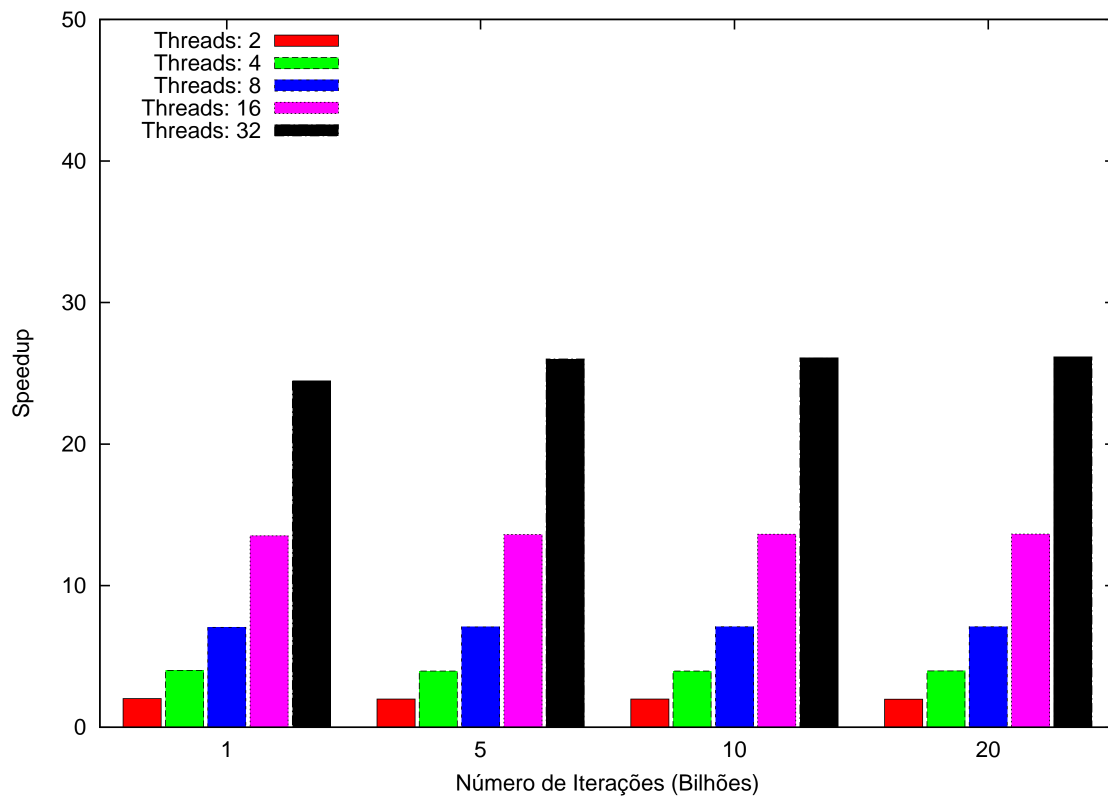


Figura 26 – Speedup da Regra dos Trapézios em Processadores (OpenMP)

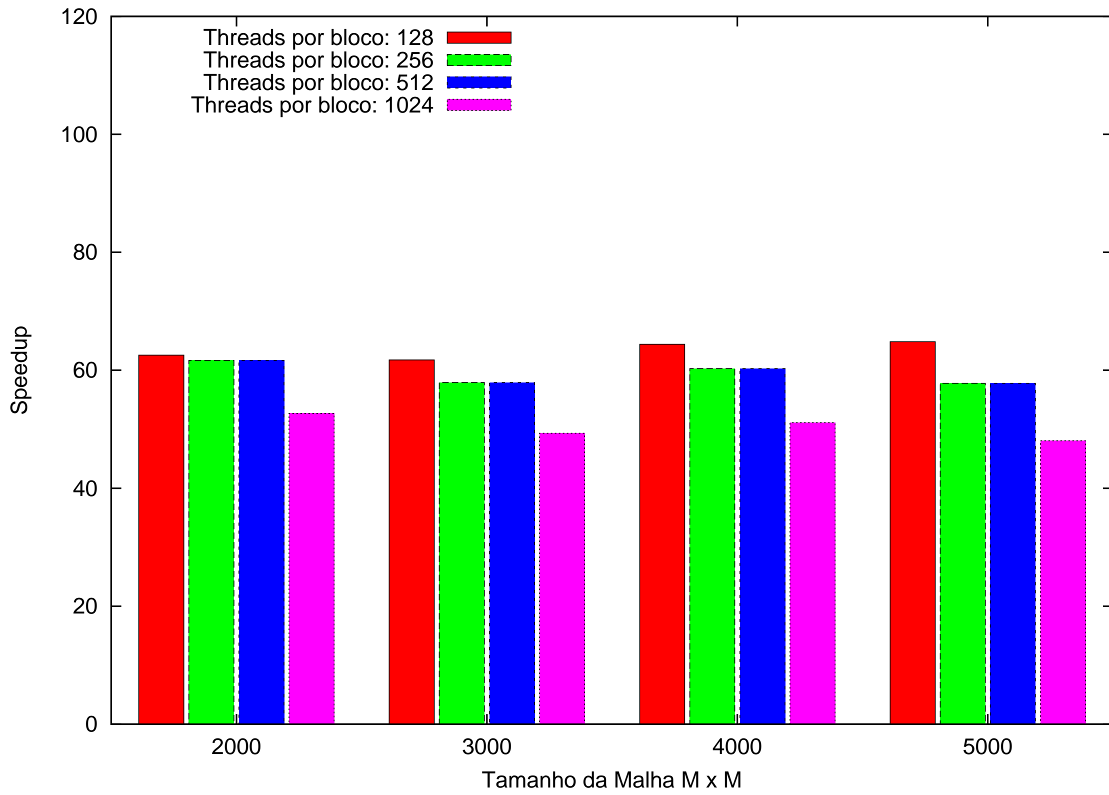


Figura 27 – Speedup da Difusão de Calor em Placas Gráficas (CUDA)

