CENTRO UNIVERSITÁRIO VILA VELHA

CURSO DE CIÊNCIA DA COMPUTAÇÃO

Kassiane de Almeida Pretti Rocha

Luciano José Varejão Fassarella Filho

INTRODUÇÃO AO CUDA UTILIZANDO MÉTODOS NUMÉRICOS

VILA VELHA

2010

Kassiane de Almeida Pretti Rocha
Luciano José Varejão Fassarella Filho

INTRODUÇÃO AO CUDA UTILIZANDO MÉTODOS NUMÉRICOS

Trabalho de Conclusão de Curso apresentado ao Centro Univertário Vila Velha como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.
Orientador: Leonardo Muniz de Lima

VILA VELHA

2010

Kassiane de Almeida Pretti Rocha
Luciano José Varejão Fassarella Filho

INTRODUÇÃO AO CUDA UTILIZANDO MÉTODOS NUMÉRICOS

BANCA EXAMINADORA

Prof. Msc. Leonardo Muniz de Lima
Centro Universitário Vila Velha
Orientador

Prof. Msc. Vinícius Rosalen da Silva
Centro Universitário Vila Velha

Prof. Msc. Ester Maria Klippel
Centro Universitário Vila Velha

Trabalho de Conclusão de Curso

aprovado em 25/11/2010.

"Nós, Kassiane de Almeida Pretti Rocha e Luciano José Varejão Fassarella Filho, autorizamos que a UVV, sem ônus, promova a publicação da nossa monografia em página própria na Internet ou outro meio de divulgação de trabalho científico."

Data: 25/11/2010

Kassiane de Almeida Pretti Rocha
Centro Universitário Vila Velha

Luciano José Varejão Fassarella Filho
Centro Universitário Vila Velha

Aos nossos pais, familiares e amigos...

AGRADECIMENTOS

Agradecemos primeiramente a Deus por ter nos dado a força e a perseverança necessárias para nos mantermos firmes na direção do nosso objetivo.

Agradecemos aos nosso pais que sempre acreditaram e nos apoiaram em todos os momentos.

Agradecemos aos nossos familiares e amigos que, direta ou indiretamente nos ajudaram a transformar esse sonho em realizada.

Agradecemos aos nossos amigos de classe e professores que fizeram parte de nossas vidas durante esses quatro anos.

Agradecemos ao nosso orientador Leonardo Muniz Lima pela oportunidade de desenvolver este trabalho e por toda confiança e dedicação depositadas em nós.

Enfim, queremos deixar o nosso "Muito Obrigado".

“*Penso noventa e nove vezes e nada descubro; deixo de pensar, mergulho em profundo silêncio - e eis que a verdade se me revela.*”

Albert Einstein

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE SIGLAS

ANTIC	Alpha-Numeric Television Interface Controller
API	Interface de Programação de Aplicações
BLAS	Basic Linear Algebra Subroutimes
CAD	Computação de Alto Desempenho
CAS	Academia Chinesa de Ciência
CCOE	Centro de Excelência CUDA
CPU	Unidade Central de Processamento
CT	Centro Tecnológico
CUBLAS	Compute Unified Basic Linear Algebra Subprograms
CUDA	Compute Unified Device Architecture
CUFFT	Compute Unified Fast Fourier Transform
DRAM	Dynamic random access memory
EDG	Edison Design Group
FFT	Fast Fourier Transform
GCC	GNU Compiler Collection
GFLOPS	Giga Operações de Ponto Flutuante por Segundo
GMRES	Método do Resíduo Mínimo Generalizado
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
IPE	Instituto de Engenharia de Processo
LCAD	Laborátorio de Computação de Alto Desempenho
LCD	Direções Conjugadas à Esquerda
MGC	Método do Gradiente Conjugado
MIMD	Multiple Instructions Multiple Data
MISD	Multiple Instruction Single Data
MPI	Message Passing Interface
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PCI	Peripheral Component Interconnect
PNN	Probabilistic Neural Network
PTX	Parallel ThreadXexcution
RAM	Random Access Memory
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SISD	Single Instruction Single Data
SPC	Secure Copy

SSH	Secure Shell
UFES	Universidade Federal do Espírito Santo

SUMÁRIO

RESUMO

No presente trabalho é realizada uma breve contextualização sobre alguns conceitos da Computação de Alto Desempenho (CAD), dos fundamentos da Computação Paralela e da Unidade de Processamento Gráfico (GPU) com o propósito de fornecer uma base teórica para o estudo da tecnologia CUDA, *Compute Unified Device Architecture*. Sobre esta nova tecnologia será realizado um estudo dos seus principais conceitos de arquitetura e programação. Para demonstrar o alto desempenho da utilização de GPUs NVIDIA com a tecnologia CUDA é apresentada a implementação das duas operações básicas para a resolução de Sistemas Lineares Iterativos Não-Estacionários, Produto Interno e Produto Matriz Vetor. Por fim são apresentados testes de desempenho medindo *speed-up*, eficiência das operações e uma projeção da paralelização dessas operações no Método dos Gradientes Conjugados.

Palavras-chave: CAD (Computação de Alto Desempenho), Computação Paralela, GPU (Unidade de Processamento Gráfico), CUDA (*Compute Unified Device Architecture*), Métodos Numéricos

1 INTRODUÇÃO

Desde os primórdios da história da humanidade, a velocidade na realização dos cálculos sempre foi um ponto preocupante, principalmente em situações, nas quais há a necessidade de obtensão dos resultados em tempo hábil. Assim, para facilitar na contagem o homem começou a construir ferramentas simples como o ábaco e conforme a necessidade de cálculo aumentava as ferramentas começaram a ser aperfeiçoar até chegar nas máquinas chamadas de calculadoras. Mas como a necessidade e a busca da diminuição no tempo de obtensão das respostas cresceu de forma rápida, essas máquinas se tornaram obsoletas, por não conseguirem suprir a nova necessidade de cálculos, sendo utilizadas apenas para pequenas operações.

A necessidade de realizar cálculos tem crescido, no decorrer dos anos, mais rápido do que a tecnologia consegue suprir. Computadores foram lançados, modificados, incrementados, mas jamais abastaceram a necessidade de cálculo de todas as áreas. Computadores sequenciais - que tratam instruções em série, uma após a outra - já encontram limitaçãos físicas e financeiras para o aumento da capacidade de processamento, por exemplo, problemas de aquecimento e preço dos componentes. Como aumentar a velocidade do computador tornou-se uma solução cheia de limitadores, passou-se a utilizar a ideia de dividir e conquistar para alcançar mais capacidade de processamento. Assim, os problemas passaram a ser divididos entre vários computadores, que trabalham juntos para o processamento dos dados do problema.

A maioria dos problemas computacionais podem ser divididos em partes menores que são independentes uma das outras, por isso estas partes podem ser executadas ao mesmo tempo, isto é, em paralelo. Este fato fez com que novas arquiteturas computacionais que pudessem executar partes de um mesmo problema em paralelo começassem a ser estudadas e projetadas. Surge então a **Computação Paralela**.

Diferentes formas de arquiteturas paralelas tem sido estudadas através dos anos, sempre buscando o aumento da eficiência e a diminuição do custo. Dentre as op-

ções que existem atualmente tem-se desde supercomputadores a *clusters*. A partir da década de 90 a distinção entre supercomputadores e "computadores de mesa" se tornou uma questão de difícil compreensão, surgindo assim o termo **Computação de Alto Desempenho (CAD)**. Este termo, então passou a ser muito utilizado para caracterizar o uso de recursos computacionais para a obtenção de maior poder de processamento. Mas a definição do CAD não fica restrita apenas a supercomputadores e *clusters*, abrangendo também a tecnologia de rede e algoritmos de otimização [20]. Empulsionados pela necessidade crescente e o surgimento de um novo ramo da ciência focado na busca de poder de processamento, os pesquisadores começaram a buscar novas opções de dispositivos para serem utilizados nas novas arquiteturas paralelas.

Uma dessas novas opções de dispositivos é a **Unidade de Processamento Gráfico (GPU)**. Primeiramente criada apenas para processamentos gráficos - renderização de imagens, cálculos vetoriais, entre outros, a GPU adquiriu um grande poder de processamento e a possibilidade de ser programável. Sua enorme quantidade de núcleos de processamento chamou a atenção de estudiosos do CAD e a partir deste ponto, iniciou-se um estudo sobre a possibilidade da utilização da GPU no processamento de dados gerais, isto é, o foco do processamento não seria mais dados gráficos.

Como no início apenas APIs Gráficas, por exemplo o OpenGL, poderiam ser utilizadas para a programação na GPU, houve uma grande dificuldade na criação de programas que não fossem gráficos devido a falta de abstração do dispositivo, forçando o programador a ter conhecimento da arquitetura e funcionamento da unidade de processamento. Este fato mudou quando em 2006 foi lançada uma nova arquitetura de abstração, CUDA (*Compute Unified Device Architecture*) pela NVIDIA. Esta nova arquitetura possibilitou o uso em conjunto da CPU (Unidade Central de Processamento) e GPU para o processamento de aplicações de propósito geral, sem que o programador necessitasse do entendimento do processamento gráfico.

Este trabalho se propõe a mostrar a utilização da GPU para processamento de aplicações focadas em métodos numéricos com o uso da arquitetura CUDA utilizando a linguagem de programação CUDA+C que mescla instruções próprias desta arquitetura com algumas diretivas da linguagem C. Para realizar este tipo de análise é preciso apresentar um embasamento teórico de conceitos importantes para o entendimento da execução e comportamento de algorimo CUDA+C, desta forma no capítulo 3 é apresentado o ramo da Ciência da Computação responsável pelas pesquisas re-

lacionadas a ganho de desempenho, o CAD. Dando sequência, no capítulo 4 serão descritos algumas taxonomias da Programação Paralela utilizadas para classificar as arquiteturas. No capítulo 5 é apresentado o histórico e a arquitetura da GPU. Após todos esses conceitos importantes nos capítulos 6 e 7 serão descritas a arquitetura CUDA e a utilização da arquitetura em conjunto com a linguagem C. No capítulo 8 será realizado um estudo aprofundado sobre os códigos desenvolvindos em CUDA+C deste trabalho, além de fornecer um embasamento teórico de Sistemas Lineares. E por fim, no capítulo 9, será realizada a análise de desempenho dos algoritmos apresentados no capítulo 8, a partir dessa comparação, será realizada uma projeção de paralelização do algoritmo do Método de Resolução de Sistemas Lineares Iterativos Não-Estacionários, o Método dos Grandientes Conjugados.

2 OBJETIVOS

Este trabalho apresenta vários objetivos que têm como ponto comum o estudo da nova tecnologia da NVIDIA, o CUDA, que utiliza a GPU no processamento de programas de propósito geral. Segue abaixo a lista dos objetivos da realização deste trabalho:

- Realizar testes de desempenho das operações básicas para a resolução de Sistemas Lineares Iterativos Não-Estacionários, Produto Interno e Produto Matriz Vetor;
- Realizar a projeção da paralelização das operações básicas no Método dos Gradientes Conjugados;

3 CAD - Computação de Alto Desempenho

A preocupação com a crescente necessidade de grande poder de processamento apareceu no começo da década de 60. Foi nessa época que o primeiro supercomputador, CDC600 criado pela IBM, foi lançado [20]. A partir desse momento, novas tecnologias foram pesquisadas, criadas e misturadas a fim de suprir esta crescente demanda por poder de processamento. Desta pesquisa por novas tecnologias e meios, surge um novo ramo da Ciência da Computação, a Computação de Alto Desempenho (CAD). A Computação de Alto Desempenho agrupa:

> [...] uma coleção de sistemas de hardware, ferramentas de software, linguagens e abordagens genéricas de programação, que tornam possíveis, a um preço apropriado, aplicações antes infactíveis. [20]

Como a própria definição diz, uma parte do foco da Computação de Alto Desempenho é a construção de supercomputadores, com alto poder de processamento, que se utilizam apenas de tecnologias disponíveis. Atualmente, esses supercomputadores são utilizados e muitas vezes essenciais em várias áreas da ciência, engenharia e indústria. Em alguns casos, cálculos de pesquisas, como por exemplo, na área de ciência moderna, clima e nanotecnologia, por serem complexos ou conterem uma grande quantidade de dados, só podem ser executados em supercomputadores paralelos [18].