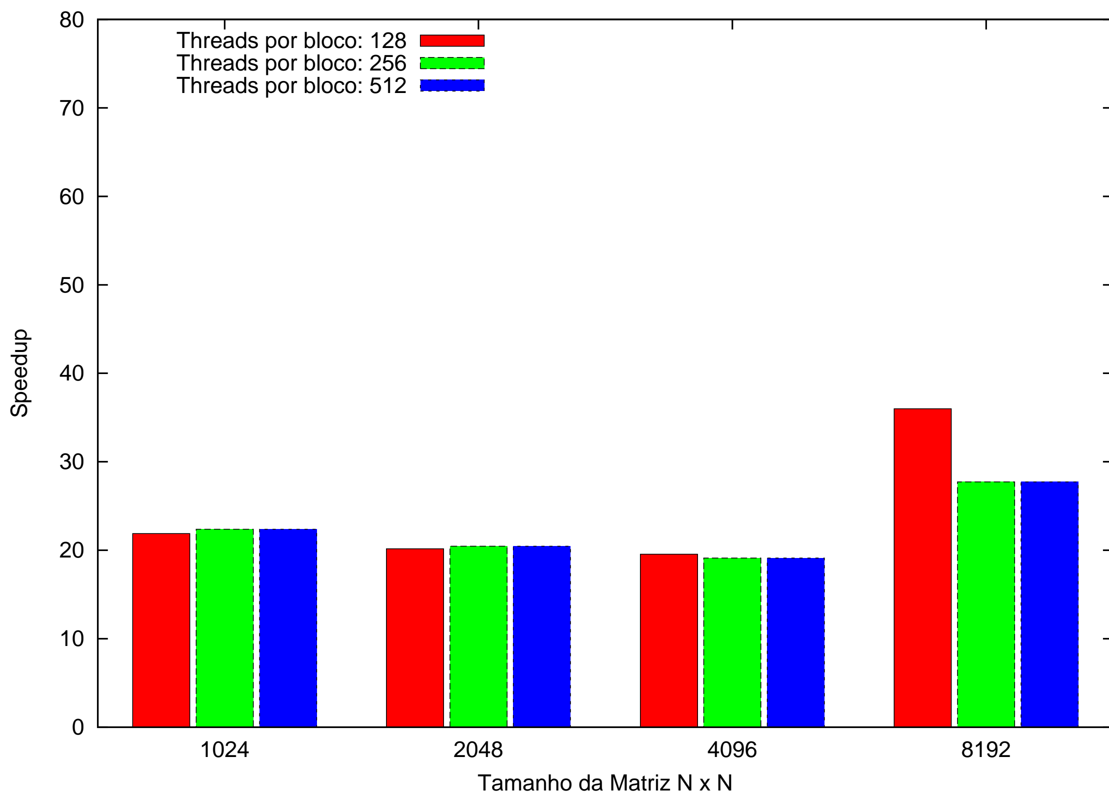


Figura 28 – Speedup do Método de Jacobi em Placas Gráficas (CUDA)

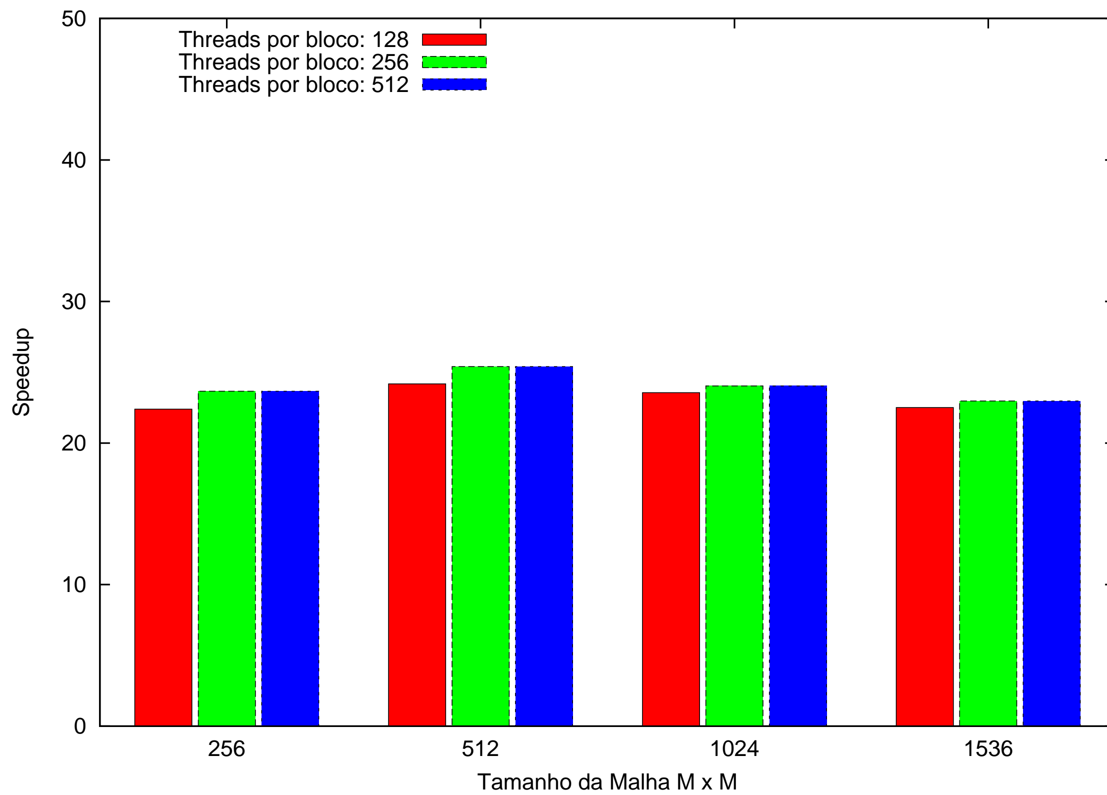


Figura 29 – Speedup do Método de Lattice Boltzmann em Placas Gráficas (CUDA)