Além dos supercomputadores, há outras tecnologias que também são estudadas pelo CAD. Uma dessas tecnologias é a bem difundida e barata *cluster*. De acordo com [17], *clusters* tem como definição ser um conjunto de "nós" integrados (conectados) e independentes, os quais são um sistema por si só capazes de realizar operações individualmente. Um grande representante dos *clusters*, que popularizou a construção de sistemas paralelos, é o conhecido *Beowulf-class cluster*. *Clusters* deste estilo

utilizam componentes livres e populares, tanto de hardware quanto de software, para encontrar um melhor custo. Além disso, tem como objetivo não possuir ou possuir pouca dependência de fornecedores.

Há dois tipos de *clusters* quanto o seu nível de paralelismo, os *clusters-NOW* e os *costellations*. Os *clusters-NOW* possuem mais nós do que microprocessadores em cada um dos nós. Já os *costellation* possuem mais microprocessadores em cada nó do que nós em todo o *cluster*. Essa diferença na arquitetura pode afetar e muito o modo como a programação deste *cluster* é feita. Um exemplo é a utilização, quase em sua totalidade, do *Message Passing Interface* (MPI) para a programação dos *clusters-NOW* [17]. O MPI pode ser definida como:

> [...] uma biblioteca que fornece todas as funcionalidades básicas para a comunicação dos processos, ou seja, os processos se comunicam chamando rotinas de sua biblioteca para enviar e receber mensagens. O MPI é formado por conjuntos de processos onde cada processo é criado para um processador; possui paralelismo explicito - o programador é responsável pela maior parte do esforço de paralelização, ao contrário do paralelismo implícito onde a aplicação é paralelizada sem a intervenção do programador. Sua funcionalidade consiste na subdivisão dos problemas em pequenas partes que são enviadas aos processadores, depois esses pequenos resultados são agrupados constituindo assim o resultado final. Existem duas principais distribuições livres dessa biblioteca, sendo elas LAM-MPI3 e MPICH4. [28]

Já os *costellations* são, geralmente, utilizados com *Open Multi-Processing* (OpenMP). Esta API (*Application Program Interface*) pode ser descrita como:

> [...] usada para programação multi-thread em ambientes de memória-compartilhada para plataformas UNIX e Microsoft Windows NT. Aplicações que usam OpenMp são portáveis pois são especificadas para C/C++ e Fortran. Esta API provê diretivas de compilador e rotinas de bibliotecas e variáveis, embutidas no código-fonte C/C++ e Fortran, para escopo de dados, especificação e compartilhamento da carga de trabalho e criação e sincronização de threads. Também oferece chamadas de funções para setar e obter informações sobre as threads. Algumas variáveis ajudam a controlar o comportamento do programa paralelo em tempo de execução.[14]

Apesar de mais caros, os supercomputadores tendem a ser mais eficientes, já que não necessitam de uma perda de tempo com a comunicação entre nós, como é o

caso dos *clusters*. Novas pesquisas por tecnologias acabaram por apontar um outro dispositivo a ser inserido em supercomputadores, a Unidade de Processamento Gráfico (GPU), que é o foco deste trabalho. Atualmente, este dispositivo já é utilizado em supercomputadores, incluindo o segundo supercomputador mais poderoso do mundo comercialmente disponível, de acordo com o TOP500 [38].

4 Fundamentos da Computação Paralela

O processamento paralelo pode ser definido como:

> As várias unidades ativas cooperam para resolver o mesmo problema, atacando cada uma delas uma parte do trabalho e se comunicando para a troca de resultados intermediários ou no mínimo para a divisão inicial do trabalho e para a junção final dos resultados [...] [16]

Este capítulo tem como objetivo a introdução de conceitos básicos da Programação Paralela. Descreve duas das principais formas de classificações (taxonomia) das arquiteturas paralelas, a classificação segundo níveis de paralelimso e a classificação de FLYNN. Será também identificado em qual seção destas classificações a paralelização por meio da GPU se encontra.

4.1 Classificação segundo níveis de paralelismo

A Classificação segundo níveis de paralelismo, ou classificação por Granulosidade, tem como parâmetro o tamanho das sub-tarefas que podem ser executadas em paralelo ou a quantidade de trabalho entre as iterações dos processadores. Esta classificação divide as arquiteturas paralelas em três partes: as que possuem granulosidade fina, média e grossa [28].

A granulosidade fina tem como característica a paralelização em nível de instruções, ou seja, estágios das instruções são executados em paralelo. Esse tipo de paralelismo pode ser conseguido através de um ou vários *pipelines*, que são instruções divididas em conjuntos de estágios que equivalem à tarefa total e que são executadas com concorrência de tempo [28].

A granulosidade média acontece em nível de *thread*. É constituída de vários processos médios nos quais as unidades de paralelismo são blocos e sub-rotinas [28]. É neste nível que o paralelismo utilizando a GPU mais CUDA se encontra. A GPU possui vários pequenos núcleos de processamento, que quando utilizados através da API CUDA, ficam responsáveis por executar cada uma das *threads* geradas em uma única chamada do programa principal. Nos capítulos 5, 6 e 7 este fato será melhor explicado.

O último nível é a granulosidade grossa. Essa tem como nível de paralelismo o próprio processo, ou seja, é o paralelismo conseguido através da utilização de vários processadores. Este nível é encontrado em Multiprocessadores ou em Multicomputadores. A utilização da GPU mais CUDA não é dita como granulosidade grossa, pois todos os núcleos de processamento são responsáveis por executar *threads* de um mesmo processo.

4.2 Classificação de FLYNN

A classificação de FLYNN surgiu em meados dos anos 70 e ainda continua válida e difundida. Esta classificação se baseia no fato do computador executar um sequência de instruções sobre uma sequência de dados, ou seja, há o fluxo de instruções e o fluxo de dados. Cada fluxo, para a classificação FLYNN, possui duas possibilidades, múltiplos ou não. Através da combinação dessas possibilidades, FLYNN propôs quatro classes, como mostra a figura 1 [16] [28].

	SD (*Single Data*)	**MD** (*Multiple Data*)
SI (*Single Instruction*)	SISD Máquinas von Neumann convencionais	SIMD Máquinas *Array* (CM-2, MasPar ,Cuda)
MI (*Multiple Instruction*)	MISD Sem representante	MIMD Multiprocessadores e multicomputadores (nCUBE, Intel Paragon, Cray T3D)

Figura 1: As diferentes classificações do modelo FLYNN [16].

A classe *Single Instruction Single Data* (SISD), possui apenas um fluxo de instruções que atuam apenas a um único fluxo de dados, sendo executada uma operação de cada vez [16] [28].

A classe *Multiple Instruction Single Data* (MISD) é um ponto de discordância entre alguns autores. Por se ter vários fluxos de instruções executando a um único fluxo de dados, alguns autores afirmam que esta classe é tecnicamente impraticável, já que seria necessária a utilização de uma mesma posição de memória. Outros autores, encaram máquinas *Pipeline* como sendo MISD [16] [28].

As máquinas paralelas se encontram nas duas classes restantes, a *Single Instruction Multiple Data* (SIMD) e a *Multiple Instructions Multiple Data* (MIMD). A SIMD possui uma única instrução que é executada ao mesmo tempo sobre múltiplos dados. Ou seja, todos os seus processadores executam uma mesma instrução sobre diferentes fluxos dados. Como será visto nos capítulos 5, 6 e 7, a execução de um programa em uma GPU através da API CUDA se assemelha à forma de execução desta classificação.

Por fim, a classe MIMD é capaz de receber vários fluxos de instruções (programas) para executar sobre diferentes fluxos de dados. Cada unidade de controle e unidade de processamento desta máquina recebe um programa distinto, esse será executado sobre um fluxo de dados próprio. Dessa forma os programas são executados de forma assíncrona [16].

5 GPU - *Graphics Processing Unit*

A Unidade de Processamento Gráfico, ou em inglês *Graphics Processing Unit* (GPU), é um microprocessador especializado em processamento gráfico. É encontrada nos dias de hoje em computadores pessoais, estações de trabalho, videogames e até em celulares, como o *iPhone*. Foi criada principalmente com o objetivo de diminuir a utilização da CPU para processamentos gráficos, mas atualmente tem sido também utilizada para processamentos de dados gerais [30] [42].

Um dos primeiros *chips* produzido a possuir características semelhantes a uma GPU foi o *Alpha-Numeric Television Interface Controller* (**ANTIC**). Esse *chip* foi utilizado no videogame Atari 5200 e em microcontroladores e foi construído especialmente para *mapping* (mapeamento) [30] [42].

Outro dispositivo que compõe a era pré-GPU é o *Commodore Amiga*. Este foi criado primeiramente para ser um videogame, mas acabou sendo re-projetado para ser um computador pessoal. O *Amiga* carregava todas as funções de gerações de vídeo para o hardware e ainda foi o primeiro a conter um *blitter* (*Block Image Transfer*). *Blitter* trata-se de um bloco lógico com rápida transferência de dados, utilizado na transferência de *bitmaps*. Possui a capacidade de enviar blocos ao invés de enviar bit a bit e trabalha paralelamente à CPU, deixando-a livre para executar outras tarefas [30] [42].

Com o crescimento do interesse por exibição gráfica e com a grande competitividade entre as empresas para ganhar este mercado, o setor de placas gráficas se desenvolveu de uma forma espetacular. Várias placas foram criadas e novas tecnologias foram aparecendo em suas GPUs a cada lançamento. Estas GPUs são divididas conforme sua arquitetura e a possibilidade de dinamismo de suas funcionalidades. Esta divisão resulta, até o momento, em quatro gerações de GPUs:

- Primeira: placas lançadas até o ano de 1998;

- Segunda: placas lançadas do ano de 1998 até o fim 1999;
- Terceira: placas lançadas do fim de 1999 até o começo de 2001;
- Quarta: placas lançadas do início de 2001 até os dias atuais.

A primeira geração eram GPUs criadas apenas para enviar dados ao monitor. A segunda geração possuía funcionalidades, antes feitas pela CPU, realizadas pela GPU. As principais eram: transformações geométricas e aplicação de iluminação. Ainda não era possível de se desenvolver programas para a GPU. Com a terceira geração, passa a ser possível construir programas a serem executados diretamente pela GPU. Por enquanto estes programas apenas poderiam ser *vertex shaders*, que são funções que rodam na GPU para processamento de vértices. As GPUs passam a ser vistas como hardware programável, mas ainda não havia uma linguagem voltada a este propósito, dificultando a vida do programador. A quarta e última geração possui dois pontos de implementação possíveis no *pipeline* gráfico. Além de funções *vertex shaders*, também é permitido funções *fragment shaders*, que são funções para processamento de fragmentos. Fragmentos são *pixels* da imagem final, mas com profundidades variadas.