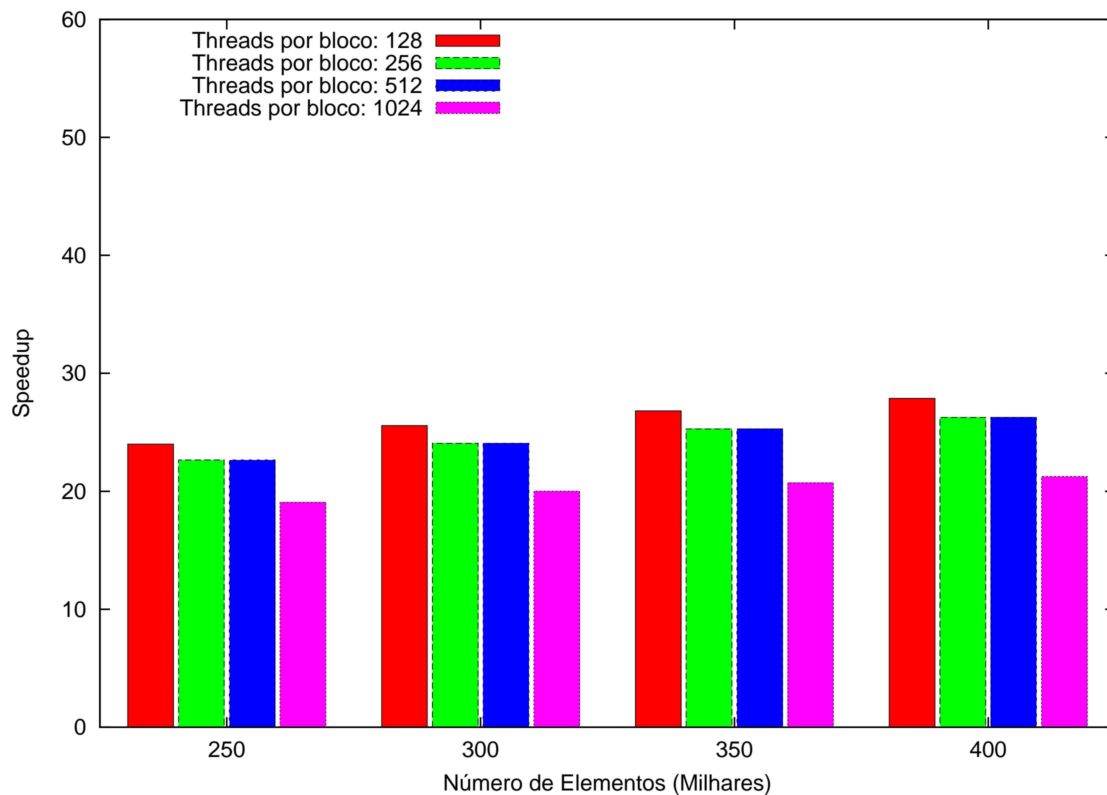


Figura 30 – Speedup da Ordenação Par-Ímpar em Placas Gráficas (CUDA)

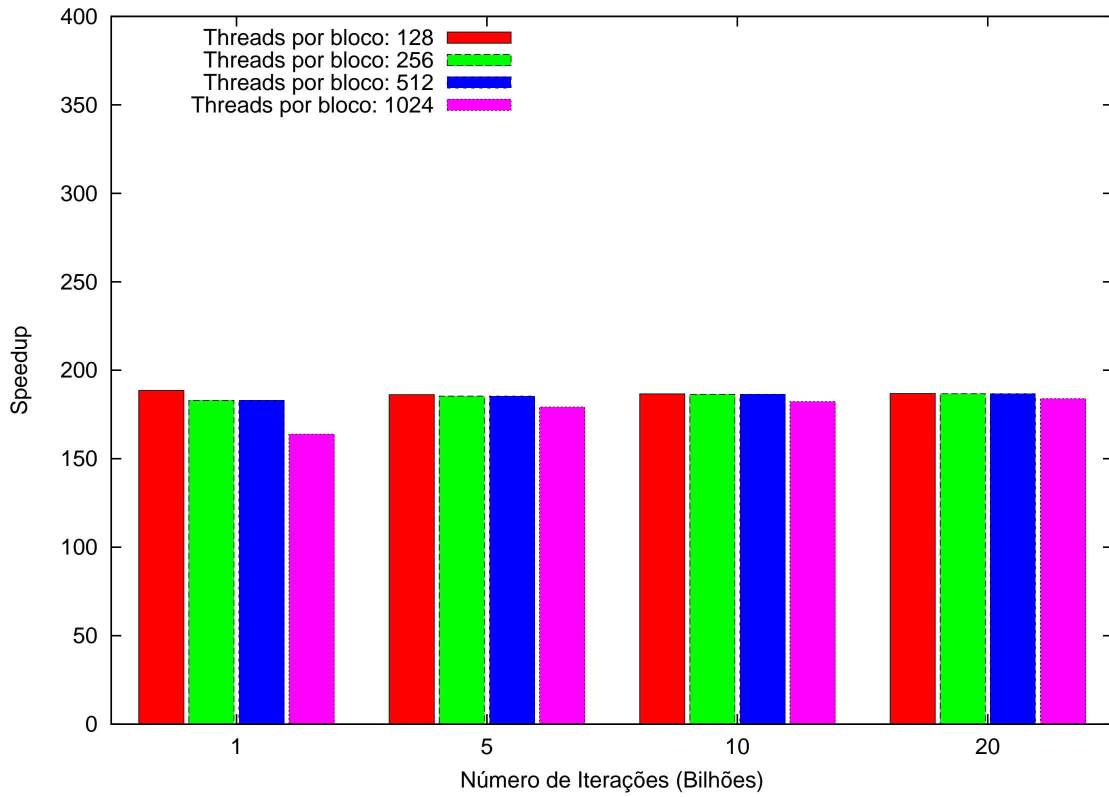


Figura 31 – Speedup da Quadratura de Gauss em Placas Gráficas (CUDA)