Como dito anteriormente a GPU é um microprocessador especializado em processamento gráfico. E este processamento gráfico são operações gráficas, independentes uma da outra, de uma grande gama de dados. Ou seja, vários cálculos podem ser executados paralelamente sem que haja alterações no resultado final. Este fato fez com que a GPU se especializasse em barramentos, para transportar a grande quantidade de dados envolvida, e em paralelismo, para realizar os vários cálculos. Por exemplo:

> Uma GPU atual, como a NVIDIA GeForce GTX 280, possui duzentos e quarenta núcleos, portanto, são capazes de processar em paralelo o mesmo número de vértices ou fragmentos. Essa placa é provida de um grande barramento de dados para acesso rápido à memória de vídeo. O barramento possui 512 bits e largura de banda de 140 GBytes/s. A comunicação com a memória RAM (*Random Access Memory*) é feita através do barramento PCI express 2.0 x16, cuja largura de banda é de 8 GBytes/s [13].

A GPU utiliza muito mais transístores para processamento de dados do que para controle de fluxo, isso acontece pois o mesmo programa é executado por cada um dos

elementos de dados (ULAs), diferentemente da CPU. Uma outra grande diferença é a não necessidade de grandes memórias *caches*, pois, por ter vários elementos de dados e um alto poder de processamento, a latência da memória pode ser desprezada pela quantidade de cálculos que são realizados [6]. A Figura 2 representa as diferenças nas arquiteturas dos dois microprocessadores.

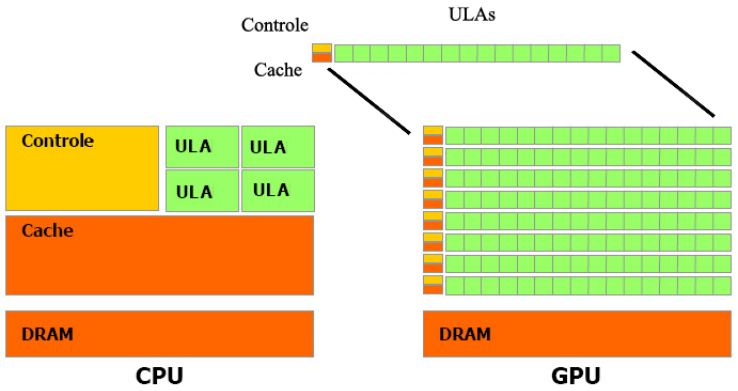


Figura 2: Comparação entre as arquiteturas da CPU e GPU. [6]

Além da grande diferença entre as arquiteturas dos dois dispositivos, a expansão do setor gráfico acarretou aumento ainda maior do poder de processamento das GPUs. A Figura 3 representa um comparativo entre os dois dispositivos e o pico da quantidade de cálculos de pontos flutuantes por segundo, em bilhões, (GFLOP/s) realizada por eles. No ano de 2003 a GPU já se mostrava mais poderosa do que a CPU e em 2005 esta diferença aumentou ainda mais. Realmente a GPU pode ser utilizada em conjunto com a CPU como dispositivo para processamento, ficando a CPU apenas com a tarefa de gerenciamento do fluxo de execução do programa. Mas a utilização dessas duas unidades em conjunto deve ser pensada, pois nem todos os algoritmos conseguirão atingir ou quase atingir o pico de cálculos realizados por segundo, o que pode acarretar, em vez de um ganho, uma perda de desempenho.

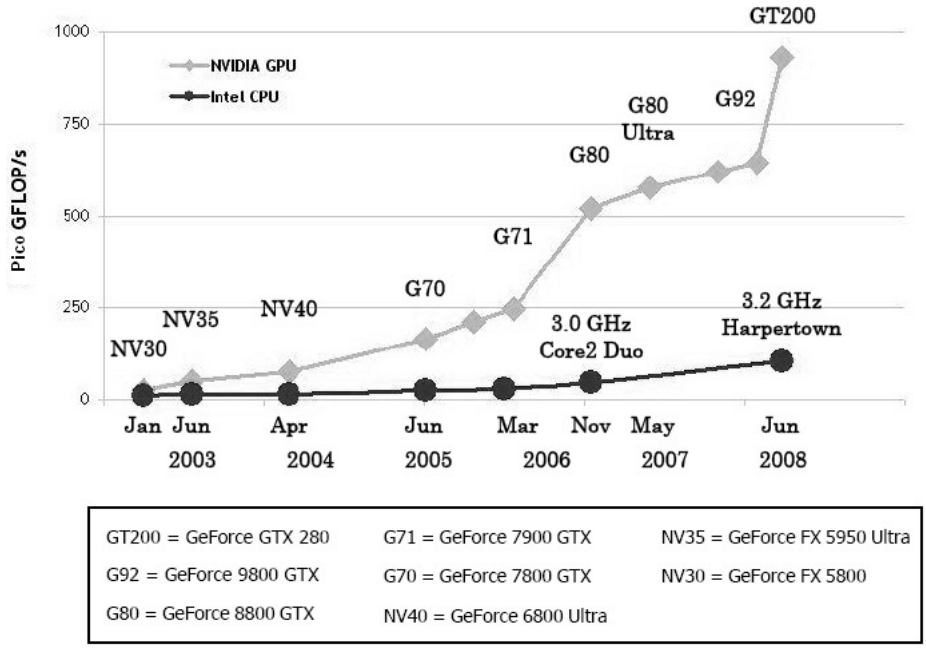


Figura 3: Gráfico de processamento. [6]

A partir da terceira geração, que já eram programáveis, começam a surgir programas não gráficos desenvolvidos para a GPU. Surge então a **GPGPU**, ou *General Purpose GPU*. Havia, porém, apenas APIs gráficas que permitiam a programação própria a GPU e sua utilização para a construção de programas de propósito geral era trabalhosa e possuía limitações. A expansão deste setor aconteceu com a quarta geração de placas e com a criação de linguagens próprias à esta programação.

A GPGPU é uma técnica de programação para construir programas que são tradicionalmente resolvidos pela CPU para serem executados pela GPU. Mas como dito anteriormente, nem todos os algoritmos terão um ganho de desempenho se resolvidos pela GPU. Para que isto aconteça, é necessário que eles possuam as seguintes características:

Volume de dados e processamento: aplicações gráficas trabalham com bilhões de *pixels* por segundo, cada *pixel* sendo submetido a centenas de operações aritméticas. Isso indica que GPUs precisam trabalhar com grandes volumes dados que exigem bom desempenho computacional de execução. Problemas cujo cálculo da solução é lento devido a alta complexidade de execução do algoritmo e não ao volume de dados por ele manipulado não são bons candidatos a serem mapeados para GPUs, pois possivelmente não serão capazes de utilizar eficientemente a alta capacidade de paralelização desse hardware;

Paralelismo: os vértices de uma cena de uma aplicação gráfica podem ser calculados independentemente. Da mesma maneira, o conjunto de dados da sua

aplicação deve possuir um alto nível de independência, para que os núcleos da GPU sejam utilizados eficientemente;
Escalabilidade: o programa aumenta seu desempenho automaticamente quando recebe mais núcleos de processamento;
Vazão: quando utilizamos *pipeline* para solucionar problemas, geralmente a latência na resolução de um problema se torna maior, porém mais problemas são solucionados em um determinado período de tempo. Da mesma maneira, problemas escritos para GPUs precisam priorizar a alta vazão, e não a baixa latência. [19]

Além dessas características, uma preocupação necessária é sobre a precisão dos dados. A maioria das GPUs segue o mesmo padrão de armazenamento de ponto flutuante utilizado nas CPUs, o **IEEE 754** [2], e a precisão numérica utilizada é de no máximo 32 bits. Mas diferentemente do armazenamento, estes dispositivos divergem na execução das operações. Em algumas GPUs nem todos os 32 bits são preenchidos conforme o padrão IEEE 754, após uma operação matemática. A divergência nos resultados aumenta ainda mais, caso esses valores sejam reutilizados como entradas de outras operações [29]. Além disso, a utilização de dupla precisão (utilização de todos os 32 bits) acarreta em uma perda de eficiência na execução do algoritmo. Por isso, caso o resultado do programa necessite de uma ótima precisão, a utilização da GPU para este processamento, talvez não seja a melhor indicada.

Atualmente, programas em GPGPU tem sido criados com a utilização de plataformas próprias a este tipo de programação. Estas plataformas são de mais fácil entendimento e associação se comparadas às API Gráficas por abstraírem questões relacionadas ao processamento gráfico. Entretanto, sua implementação ainda exige um maior esforço do programador, já que necessita de um entendimento quanto a arquitetura da GPU e a forma como estes programas são executados. Um exemplo de uma plataforma é a criação da NVIDIA, o CUDA (vide capítulo 6). Para possibilitar uma curva de aprendizagem dos programadores mais amena, esta plataforma disponibiliza bibliotecas (Biblioteca CUDA e CUDA *Runtime*), que contêm um conjunto de diretrizes de programação utilizadas em conjuntos com diferentes linguagens e interfaces (C, OpenCL, Fortran CUDA, *DiretictCompute*). Junto com estas bibliotecas, a plataforma disponibiliza um ambiente que possibilita aos programadores utilizar estas linguagens como se fossem de alto nível. Logo, não há a necessidade, inicialmente, de um estudo mais profundo do comportamento da comunicação da plataforma com os dispositivos (GPU e CPU).

A plataforma CUDA tem se mostrado eficiente e provocado grandes *speed-up* em várias tipos de aplicações. *Speed-up* é um termo utilizado pelo CAD para avaliar o ganho de desempenho de programas paralelos e é obtido pela divisão do tempo de execução serial pelo tempo de execução paralela. Para exemplificar esse ganho de desempenho, na Tabela 1, são apresentados os tempos de execução de um algoritmo de *Probabilistic Neural Network* (PNN) [41].

Tabela 1: Média dos tempos de categorização de um documento para os algoritmos sequencial e paralelos e o *Speed-up* ocasionado pela utilização da plataforma CUDA [41], onde |TV| é a entrada de dados, C(s) é o tempo de execução do algoritmo sequencial e C+CUDA(s) é o tempo de execução do algoritmo paralelo utilizando a plataforma CUDA

|TV|	C(s)	C+CUDA(s)	*Speed-up*
691	0,01963	0,00055	35,7
1382	0,03824	0,00085	45,1
2073	0,05684	0,00107	52,9
2764	0,07541	0,00128	58,8
3455	0,09398	0,00152	61,8
4146	0,11258	0,00174	64,8
4837	0,13117	0,00193	67,9
5528	0,14976	0,00216	69,4
6219	0,16833	0,00237	71,1
6910	0,18694	0,00259	72,3

> A Tabela 1 apresenta as médias dos tempos (em segundos) de 100 execuções do algoritmo PNN, sequencial e paralelo, para cada tamanho de $|TV|$ (usado para treinar e validar eventuais parâmetros do sistema). Ela também apresenta o *speed-up* alcançado pelo algoritmo paralelo executando na GPU. Como a Tabela 1 mostra, o *speed-up* cresce com o tamanho de $|TV|$. A nossa versão paralela do PNN chegou a 72 vezes mais rápido do que a seqüencial. [41]

Além de grandes *speed-ups* ocasionados, um outra vantagem encontrada é o interesse da NVIDIA, pelo desenvolvimento deste setor. Há várias documentações disponibilizadas, fóruns criados pela própria empresa para troca de conhecimento entre os programadores e principalmente a criação de placas especializadas nesta vertente. Por isso, no momento a escolha pela utilização plataforma CUDA para este tipo de desenvolvimento é vista como a mais correta.

Um dos últimos lançamentos dessa mesma empresa são as placas ***Tesla***. Estas placas foram lançadas principalmente para o mercado de GPGPU. São placas com

alto poder de processamento, maior precisão numérica, mais memória disponível, entre outras inovações. Suas últimas versões lançadas são a NVIDIA *Tesla™* C2050 e C2070. Estas últimas foram baseadas na nova arquitetura lançada chamada ***Fermi***, e possuem como principais inovações, um aumento da velocidade de cálculos com dupla precisão (32 bits), de acordo com a própria empresa, em 4 vezes (figura 4), 512 CUDA *cores*, a tecnologia *NVIDIA Parallel Data Cache*, dentre outras [10].

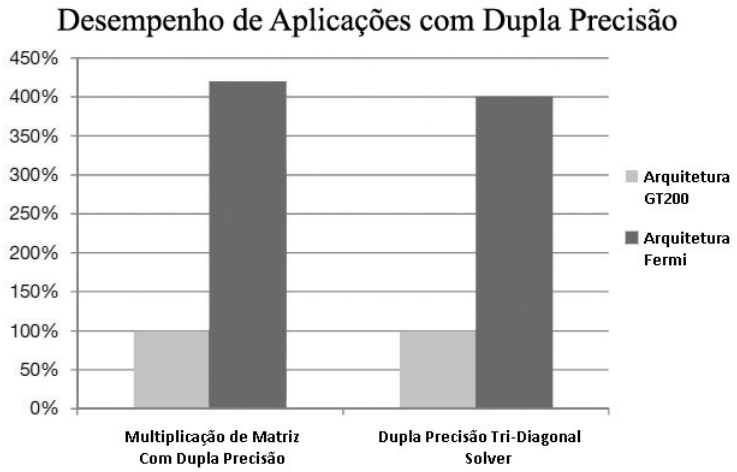


Figura 4: Gráfico de Dupla Precisão [10]

6 CUDA - *Compute Unified Device Architecture*

CUDA é uma arquitetura de abstração com um modelo de programação embutido, desenvolvida para suportar a computação estrita e altamente paralelizada [8]. Essa plataforma possibilita o ganho significativo de desempenho das GPUs, que tradicionalmente apresentam grande potencial de processamento, como apresentado na capítulo 5. Com essa nova plataforma o desafio de desenvolver uma aplicação de software capaz de transparentemente aumentar o seu paralelismo para otimizar a utilização do processador se tornou viável. Assim as aplicações aceleradas com CUDA passaram a permitir o máximo de paralelismo possível, dando suporte para o uso de vários processadores ao mesmo tempo [6].

Lançada oficialmente em novembro de 2006 pela ***NVIDIA***, fabricante de placas de vídeo, a plataforma CUDA, surgiu para dar suporte ao novo paradigma de computação, que utiliza em conjunto a CPU e GPU para o processamento de aplicações de propósito geral, possibilitando assim a utilização da GPU tanto para renderização de gráficos quanto para outros programas de fins genéricos. Assim o desenvolvimento de aplicações aceleradas com a plataforma CUDA, utiliza a CPU como processador central, isto é, o controle de fluxo da aplicação é de responsabilidade dessa unidade de processamento. Enquanto a GPU trabalha como co-processador, realizando o processamento paralelo dos cálculos necessários para a execução do programa. Esse trabalho cooperativo entre as unidades de processamento, cada qual, realizando uma tarefa focada nas suas melhores características mais a utilização da plataforma CUDA como interface de comunicação entre elas, resulta diretamente em um significativo ganho de desempenho [31].

Devido esse grande potencial promovido, vários setores que trabalham com tecnologias que necessitam de computação de alto desempenho, viram neste novo paradigma uma forma de ampliar e desenvolver aplicações de grande porte aceleradas

com a tecnologia CUDA. O alto grau de paralelismo, desta plataforma em algumas aplicações, transforma as GPUs em verdadeiros *clusters*. Diante das vantagens óbvias como economia de processador, menor gasto com máquinas, menor tempo de manutenção, menor tempo gasto com projetos, maior rendimento diário, dentre várias outras, a aceitação da plataforma, tanto no meio acadêmico quanto no comercial, foi consideravelmente rápida nos seus pouco mais de 4 anos oficialmente no mercado. Seguem abaixo algumas notícias da utilização da plataforma CUDA:

> O novo supercomputador instalado em Taiwan [...] tentará analisar mais de perto o comportamento das partículas subatômicas. [...] "Com o nosso sistema equipado com GPUs da NVIDIA, estamos entregando 15 teraflops, ao preço de US$ 200.000, isto é, 1% do custo de um supercomputador convencional como o IBM BlueGene / L."[...] Estes cálculos são possíveis graças à arquitetura de computação CUDA dos chips da NVIDIA. [34]

> O Kaspersky equipou o seu laboratório com alguns computadores Tesla S1070 com o objetivo de utilizar a tecnologia CUDA (que permite que programas comuns sejam executados pelo chip gráfico) para acelerar a verificação e identificação de novos vírus. De acordo com testes internos feitos pela empresa uma máquina Tesla S1070 é 360 mais rápida para rodar um algoritmo de verificação de vírus do que um computador equipado com o processador Core 2 Duo de 2,6 GHz. A empresa planeja integrar a tecnologia CUDA em seus produtos. [1]

> Com milhões de GPU's habilitadas para CUDA vendidas até o momento, desenvolvedores de software, cientistas e pesquisadores continuam achando os mais diversos usos para o CUDA, incluindo processamento de vídeos e imagens, biologia e química computacionais, simulação de dinâmica de fluídos, reconstrução de imagens de Tomografia Computadorizada, análise sísmica, traçado de raios e muito mais. Um indicador da adoção da tecnologia CUDA é a evolução da *GPU Tesla* para a computação via GPU. Atualmente, há mais de 700 *clusters* de GPUs instalados no mundo inteiro em empresas que fazem parte da *Fortune 500*, desde a *Schlumberger* e a *Chevron* no setor energético até o *BNP Paribas* no setor bancário. [12]

Para o desenvolvimento da tecnologia e desenvolvimento de aplicações avançadas com a tecnologia CUDA, foi criado, pela NVIDIA, o programa *Centro de Excelência CUDA* (CCOE) que reconhece, recompensa e incentiva a colaboração das Universidades na pesquisa de Computação Maciçamente Paralela. Com essa iniciativa a NVIDIA

espera capacitar pesquisadores acadêmicos para realizarem pesquisas que mudarão o mundo ao melhorar drasticamente o poder de computação disponível a cientistas e engenheiros; melhorar o estado da computação para um mundo de computação maciçamente paralela; estabelecer relações de pesquisa, educação e recrutamento com as principais instituições acadêmicas do mundo. Fazem parte da CCOE:
Instituto de Engenharia de Processo (IPE) na Academia Chinesa de Ciência (CAS), Universidade Nacional de Taiwan, Universidade de Tsinghua, Universidade de Cambridge, Universidade de Harvard, Universidade do Tennessee, Universidade de Harvard, Universidade de Illinois em Urbana-Champaign, Universidade de Maryland, Universidade de Utah[12].

Atualmente a plataforma se encontra na versão 3.0, lançada em março de 2010 para suportar a mais nova arquitetura da NVIDIA apelidada de ***Fermi***. Essa nova placa de vídeo fornece suporte para a arquitetura CUDA sendo uma das primeiras placas compostas por 3 bilhões de *transistores* [10].

Após o estudo das características da GPU no capítulo 5 e uma breve introdução a tecnologia CUDA, nas próximas seções serão listados os aspectos técnicos da plataforma como o modelo de programação escalável e as hierarquias de *thread* e memória. Tais aspectos fornecem a compreensão da importância dessa arquitetura na busca do ganho de desempenho das aplicações de problemas computacionais complexos.

6.1 API CUDA

O entendimento do funcionamento da plataforma CUDA é baseado em componentes de software e de hardware. A comunicação entre a CPU e GPU através dessa plataforma CUDA necessita de uma pilha de softwares que determina a hierarquia de compilação, conhecida como ***API CUDA***. A utilização de uma pilha de software viabiliza a abstração no desenvolvimentos de aplicações utilizando a plataforma. Desta forma a complexidade da comunicação entre CPU e GPU fica encapsulada na API. O modelo proposto visualiza a GPU como co-processador (*Device*) e a CPU como responsável pelo processamento principal (*Host*). Na figura 5 são representadas as camadas dessa pilha: *Biblioteca do CUDA, CUDA Runtime, CUDA Driver*. A camada de aplicações refere-se aos aplicativos que utilizam o modelo de programação CUDA que pode se comunicar com todos os níveis da pilha direta ou indiretamente.

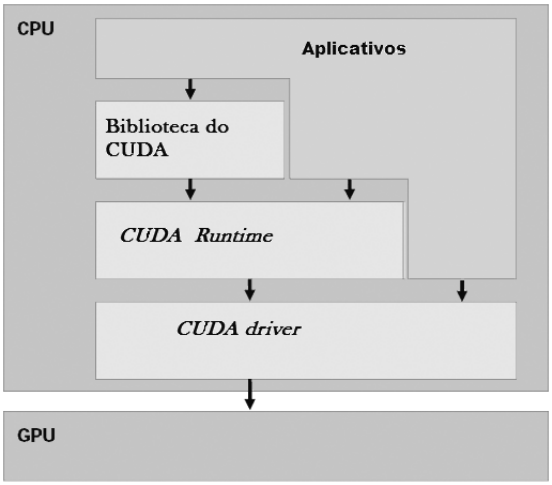


Figura 5: Camadas da pilha de API da arquitetura CUDA. [39]

O CUDA fornece suporte a diversas funções matemáticas, que são representadas por duas bibliotecas prontas: **CUBLAS** *(Basic Linear Algebra Subroutimes - BLAS)* e **CUFFT** *(Fast Fourier Transform - FFT)* presentes na camada de biblioteca do CUDA (vide figura 5).

CUBLAS e *CUFFT* são implementações da biblioteca *BLAS* e *FFT*, respectivamente, para dar suporte a NVIDIA CUDA *driver*. A funcionalidade principal do CUBLAS é permitir o acesso aos recursos computacionais das GPUs NVIDIA. Apresentando funções auxiliares para criar e destruir objetos de vetores e matrizes no espaço de memória da GPU, sendo estes utilizados para a gravação e recuperação de dados que serão transferidos para a CPU. A biblioteca é autossuficiente no seu nível da pilha, isto é, não é necessária nenhuma interação direta com o CUDA *driver* [5]. Já CUFFT utiliza a eficiência do algoritmo de divisão e conquista para algoritmos numéricos do *FFT* incorporando uma interface simples para computação paralela de FFTs em GPUs NVIDIA. Desta forma, permite o alavancamento do poder de ponto flutuante e paralelismo da GPU sem ter que desenvolver algum tipo de customização [7].

O *CUDA Runtime API* atua como um intermediário entre o desenvolvedor e o *driver*, de forma a facilitar a programação mascarando alguns detalhes de baixo nível. A biblioteca *Runtime* é dividida em componentes que gerenciam operações do *host* (principalmente a comunicação deste com o dispositivo - *Device*), do dispositivo (as funções específicas suportadas pela GPU) e componentes comuns, como tipos adicionais e subconjuntos de funções da biblioteca padrão C que podem ser utilizados no *host* quanto no dispositivo. A API *Runtime* é responsável por realizar implicitamente a inicialização, gerenciamento de contexto e dos módulos [6].