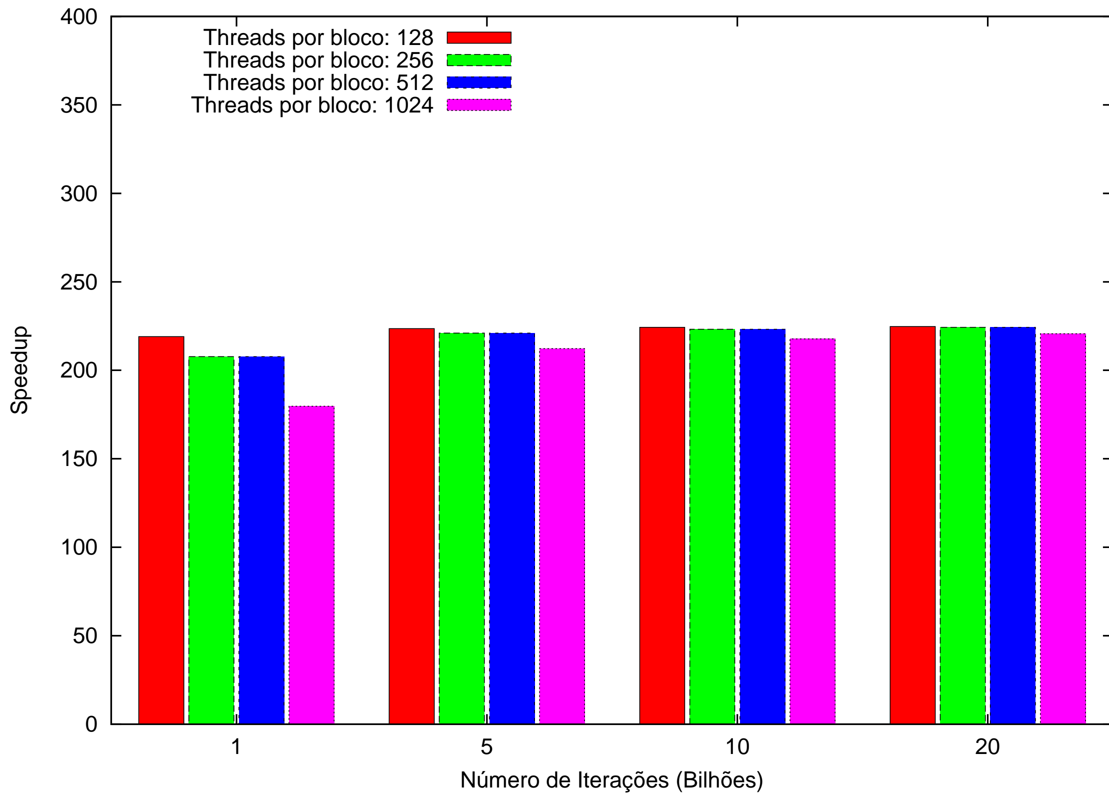


Figura 32 – Speedup da Regra dos Trapézios em Placas Gráficas (CUDA)