O *CUDA Driver API* corresponde a camada intermediária entre o código compilado e a GPU, sendo implementada na biblioteca dinâmica ***nvcuda***. Por atuar no último nível de abstração, apresenta maior complexidade na sua manipulação direta pela necessidade de maior entendimento das funcionalidades de comunicação com a GPU utilizada pela plataforma CUDA. Devido essa flexibilidade de implementação possibilita um controle maior dos recursos. Com isso o desenvolvedor pode realizar otimizações manuais de algumas funcionalidades presentes nessa camada, viabilizando o desenvolvimento de dispositivos mais complexos e específicos [6].

A partir das características dessas duas camadas pode ser feita a seguinte associação, a API *Runtime* CUDA é vista como uma linguagem de alto nível e o *Driver* API como um intermediário entre o alto e o baixo nível, permitindo uma otimização manual e mais profunda do código. Analisando a relação entre as APIs de *Runtime* e *Driver* com os aplicativos na figura 5 é possível identificar que as duas podem ser utilizadas diretamente pela aplicação. Mas essa utilização não pode ocorrer ao mesmo tempo, isto é, se a aplicação é criada no contexto da API *Runtime* (utilização direta da API) é realizada uma sequência de chamadas a funcionalidades da API *Driver* para a recuperação de informações utilizando o mesmo contexto, sem criar um novo. Esse esquema funciona da mesma forma para contextos criados através do *Driver API*.

Após o entendimento das camadas da API é possível tirar algumas conclusões. A principal é que a NVIDIA utiliza a API para esconder a complexidade de sua GPU. Desta forma os programadores não necessitam escrever diretamente na placa, e sim utilizar funções gráficas pré-definidas na API para operar a GPU. Com isso surgem duas vantagens. A primeira é que os programadores não precisam se preocupar com os detalhes complexos de hardware da GPU, focando apenas no desenvolvimento das aplicações. A outra vantagem, conveniente para a fabricante, é a flexibilidade, que permite a NVIDIA mudar bastante e frequentemente a arquitetura de sua GPU sem tornar a API obsoleta e quebrar softwares já existentes. É importante notar que, apesar da API não se tornar obsoleta, será necessário que programadores de novos modelos utilizem os recursos introduzidos a fim de obter um software otimizado para desempenho [22].

6.2 Modelo de Programação Escalável

A plataforma CUDA na sua essência apresenta abstrações de **memória** (hierarquia de Memória), **grupos de *thread*** (hierarquia de *Thread*) e **sincronização de barreira** (impõe uma barreira de sincronização no código). O programador pode utilizar esses recursos de forma simples, através de um conjunto mínimo de extensões da linguagem. As abstrações proporcionam paralelismo de dados e de *threads* (tarefas), particionando o problema em sub-problemas que podem ser resolvidos de forma independente, em paralelo por blocos de *threads*. Ainda esses sub-problemas podem ser particionados em partes menores sendo resolvidos de forma cooperativa em paralelo por *threads* dentro de cada um dos blocos formados. Esse tipo de decomposição permite que as *threads* cooperem para a resolução de cada sub-problema e, ao mesmo tempo uma escalabilidade automática [6]. A escalabilidade em questão refere-se a locação dos blocos de *threads* nos núcleos disponíveis dos processadores conforme a figura 6.

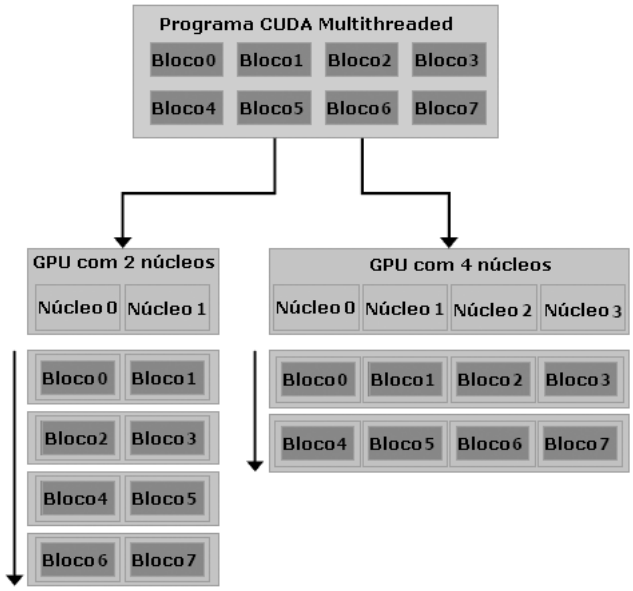


Figura 6: Execução de um Programa *Multithread* CUDA em GPUs com diferentes quantidade de núcleos. [6]

A independência entre os blocos permite a locação destes nos núcleos da GPU em qualquer ordem, simultaneamente ou sequencialmente. Desta forma um programa compilado CUDA pode executar qualquer número de núcleos do processador. Esta configuração tende a maximizar a utilização dos núcleos do processador, evitando que fiquem ociosos.

Na figura 6 é apresentado um programa *multithread* CUDA divido em vários blocos de *threads* independentes uns dos outros que são alocados em GPUs com quantidade de núcleos diferentes. Os blocos são alocados de forma a consumir todo o recurso disponível do processador. Assim a GPU que contém um número maior de núcleos executará em menor tempo o programa do que a GPU com menos núcleos. Visando o fator da divisão dos blocos entre os núcleos, os programadores devem escolher a melhor maneira de dividir os dados em blocos menores.

Em busca de um número ótimo de *threads* e blocos que aproveitem o poder de processamento da GPU por completo, evitando ociosidade dos núcleos é necessário realizar uma analise de diversos fatores. Sendo estes o tamanho do conjunto global de dados, a máxima quantidade de dados locais que um bloco de *threads* pode compartilhar, o número de *stream processors* na GPU e o tamanho das memórias locais.

Essa forma de estruturação da execução altamente paralelizada dos programas utilizando a arquitetura CUDA diminuem o tempo de execução drasticamente. Para tal feito são implementados os *Kernels, Hierarquia de Threads e Hierarquia de Memória* que em conjunto tornam a aplicação extremamente paralelizável influenciando diretamente no desempenho destas.

6.2.1 *Kernel*

Os códigos das aplicações a serem aceleradas utilizando a plataforma CUDA são fortemente modularizados. Isso significa que os programas apresentam suas funcionalidades implementadas em funções. No contexto CUDA a função executada na GPU que é chamada pela CPU (*host*) é denominada de ***Kernels***.

O *Kernel* é mapeado por um conjunto de *Threads*, sendo assim o código presente neste é executado por cada *Thread* do conjunto. Desta forma, em cada chamada ao *Kernel* deve ser especificada sua configuração, com o número de blocos em cada *grid* e o número de *threads* em cada bloco, já a definição da quantidade de memória compartilhada a ser alocada é opcional (vide subseção 7.3.3). Quando é realizada a chamada ao *Kernel* o fluxo de execução sai do CPU (*host*) e passa para a GPU (*device*). Então o código presente no *Kernel* é executado *N* vezes em paralelo por *M* diferentes *threads* de CUDA, sendo a paralelização realizada de forma automática. Quando a execução paralela do *Kernel* termina o fluxo de execução volta para a CPU (*host*) retomando a execução do programa principal [6].

6.2.2 Hierarquia de *Thread*

A plataforma CUDA é altamente paralelizável, devido essa característica a base da execução dos programas são as *threads*. Para organizar essas unidades de execução paralela, o CUDA trabalha com uma Hierarquia de *thread*. Assim são introduzidos dois novos conceitos para o escalonamento das *threads*; ***blocos e grids***. É baseado nestes conceitos que é organizada a repartição dos dados entre as *threads*, bem como sua organização e distribuição ao hardware.

Na figura 7 é representada a distribuição das *threads* de uma parte de um programa. Nos blocos são agrupadas uma quantidade *N* de *threads*, já os blocos são agrupados em *grids*. As coordenadas presentes no *grid*, blocos e *threads* são utilizadas na recuperação e identificação dos mesmos. O *grid* é composto por um conjunto de blocos de *threads*, sendo uma estrutura completa de distribuição das *threads* que executam uma função (*Kernel*). Nele estão definidos o número total de blocos e de *threads* que serão criados e gerenciados pela GPU para uma dada função. O *grid* pode ser *unidimensional* ou *bidimensional*. A figura 7 representa um *grid* de dimensões *2x3*.

A organização das *threads* em blocos é necessária para que estas possam ser executadas de forma independente, garantindo a escalabilidade dos recursos disponíveis na GPU. Assim é possível executá-las em qualquer ordem, em paralelo ou em série. As *threads* dentro de um bloco cooperam com a partilha de dados através da memória compartilhada *(Shared Memory)* e sincroniza sua execução para coordenar os acessos a memória.

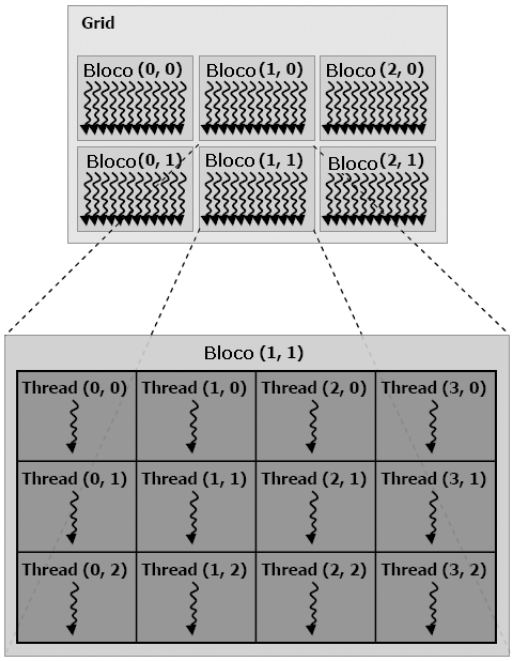


Figura 7: Representação da distribuição das *Threads* nos blocos e estes em *grids*. [6]

Existe um limite de *threads* por blocos, pois o bloco de *threads* deve residir no mesmo núcleo do processador e deve compartilhar os recursos de memória que é limitada nos núcleos. Assim a quantidade total de *threads* designada a um bloco deve ser compatível com os recursos de memória disponíveis no núcleo de processamento da GPU, para que não ocorra a subutilização e nem extrapolação dos mesmos. A plataforma CUDA durante toda sua definição sempre visa a utilização por completo dos recursos disponíveis de hardware para ganho de desempenho. Em GPUs atuais, um bloco de *thread* pode conter até *512 threads*, mas a definição do número total de blocos e *threads* por bloco é geralmente especificada de acordo com o tamanho dos dados que estão sendo processados ou o número de processadores no sistema [6].

Após a definição dos conceitos de *Kernel* e hierarquia de *threads* é possível esquematizar a representação das funções escritas em um código CUDA no dispositivo (GPU). A figura 8 representa de forma simplificada essa estrutura. Um programa é composto por diversas funções, denominadas *Kernels*, que são mapeadas por *threads*. Estas são organizadas em blocos e estes em *grids*. Cada *Kernel* é representado por um *grid* composto por *N* blocos e este por *M threads*. Essas quantidades *N* e *M* são especificadas via código na chamada dos *Kernels*.

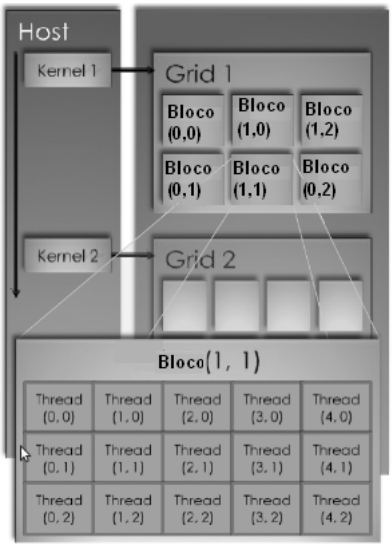


Figura 8: Representação da organização dos dados de um programa CUDA. [35]

Um programa CUDA pode ser composto por centenas de *threads*, contudo nem todas estas *threads* executam simultaneamente com paralelismo real. Para gerenciar a execução destas *threads* existe um escalonador implementado via hardware em cada multiprocessador da GPU, cada um desses multiprocessadores é responsável por cuidar de um agrupamento de 32 *threads* conhecido como ***warp***. O agrupamento e gerenciamento das *threads* é realizado pela nova arquitetura *Single Instruction Multiple Thread* (***SIMT***). Quando um *grid* ou bloco de *thread* for executado por um multiprocessador, são divididos em *warp* que são escalonados pelo SIMT. O escalonamento destas *threads* ocorre, segundo a NVIDIA, com zero *overhead* ([20]). Se durante a execução ocorrer um *branch*, ou seja uma ramificação, em uma *thread* devido a alguma operação de controle de fluxo como um ***"if"***, a *thread* também se divide fazendo que um ramo da execução execute em um ciclo de *clock* e o outro ramo no outro ciclo levando a uma queda de desempenho [6].

6.2.3 Hierarquia de Memória

Durante toda a sua execução as *threads* CUDA podem acessar várias espaços de memória. Na figura 9 são representadas todos os espaços de memória do dispositivo relacionados com uma estrutura do programa.

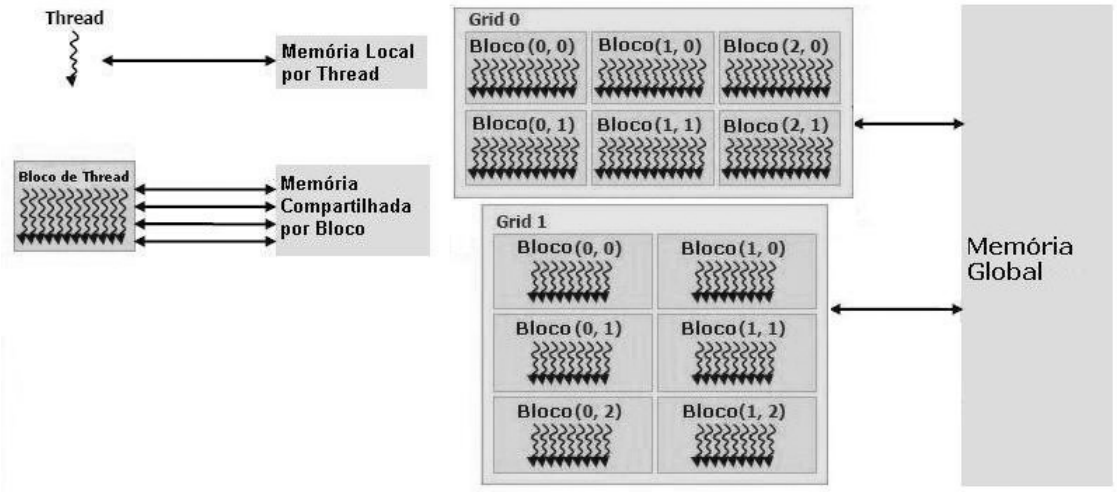


Figura 9: Associação de memória e *Threads*. [6]

Cada *thread* do programa tem uma memória local *(Local Memory)*, acessada única e exclusivamente por ela, para o armazenamento dos dados. Cada bloco acessa uma área específica da memória compartilhada *(Shared Memory)*. Desta forma todas as *threads* pertencentes a um mesmo bloco podem acessar esse espaço de memória e compartilhar dados entre si. Os *grids* tem acesso a memória global *(Global Memory)*, que é a memória principal da GPU, desta forma todas as *threads* que compõem o programa podem compartilhar os dados processados. Assim *threads* de blocos e *grids* diferentes podem compartilhar dados uns com os outros [6]. Na figura 10 é possível visualizar esta interação da Hierarquia de *threads* com as Hierarquias de Memória e o sentido de leitura/escrita dos diferentes tipos de memória.

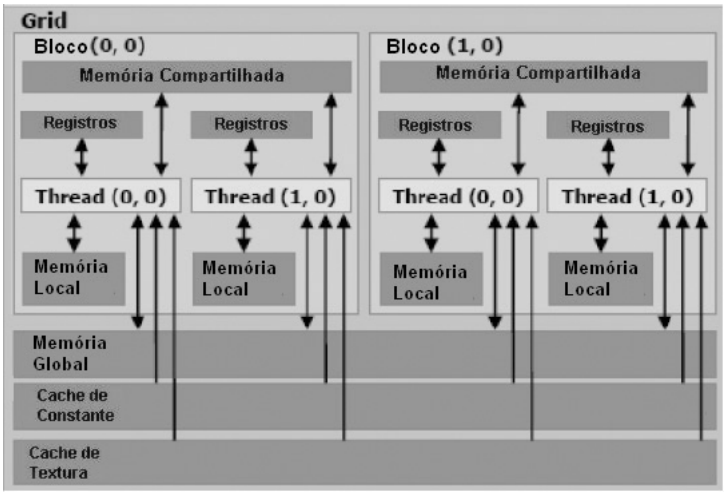


Figura 10: Representação dos acessos dos diferentes tipos de memórias. [32]

Ainda existem outros dois espaços de memória somente de leitura, *textura* e *constante*, que são otimizados para diversos usos da memória. A memória Global, Textura

e Constante são persistentes, não deixando de existir quando a execução do *Kernel* termina. A memória de Textura oferece diferentes modos de endereçamento, bem como dados de filtragem para alguns formatos de dados específicos [6].

A abordagem CUDA , é como a GPU, altamente paralela, porém ele divide o conjunto de dados em pedaços menores na memória ***on-chip***. Esta memória apresenta os tipos: *Cache Constante* para acesso à memória de Constante, *Cache Texture* para acesso à memória de textura, conjunto de registradores de 32 bits e memória compartilhada *(Sharead Memory)*, conforme apresentado na figura 11. Assim as memórias de constante e de textura possuem melhor desempenho do que a local e global devido a utilização de *cache* no acesso aos dados. Já as memórias compartilhadas *(Shared Memory)* e *register* tem o melhor desempenho entre todos os tipos por estarem no mesmo *chip* do microprocessador. Sendo a memória global *(Global Memory)* a mais custosa no acesso, por estar mais distante fisicamente do processador. [6]

Todo o gerenciamento de memória passa pelo *runtime* do CUDA, incluindo a alocação, desalocação e transferência de dados entre as memórias do *host* (CPU) e do *device* (GPU). As diretivas responsáveis por esse gerenciamento estão disponíveis na biblioteca CUDA.

A grande novidade do CUDA juntamente com a arquitetura da GPU é a maneira eficiente que possibilita o desenvolvimento de aplicações que podem explorar ao máximo o paralelismo dos dados. Um mesmo trecho de código é executado em paralelo para pequenos blocos de dados, com a existência de várias pequenas *caches* e níveis de hierarquia de memória ajudam a esconder a *latência* de acesso a estes blocos. A latência refere-se ao tempo de acesso a memória, esse tempo é influenciado diretamente pela distância física que a unidade de memória está da unidade que deseja acessá-la. Assim quanto mais perto do processador a unidade de memória estiver mais rápido será o acesso aos dados (menor latência).

A figura 11 apresenta a revolucionária arquitetura de computação, *GPU Tesla*, baseada no NVIDIA CUDA com até 960 núcleos de processamento paralelo [11]. Nesta figura é representada a posição de cada espaço de memória em relação ao processador. As setas representam o sentido do processo de escrita e leitura nos espaços de memória referenciados.

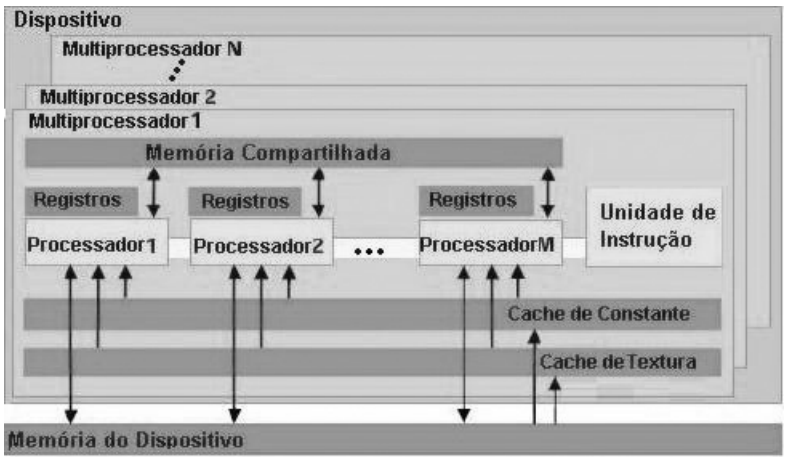


Figura 11: Arquitetura da GPU *Tesla* representando a interação das unidade de memória. [27]

7 CUDA C

A plataforma CUDA apresenta suporte a várias linguagens e interfaces de programação como ***C/C++, Fortran CUDA, OpenCL*** e ***DirectCompute***. Pelo fato do modelo CUDA ser utilizado para solucionar problemas complexos utilizando a GPU, a programação neste modelo não é considerada tão simples quanto a procedural. Assim, a NVIDIA disponibiliza um ambiente para desenvolvimento que possibilita a utilização da linguagem C, por exemplo, como uma linguagem de alto nível. Com o objetivo de tornar a curva de aprendizagem dos programadores de linguagens padrões, como a C, mais amena [6].

Os programas desenvolvidos e analisados neste trabalho (vide capítulo 8, Anexos A e B) utilizam a plataforma CUDA juntamente com a *linguagem C*. O ***CUDA C*** faz uma mescla de um conjunto mínimo de extensões da linguagem C e as diretivas próprias de programação da plataforma, *biblioteca de Runtime*, para o desenvolvimento de aplicações paralelas.

7.1 Compilação com NVCC

A NVIDIA disponibiliza o compilador ***NVCC*** para a compilação dos programas CUDA. O NVCC é usado na compilação dos códigos escritos em *Parallel ThreadXexcution* (*PTX*) e *C*. De modo a facilitar o processo de compilação fornecendo opções de linhas de comando simples e familiares e invoca diversas ferramentas para cada estágio de compilação [6]. A figura 12 representa o fluxo de compilação de um simples programa CUDA.