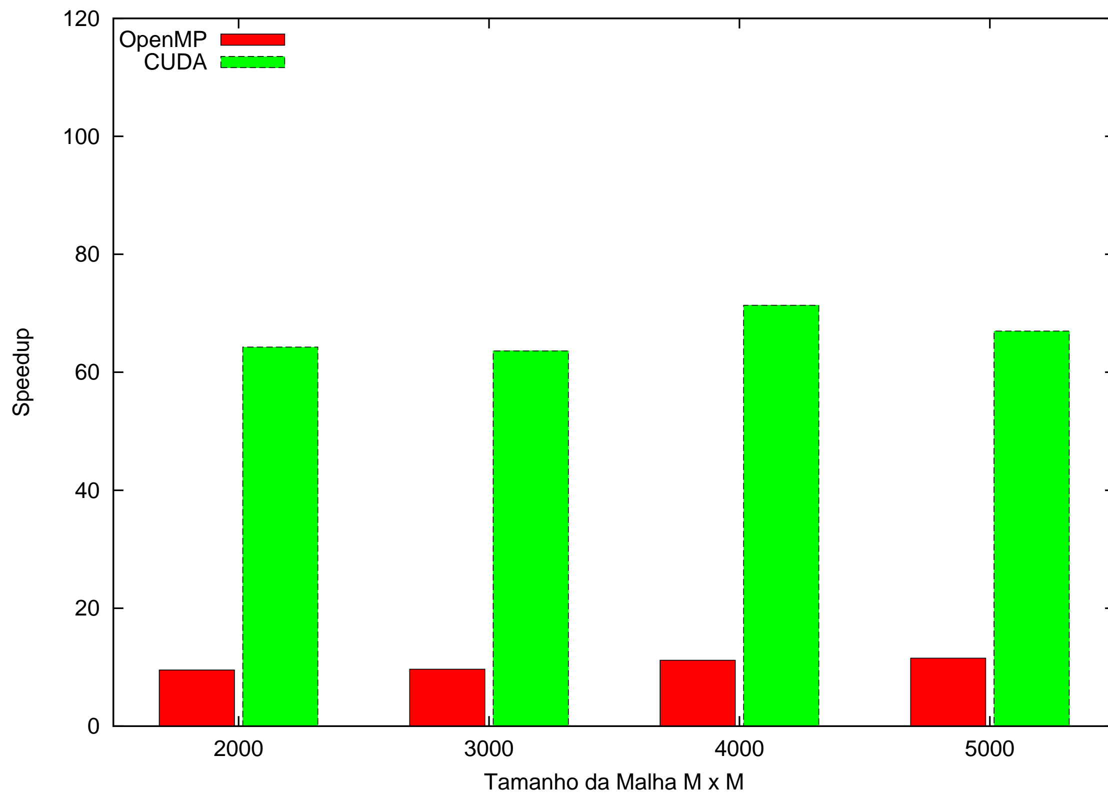


Figura 33 – Comparação entre Processador e Placa Gráfica - Difusão de Calor

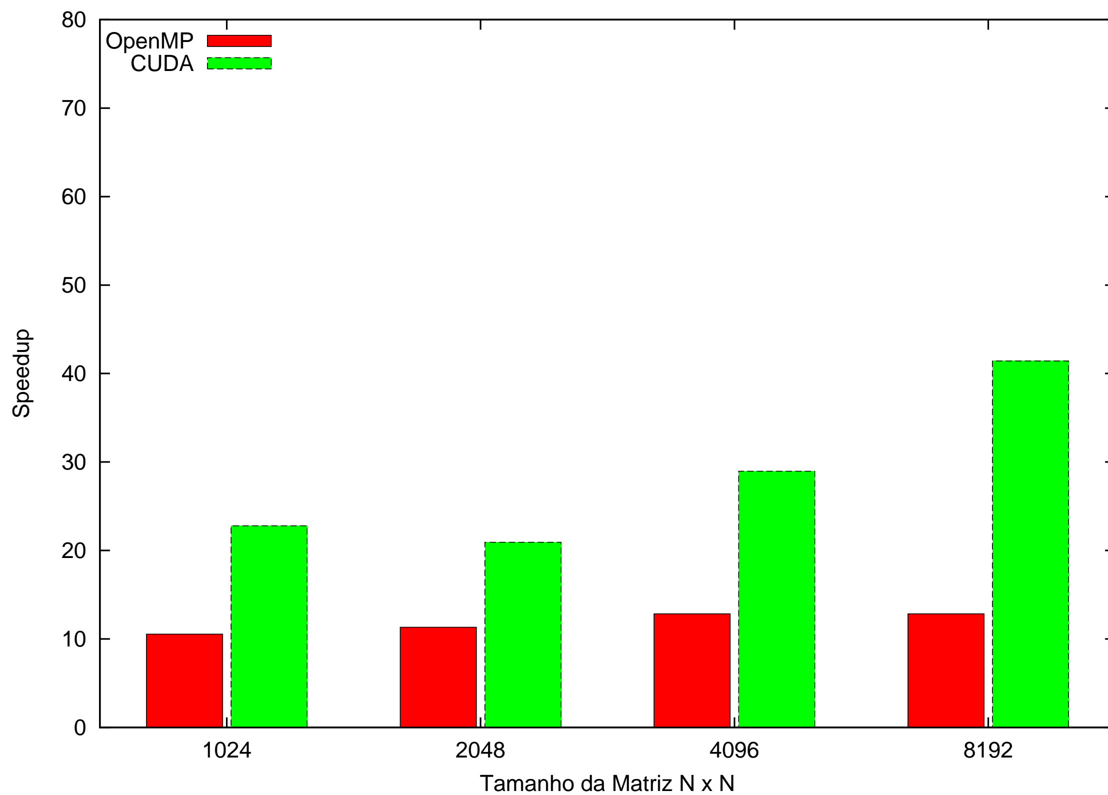


Figura 34 – Speedup do Método de Jacobi

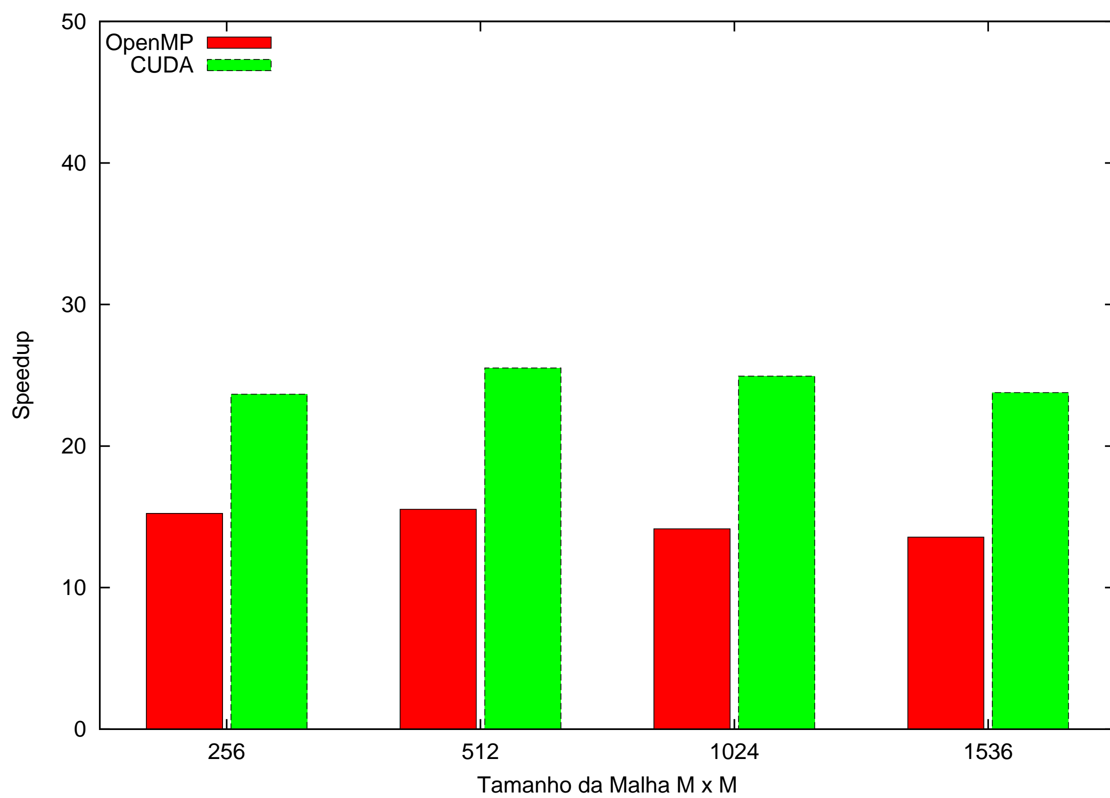


Figura 35 – Speedup do Método de Lattice Boltzmann

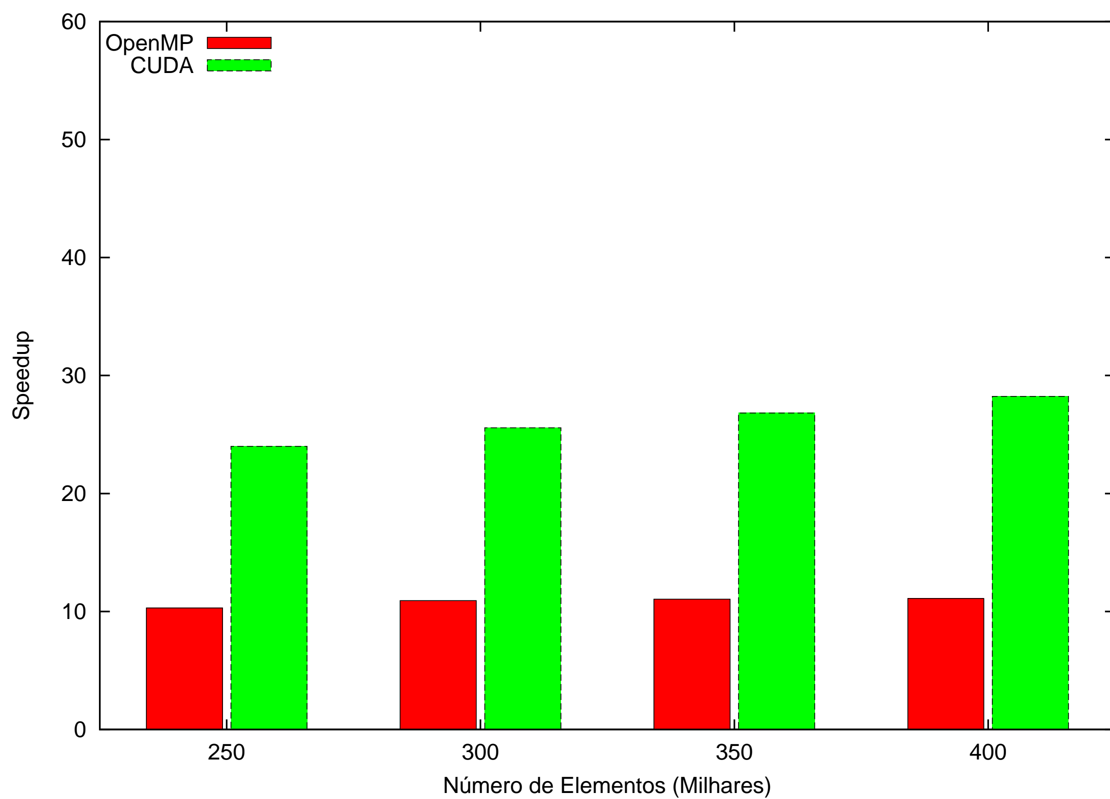


Figura 36 – Comparação entre Processador e Placa Gráfica - Ordenação Par-Ímpar

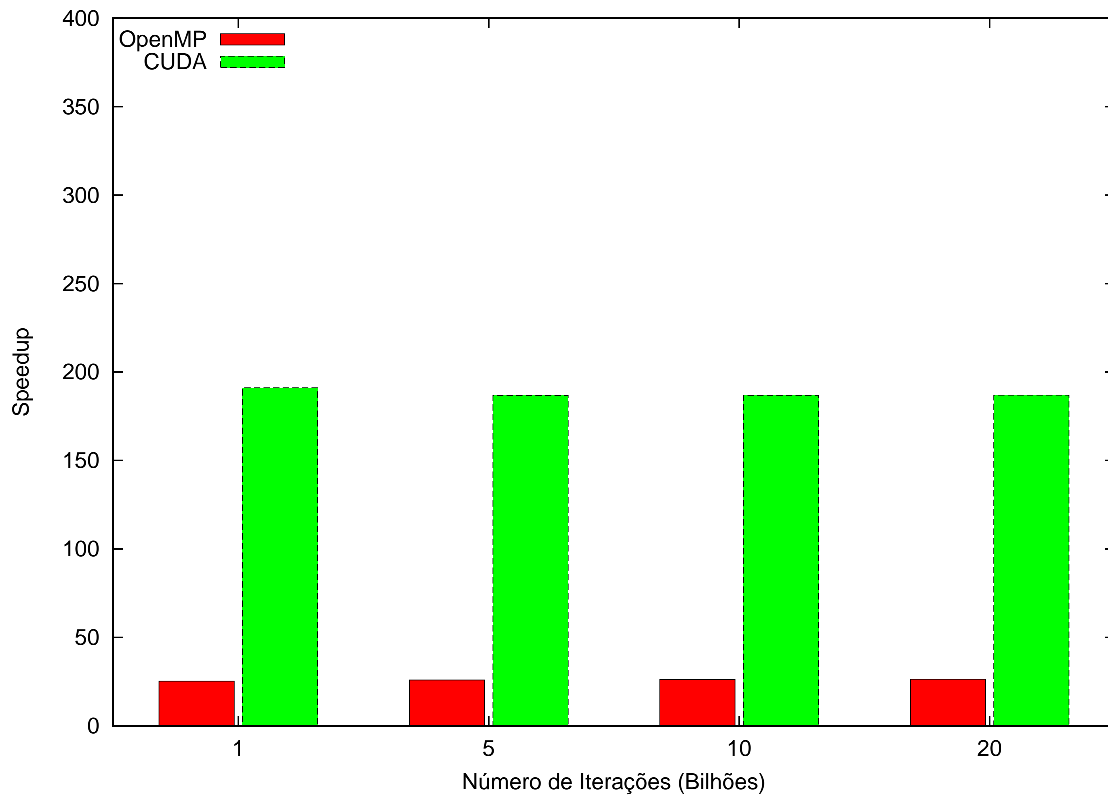


Figura 37 – Comparação entre Processador e Placa Gráfica - Quadratura de Gauss

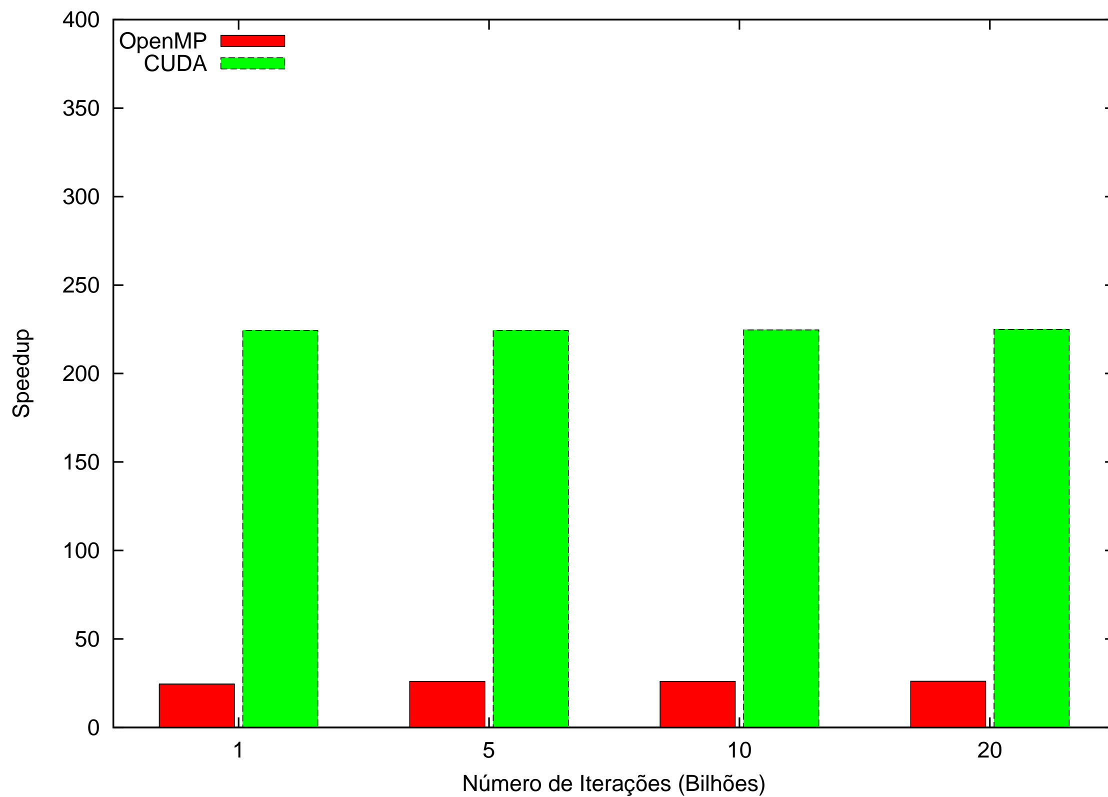


Figura 38 – Comparação entre Processador e Placa Gráfica - Regra dos Trapézios

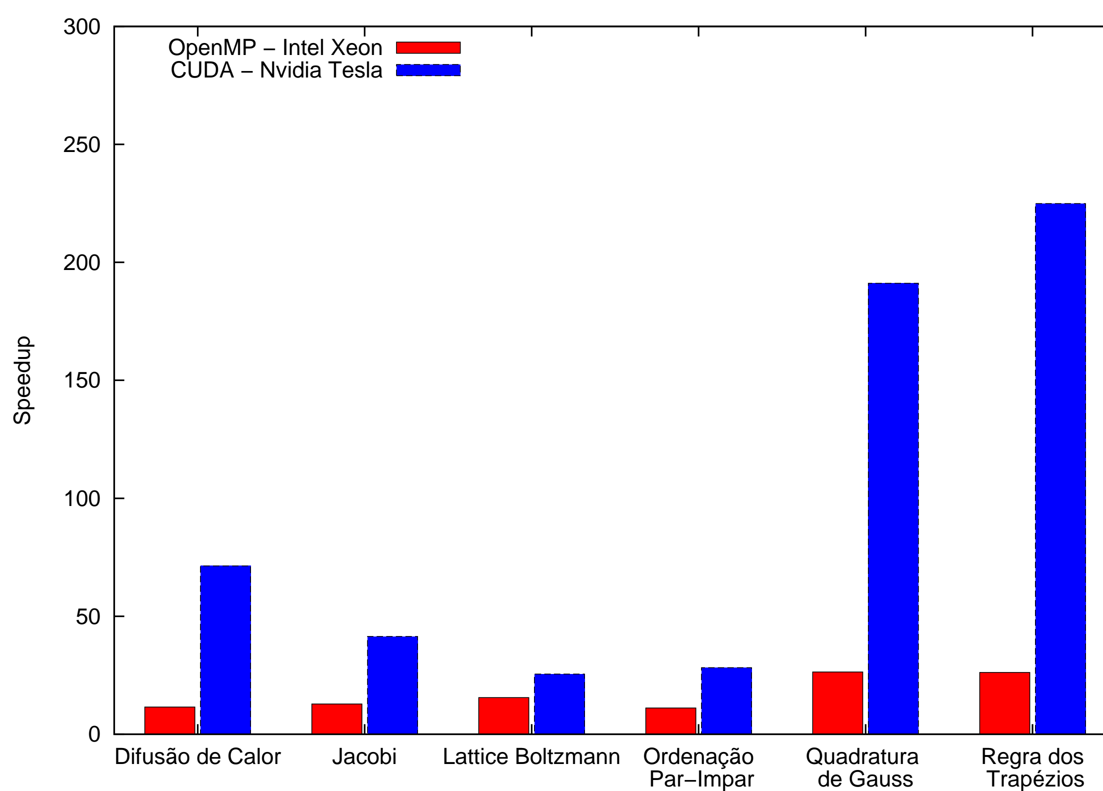


Figura 39 – Speedup do conjunto de aplicações em Processador e Placa Gráfica

5 Conclusões

Este trabalho realizou um estudo sobre o desenvolvimento de aplicações para processadores utilizando OpenMP e para placas gráficas GPUs com CUDA. Para isso, estudamos ambas bibliotecas (IPPs) e com isso, desenvolvemos um conjunto de aplicações científicas que possuem características diferentes: A difusão do calor, modelo matemático da equação do calor. O método de Jacobi para