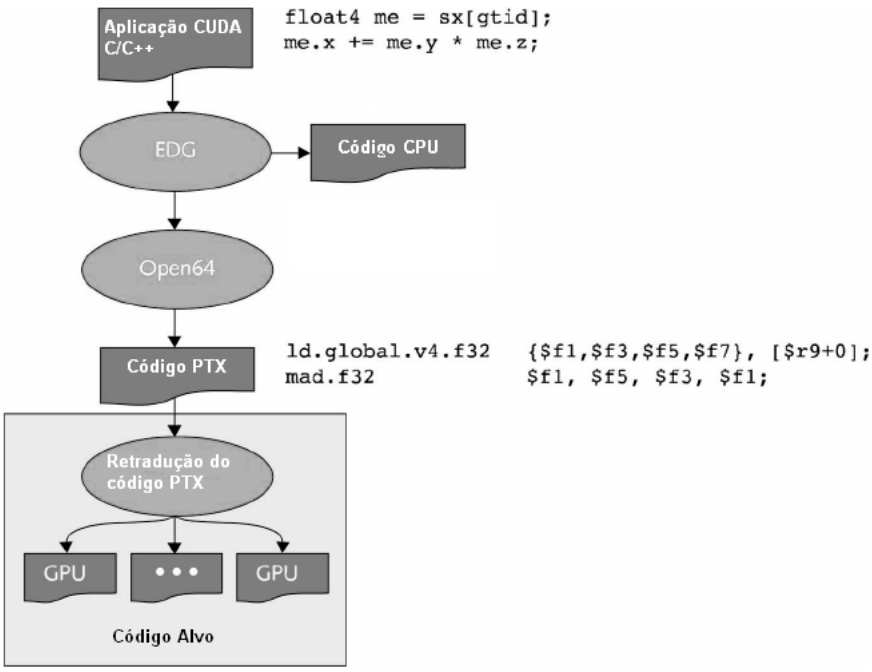


Figura 12: Fluxo de Compilação de um programa CUDA. [22]

Seguindo o fluxo da figura 12, o primeiro estágio de compilação é realizar a separação do código da CPU do código da GPU. Para realizar esta divisão é usado um processador no *front-end*, ***EDG***. Este realiza a análise do código produzindo duas saídas: código C puro para o *host* (CPU) e um objeto *cubin* (formato binário) para o *device* (GPU). O próximo passo é realizar a compilação dos códigos. O código C puro é compilado pelo compilador nativo (GCC, *Microsoft Compiler*, dentre outros) e o código da GPU é compilado com uma versão customizada do ***Open64***. Este é um compilador de código aberto que gera código para execução em paralelo. Após a compilação pelo *Open64* é gerado um código no formato *assembly* chamado ***PTX***. Este código é convertido para código executável em tempo de execução (*Just-in-time*). Assim é utilizado um retradutor de *assembly* que gera um código, específico para determinada modelo de GPU, utilizando o código PTX [22].

A compilação *Just-in-time* do código ocasiona o aumento do tempo de carregamento do aplicativo CUDA, mas permite que os aplicativos possam tirar proveito das melhorias mais recentes do compilador. Além de ser a única forma das aplicações rodarem em dispositivo que não existiam no momento, no qual, a aplicação foi compilada [6].

7.2 Fluxo de Execução

Após a definição do fluxo de compilação de uma aplicação CUDA, é necessário entender o *fluxo de execução* da mesma. Este entendimento é fundamental para a codificação (vide subseção 7.3.4), pois define os passos a serem implementados para que a GPU funcione em conjunto com a CPU.

A figura 13 representa de forma simplificada o fluxo de execução do programa CUDA realizando a seguinte divisão em etapas [6]:

1. Os dados a serem processados na GPU são copiados da memória RAM para a memória principal da GPU;
2. A execução na GPU é disparada na chamada do *Kernel*. Neste momento é realizada a *Configuração de Execução* (vide subseção 7.3.3);
3. Os dados são processados em paralelo em cada núcleo disponível na GPU;
4. Após o processamento dos dados, o resultado deste é copiado da memória principal da GPU para a memória RAM da CPU, para futuros processamentos.

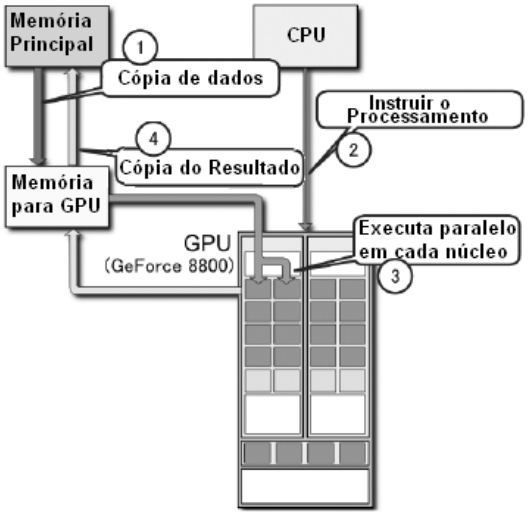


Figura 13: Fluxo de Execução de um programa CUDA. [40]

Desta forma o fluxo de execução de uma aplicação que utiliza a tecnologia CUDA alterna sua execução na CPU e GPU, como na figura 14.

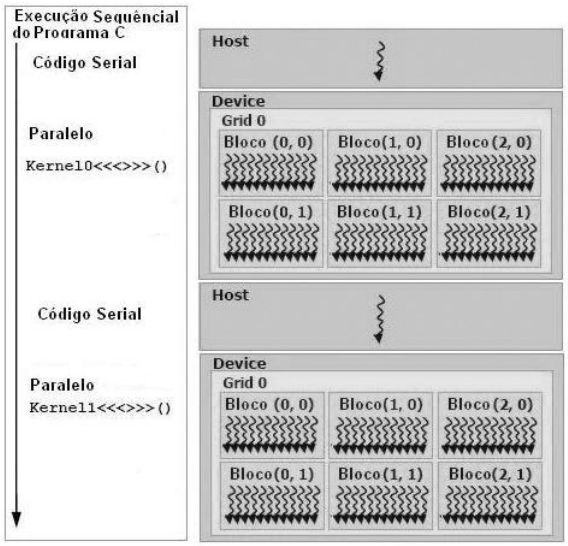


Figura 14: Alternancia da execução de um programa CUDA. [6]

7.3 Paradigma de Programação CUDA

Para iniciar o desenvolvimento de programas CUDA é necessário ter em mente alguns conceitos da linguagem utilizada pela plataforma. Nos próximos tópicos são listados, de forma sucinta, alguns dos pontos fundamentais para o início da aprendizagem dos desenvolvedores. A saber, os grupos de qualificadores, as variáveis *Built-in*, a configuração de execução, o código CUDA na CPU e as funções de sincronização.

7.3.1 Grupos de Qualificadores

Os qualificadores na plataforma CUDA são responsáveis por designar o local de execução das funções, *host* (CPU) e *device* (GPU), e a localização das variáveis nos dispositivos de memória disponíveis. Desta forma é possível identificar dois tipos de qualificadores segundo a NVIDIA [6]:

1. Qualificadores de Funções

   i. __device__ : É usado na declaração de funções que são executadas no *device* (GPU), e só poderão ser chamadas a partir do mesmo. Funções declaradas desta forma apresentam as seguintes restrições:

   - Não suportam recursão;
   - Não é possível a declaração de variáveis estáticas dentro de seu corpo;

- Não podem apresentar número variável de argumentos;
- O endereço da função não pode ser obtido.

ii. **__global__** : É usado na declaração do *Kernel*, funções que são chamadas pelo *host* (CPU) e executadas pelo *device* (GPU). Funções declaradas desta forma apresentam as seguintes restrições:

- Não suportam recursão;
- Não podem declarar variáveis estáticas dentro do seu corpo;
- Não podem apresentar número variável de argumentos;
- Devem apresentar retorno do tipo ***void***;
- Nas chamadas à função deve ser especificada a configuração de execução. (Vide subseção 7.3.3)

iii. **__host__** : É usado na declaração de funções que são executadas no *host* (CPU), e só poderão ser chamadas a partir do mesmo.

Caso a função seja declarada sem nenhum qualificador ela será considerada do tipo **__host__** por padrão. Nos casos, em que a função deve ser compilada para o *device* (GPU) e *host* (CPU) os qualificadores **__host__** e **__device__** são usados combinados. Já os qualificadores **__global__** e **__host__** não podem ser usados juntos.

2. Qualificadores de Variáveis

i. **__shared__** : É usado para a declaração de variáveis no espaço de memória de um bloco de *threads*, sendo somente acessível, durante toda a sua vida, pelas *threads* do mesmo.

ii. **__device__** : É usado na declaração das variáveis que devem residir no *device* (GPU) e na memória global. É acessível, durante toda a sua vida, pelo *host* (CPU) através da biblioteca de *Runtime* e por todas as *threads* em um *grid*.

iii. **__constant__** : É usado na declaração de variáveis que devem residir no espaço de memória de constante, tendo a mesma acessibilidade do **__device__**, durante toda a sua vida.

Os qualificadores listados apresentam as seguintes restrições:

- Estes qualificadores não podem ser utilizados na declaração de *structs*, *union*, parâmetros formais e variáveis locais dentro de uma função que é executada pelo *host* (CPU);
- Os qualificadores **__shared__** e **__constant__** implicam na declaração de variáveis estáticas;
- As variáveis do tipo **__device__**, **__shared__** e **__constant__** não podem ser definidas como externo usando a palavra reservada **extern**. Tendo como exceção as variáveis do tipo **__shared__** alocadas dinamicamente;
- Variáveis do tipo **__constant__** não pode ser atribuída a partir do *device* (GPU), sendo permitida atribuição realizada pelo *host* através da biblioteca de *Runtime*;
- As variáveis do tipo **__shared__** não podem ter uma inicialização como parte da sua declaração.

7.3.2 Variáveis *Built-in*

Variáveis *Built-in* servem para especificar o *grid*, a dimensão dos blocos e os índices das *threads*. Sendo válidas apenas dentro das funções que são executadas pelo *device* (GPU). Essas variáveis são de acordo com a NVIDIA [6]:

- **gridDim**: Variável do tipo **dim3** e corresponde a dimensão do *grid*;
- **blockIdx**: Variável do tipo **uint3** e corresponde ao índice do bloco no *grid*;
- **blockDim**: Variável do tipo **dim3** e corresponde a dimensão do bloco;
- **threadIdx**: Variável do tipo **uint3** e corresponde ao índice da *thread* no bloco;
- **warpSize**: Variável do tipo **int** e contêm o tamanho do *warp* em *threads*.

O tipo **dim3** corresponde a um vetor de inteiros especializado para a definição de dimensões. Já o **uint3** é utilizado para designar vetores, derivando do tipo básico *int*. Ambos tipos estão disponíveis na biblioteca de *Runtime* do CUDA. Para nenhuma das variáveis *Built-in* é possível realizar atribuição de valores.

7.3.3 Configuração de Execução

As funções declaradas com o qualificador **__global__** representam o *Kernel*. Este deve ser chamado a partir do *host* (CPU) para ser executado no *device* (GPU). Nessa chamada devem ser definidos alguns parâmetros para a execução da função, constituindo na *Configuração de execução* [6]. Na API *Runtime* CUDA a configuração é expressa da seguinte forma:

<<< Dg, Db, NS, S >>>

Onde:

Dg: Variável do tipo **dim3** e especifica a dimensão e o tamanho do *grid*;

Db: Variável do tipo **dim3** e especifica a dimensão e o tamanho de cada bloco;

NS: Variável do tipo **size_t** e especifica o tamanho, em bytes, do espaço que será alocado dinamicamente na memória compartilhada por bloco;

S: Variável do tipo **cudaStream_t** e especifica um *stream* adicional.

Nas chamadas ao *Kernel* não é necessário realizar a configuração completa apresenta. Os parâmetros **NS** e **S** são opcionais. Para exemplificar, no trecho de código da figura 15 é representada a assinatura do *Kernel*. E no código da figura 16 é representa a chamada deste *Kernel* no *host* (CPU), utilizando a configuração incompleta. As variáveis dentro dos parênteses correspondem aos parâmetros do *Kernel*.

```
1   __global__ void somaVetor(float *A, float *B, float *C)
```

Figura 15: Assinatura do *Kernel*.

```
1   somaVetor<<<Dg, Db, NS>>>(dA, dB, dC);
```

Figura 16: Chamada do *Kernel* **somaVetor**.