solução de sistemas de equações lineares. O método de Lattice Boltzmann para simulação de processos físicos. A ordenação Par-Impar, algoritmo de ordenação de valores de um vetor. As integrações pela Quadratura de Gauss e pela Regra dos Trapézios para cálculo de área de entre funções.

As análises realizadas sobre o desempenho do processador e da placa gráfica mostram que o conjunto de aplicações executa em menos tempo na GPU. No processador foi possível verificar que o desempenho aumenta linearmente quando o número de *threads* é aumentado. Em relação ao desempenho das placas gráficas, notou-se que o menor número de *threads* por bloco otimizou o desempenho da GPU, pois, dessa forma existiam mais blocos para se escalonar. Outro ponto a destacar-se é a escalabilidade das aplicações em ambas arquiteturas quando o tamanho da entrada cresce, sendo que foram feitas variações de 15 segundos à 15 minutos de execução.

Ao fim deste trabalho, identificamos oportunidades de trabalhos futuros. As quais incluem as seguintes expansões: número de aplicações; bibliotecas de programação (IPPs); arquiteturas paralelas embarcadas; dentre outras.

Referências

AMES, W. F. *Numerical Methods for Partial Differential Equations*. [S.l.]: Academic press, 2014. Citado na página 33.

BAILEY, D. H. et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, SAGE Publications, v. 5, n. 3, p. 63–73, 1991. Citado na página 33.

BATCHELOR, G. K. *An Introduction to Fluid Dynamics*. [S.l.]: Cambridge University Press, 2000. Citado na página 21.

BELL, G.; GRAY, J. What's next in high-performance computing? *Communications of the ACM*, ACM, v. 45, n. 2, p. 91–95, 2002. Citado na página 23.

BIENIA, C. et al. The parsec benchmark suite: Characterization and architectural implications. In: ACM. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. [S.l.], 2008. p. 72–81. Citado na página 33.

BIFERALE, L. et al. Second-order closure in stratified turbulence: Simulations and modeling of bulk and entrainment regions. *Physical Review E*, APS, v. 84, n. 1, p. 016305, 2011. Citado na página 36.

BROWN, J. W.; CHURCHILL, R. V. Fourier Series and Boundary Value Problems. *AMC*, v. 10, n. 12, 2012. Citado na página 33.

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: Portable Shared Memory Parallel Programming*. [S.l.]: MIT press, 2008. v. 10. Citado na página 27.

CHAPRA, S. C.; CANALE, R. P. *Numerical Methods for Engineers*. [S.l.]: McGraw-Hill, 2012. v. 2. Citado 2 vezes nas páginas 35 e 39.

CHEN, S.; DOOLEN, G. D. Lattice Boltzmann Method for Fluid Flows. *Annual Review of Fluid Mechanics*, 1998. Citado na página 36.

CLARKE, D. et al. Fupermod: a software tool for the optimization of data-parallel applications on heterogeneous platforms. *The Journal of Supercomputing*, Springer, v. 69, n. 1, p. 61–69, 2014. Citado na página 21.

COOK, S. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. [S.l.]: Newnes, 2012. Citado na página 24.

DIAZ, J.; MUNOZ-CARO, C.; NINO, A. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *Parallel and Distributed Systems, IEEE Transactions on*, 2012. Citado na página 27.

FLYNN, M. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, v. 100, n. 9, p. 948–960, 1972. Citado na página 23.

FOSTER, I. *Designing and building parallel programs*. [S.l.]: Addison Wesley Publishing Company, 1995. Citado na página 23.

GAUTIER, T. et al. Xkaapi: A runtime system for data-flow task programming on heterogeneous architecturesXKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In: IEEE. *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. [S.l.], 2013. p. 1299–1308. Citado na página 31.

GOYAL, P.; KAUR, N. An optimizing technique based on genetic algorithm for power management in heterogeneous multi-tier web clusters. *International Journal of Computer Applications*, Foundation of Computer Science, v. 115, n. 17, 2015. Citado na página 21.

HABERMAN, R. Applied Partial Differential Equations: With Fourier Series and Boundary Value Problems. *AMC*, v. 10, p. 12, 2013. Citado na página 33.

HONG, S.; KIM, H. An integrated GPU power and performance model. In: ACM. *ACM SIGARCH Computer Architecture News*. [S.l.], 2010. v. 38, n. 3, p. 280–289. Citado na página 30.

INTEL. Intel Core i7-3970X Processor Extreme Edition. In: . [s.n.], 2015. Disponível em: <http://http://ark.intel.com/products/70845/Intel-Core-i7-3970X-Processor-Extreme-Edition-15M-Cache-up-to-4_00-GHz>. Citado na página 21.

JR, B. R. D. et al. Applying Hyperthreading Technology for Evaluating the Performance of HTTP Server for Stored Audio/Video Retrieval. In: IEEE. *Emerging Trends in Engineering and Technology (ICETET), 2009 2nd International Conference on*. [S.l.], 2009. p. 644–647. Citado na página 50.

KALEEM, R. et al. Adaptive heterogeneous scheduling for integrated GPUs. In: ACM. *Proceedings of the 23rd international conference on Parallel architectures and compilation*. [S.l.], 2014. p. 151–162. Citado na página 30.

KIRK, D. B.; WEN-MEI, W. H. *Programming Massively Parallel Processors: A Hands-on Approach*. [S.l.]: Newnes, 2012. Citado 4 vezes nas páginas 24, 25, 26 e 29.

KOTSIANTIS, S.; KANELLOPOULOS, D. Discretization techniques: A recent survey. *GESTS International Transactions on Computer Science and Engineering*, v. 32, n. 1, p. 47–58, 2006. Citado na página 33.

KYRIAZIS, G. Heterogeneous system architecture: A technical review. *AMD Fusion Developer Summit*, 2012. Citado na página 21.

LASHUK, I. et al. A massively parallel adaptive fast multipole method on heterogeneous architectures. *Communications of the ACM*, ACM, v. 55, n. 5, p. 101–109, 2012. Citado 2 vezes nas páginas 31 e 41.

LIMA, J. V. F. *A Runtime System for Data-Flow Task Programming on Multicore Architectures with Accelerators*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, 2014. Citado na página 25.

NAVARRO, C. A.; HITSCHFELD-KAHLER, N.; MATEU, L. A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures. *Communications in Computational Physics*, 2014. Citado na página 23.

NVIDIA. CUDA programming guide. 2007. Citado na página 25.

NVIDIA. *Profiler, NVIDIA Visual*. [S.l.]: May, 2011. Citado na página 41.

NVIDIA. *Tesla C2075 Computing Processor Board*. 2011. Disponível em: <http://www.nvidia.com.br/docs/IO/43395/BD-05880-001_v02.pdf>. Citado na página 26.

PRESS, W. H. *Numerical recipes 3rd edition: The art of scientific computing*. [S.l.]: Cambridge university press, 2007. Citado na página 35.

ROSE, C. D.; NAVAUX, P. Fundamentos de processamento de alto desempenho. *Anais: 2a Escola Regional de Alto Desempenho*, p. 3–29, 2002. Citado 3 vezes nas páginas 23, 24 e 41.

SANDERS, J.; KANDROT, E. *CUDA by example: an introduction to general-purpose GPU programming*. [S.l.]: Addison-Wesley Professional, 2010. Citado na página 29.

SCHEPKE, C. *Distribuição de dados para implementações paralelas do Método Lattice Boltzmann*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, Porto Alegre, 2007. Citado na página 38.

SERPA, M.; SCHEPKE, C.; LIMA, J. V. F. Avaliação de desempenho do método de lattice boltzmann em arquiteturas multi-core e many-core. In: *Simpósio de Sistemas Computacionais de Alto Desempenho - WSCAD-WIC*. [s.n.], 2015. Disponível em: <http://XXXXX/148049.pdf>. Citado na página 22.

TREFETHEN, L. N. Is gauss quadrature better than clenshaw-curtis? *SIAM review*, SIAM, v. 50, n. 1, p. 67–87, 2008. Citado na página 39.

Índice

IPP, 21, 23, 25–27, 41, 49, 50, 63

CPU, 9, 11, 21
CUDA, 9, 13, 15, 25, 26, 29, 30, 35, 37–39, 43, 45–47, 50, 51, 63

DFC, 36
DRAM, 26

GB, 26
GDDR, 26
GPU, 9, 11, 13, 21, 24–26, 28–31, 34, 37, 39–41, 49–51, 63

CAD, 21, 23, 31

MDF, 13, 33, 34
MIMD, 13, 23, 24
MLB, 13, 36–38

OpenMP, 9, 13, 27, 28, 33–39, 42, 44–46, 49–51, 63

Xeon Phi, 24, 28

SIMD, 13, 23, 24
SM, 26, 40
SP, 26, 40

TDP, 21

ULA, 25