7.3.4 Código CUDA na CPU

Programas CUDA apresentam partes que são executadas pela CPU e outras pela GPU. Para realizar um link entre os dois meios são utilizados alguns comandos CUDA no *host* (CPU). O código da figura 17 ilustra os passos preparatórios para o processamento dos dados na GPU e a utilização desses comandos.

```
1  int main()
2  {
3      //Variáveis do host
4      float *h_a;
5      float *h_b;
6      float *h_c;
7
8      //Inicialização das variáveis do host...
9
10     //Variáveis do device
11     float *d_a;
12     float *d_b;
13     float *d_r;
14
15     //Alocação das variáveis na GPU
16     cudaMalloc((void**)&d_a, size);
17     cudaMalloc((void**)&d_b, size);
18     cudaMalloc((void**)&d_r, size);
19
20     //Cópia dos dados das variáveis do host (CPU) para o device (GPU)
21     cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
22     cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
23
24     //Definição da quantidade de blocos e threads por bloco...
25     //Chamada ao Kernel...
26
27     //Cópia dos dados das variáveis do device (GPU) para o host (CPU)
28     cudaMemcpy(h_c, d_r, size, cudaMemcpyDeviceToHost);
29
30     //Utiliza o resultado processado...
31     //Desaloca as variáveis do host (CPU)...
32
33     //Desaloca as variáveis do devide (GPU)
34     cudaFree(d_a);
35     cudaFree(d_b);
36     cudaFree(d_r);
37
38     return 0;
39 }
```

Figura 17: Exemplos de codificação no *host*.

Nas linhas 11 à 13 são declaradas as variáveis que serão manipuladas no *device* (GPU). Para a utilização das mesmas é necessário alocá-las dinamicamente na GPU, linhas 16 à 18. A alocação de memória na GPU de maneira dinâmica ocorre de duas formas: *Memória Linear* e *CUDA Array*. Nas linhas 16 à 18 é utilizado o comando ***cudaMalloc*** que realiza a alocação dinâmica de memória linear [9]. Para este tipo de alocação também são utilizados os comandos ***cudaMallocPitch()***, ***cudaMalloc3D()***. Estas funções são recomendadas para a alocação de *array* 2D e 3D [6].

Nas linhas 21, 22 e 28 é utilizado o comando ***cudaMemcpy()***, tipicamente usado na transferência dos dados entre os dispositivos CPU e GPU. Os parâmetros correspondem a variável que receberá os dados, variável da qual serão copiados os dados, tamanho total dos dados (variável do tipo ***size_t***) e o ***enum cudaMemcpyKind Kind***

[9]. O terceiro parâmetro define o sentido da transferência dos dados da seguinte forma:

- ***cudaMemcpyHostToDevice***: *host* → *device*;
- ***cudaMemcpyDeviceToHost***: *device* → *host*.

Por fim nas linhas 34 à 36 é realizada a desalocação das variáveis da GPU com o comando ***cudaFree()*** usado para liberar a memória linear alocada dinamicamente.

7.3.5 Funções de Sincronização

Durante a execução do *Kernel* as *threads* de um determinado bloco podem acessar o mesmo endereço de memória compartilhada ou global. As *threads* são executadas em paralelo ocasionando um fluxo de leitura e escrita independentes entre si. Desta forma pode ocorrer uma inconsistência dos dados utilizados no processamento. Para evitar tal risco é utilizada a função de sincronização ***__syncthreads()*** [31].

A função de sincronização realiza a sincronização das *threads* de um bloco, isto é, a sincronização é realizada entre as *threads* de um mesmo bloco e não entre *threads* de blocos diferentes. Desta forma, quando a execução do programa encontra o comando ***__syncthreads()*** é aguardada a chegada de todas as *threads* do bloco que estão em execução neste ponto para poder prosseguir a execução do programa.

8 Aplicações CUDA

A utilização da GPU para propostas gerais abriu um novo leque de possibilidades de paralelização de operações matemáticas complexas. Estudiosos que trabalham com aplicações que tem como base problemas que envolvem equações diferenciais e álgebra linear passaram a explorar a utilização da GPU na paralelização de partes críticas, isto é, nas áreas do programa que consomem mais tempo de processamento. Por exemplo, as equações diferenciais são indispensáveis em simulações físicas para jogos e detecção de características em imagens médicas, estes tipos de aplicações necessitam de um tempo de respostas pequeno. Geralmente problemas que envolvem equações diferenciais apresentam como entrada matrizes e utilização sistemas de equações, como é o caso da simulação de transferência de calor ou fluxo de fluídos. A Álgebra Linear também é muito utilizada em soluções de sistemas de equações, pelo fato desses tipos de sistemas utilizarem como base operações de matrizes e vetores [33].

Com a possibilidade de utilização da GPU para auxiliar na paralelização de problemas que apresentam como base equações diferenciais, sistema de equações e operações de álgebra linear vários trabalhos foram desenvolvidos com o intuito de tornar as aplicações mais rápidas, segue abaixo uma pequena amostra:

- Implementação do Método de Simulação de Sistemas de Partículas na GPU, incluindo inter-partículas de colisões usando a GPU para a classificação rápida dessas partículas para determinar o potencial de pares de colisões [24];
- Implementação da multiplicação de matriz adotando uma técnica de computação paralela que distribui a computação sobre uma estrutura lógica em forma de cubo de processadores. Estes processadores utilizam texturas 2D e operações simples para melhorar o desempenho da execução do produto da matriz [26];
- Implementação de um *framework* de computação de propósito geral executado na GPU *vertex processor*. Como caso de testes do *framework* foram implemen-

tadas algumas operações de álgebra linear e que tiveram sua execução comparada com as mesmas implementações na CPU. Os testes demonstraram que, em especial para matrizes grandes a GPU tem potencial para ultrapassar as soluções otimizadas para a CPU [33].

É possível perceber que várias aplicações baseadas em Equações Diferenciais e Álgebra Linear apresentam como base operações envolvendo Sistemas Lineares e manipulação de Matrizes e Vetores. Com isso, neste trabalho será realizado uma análise das implementações sequencial e paralela das operações básicas da álgebra linear, Produto Matriz Vetor e Produto Interno. Ambas operações são muito utilizadas na resolução de Sistemas Lineares.

Nas próximas seções será realizado um breve estudo para contextualização do conceito e tipos de Sistemas Lineares. Logo após será apresentada uma comparação da implementação das operações Produto Interno e Produto Matriz Vetor utilizando algoritmos sequenciais (linguagem C) e paralelos (C+CUDA). O foco da comparação será a mudança nos passos de execução do algoritmo para a sua paralelização, além de destacar características da implementação utilizando a arquitetura CUDA. Após todo este estudo será apresentado no capítulo 9 uma análise de desempenho entre os algoritmos sequenciais e paralelos apresentados.

8.1 Sistemas Lineares

Sistemas Lineares podem ser vistos como um conjunto de m equações e n incógnitas. Para representar a posição da linha e coluna aos quais um elemento pertence são usados os índices i e j respectivamente.

$$\begin{cases} a_{11} * x_1 & + & a_{12} * x_2 & + & \cdots & + & a_{1n} * x_n = & b_1 \\ a_{21} * x_1 & + & a_{22} * x_2 & + & \cdots & + & a_{2n} * x_n = & b_2 \\ \vdots & & \vdots & & \cdots & & \vdots & = \ \vdots \\ a_{m1} * x_1 & + & a_{m2} * x_2 & + & \cdots & + & a_{mn} * x_n = & b_m \end{cases} \tag{8.1}$$

Onde:

- a_{ij} representa os coeficientes das equações
- b_i representa os termos independentes

- x_i representa as incógnitas

Outra forma muito frequente e vantajosa de representação de sistemas lineares é a notação de forma matricial:

$$\boldsymbol{Ax = b} \tag{8.2}$$

Onde:

- $\boldsymbol{A}$ representa a matriz de coeficientes
- $\boldsymbol{x}$ representa o vetor de incógnitas
- $\boldsymbol{b}$ representa o vetor de termos independentes

Assim o sistema linear (8.1) pode ser representado da seguinte forma:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{8.3}$$

A obtenção do vetor $\boldsymbol{b}$ da expressão 8.2 é feito através do cálculo do Produto Matriz Vetor entre a matriz $\boldsymbol{A}$ e o vetor $\boldsymbol{x}$.

Existem diversos métodos de resolução de sistemas lineares, estes são divididos em dois grupos:

1. **Métodos Exatos:** São aqueles que geram a solução exata do sistema com um número finito de operações. A solução apresenta erros de arredondamento, que são acumulativos ao longo da resolução. Para sistemas lineares de grande porte o erro de arredondamento podem tirar o significado do resultado, isto é, a solução do sistema pode se afastar drasticamente do resultado real. Os Métodos exatos são muito utilizados em problemas que contêm matrizes densas com dimensões pequenas. Exemplos: Método de Eliminação de Gauss, Método de Gauss-Compacto, Método de Eliminação de Gauss com Pivoteamento Parcial, dentre outros.

2. **Métodos Iterativos:** São aqueles que geram a solução do sistema com uma precisão previamente estabelecida através de um processo infinito convergente. Este tipo de método é eficaz em problemas que apresentam matrizes esparsas e de grandes dimensões. Os algoritmos desses métodos apresentam uma economia na utilização de memória do computador. Uma grande vantagem é a auto correção dos erros cometidos. Em algumas condições podem ser utilizados na resolução de sistemas não lineares. Os métodos iterativos podem ser divididos em:

   (a) **Processos Estacionários:** Quando cada aproximante é obtido do anterior sempre pelo mesmo processo, isto é, a cada nova iteração a matriz de iteração não é alterada. Exemplos: Método de Jacobi-Richardson e Método de Gauss-Seidel;

   (b) **Processos Não-Estacionários:** A matriz de iteração não é constante. A cada iteração é obtida uma aproximação seguindo as restrições e utilizando as informações das iterações anteriores. O objetivo é buscar a melhor aproximação a cada iteração. São muito eficientes, tanto numericamente quanto computacionalmente, na solução de sistemas esparsos de grandes dimensões. Exemplos: Método do Gradiente, Método dos Gradientes Conjugados [23], GMRES [37] e LCD [25].

Devido as características computacionais dos métodos iterativos citados, estes são muito utilizados na resolução de sistemas de grande porte na computação. Dentre os seus tipos os não-estacionários apresentam desenvolvimento relativamente mais recente que os estacionários, por isso a compreensão é mais difícil, mas os ganhos de desempenho são altos em relação aos estacionários [15].

Nos métodos iterativos não-estacionários as constantes são tipicamente calculadas utilizando produtos internos de resíduos ou outros vetores que surgem através do método iterativo. Usualmente esses tipos de métodos são baseados na ideia de sequência de vetores ortogonais [15]. Além do produto interno ocorre o produto matriz vetor através da multiplicação da matriz A pelos vetores de aproximações x de acordo com a equação 8.4:

$$r = b - A * x \tag{8.4}$$

Assim as aplicações dos métodos de resolução de sistemas iterativos não-estacionários são baseadas nas duas operações básicas da álgebra linear, produto matriz vetor e o produto interno. Desta forma, a execução dos métodos é concentrada nessas operações. Com isso, a otimização no desempenho computacional deve focar nessas duas operações bases.

Identificando a importância e a grande utilização das operações de produto matriz vetor e produto interno nos métodos iterativos não-estacionários, na seção 9.3.1 será realizada uma projeção da paralelização dessas operações básicas do Método dos Gradientes Conjugados tomando como base os algoritmos das seções 8.2 (Produto Interno) e 8.3 (Produto Matriz Vetor), e a comparação do desempenhos dessas duas operações nas subseções 9.2.1 e 9.2.2 respectivamente.

8.2 Produto Interno

O Produto Interno é a multiplicação entre vetores com dimensões iguais resultando em um número escalar. A representação dos vetores do Produto Interno pode ser feita através de matrizes, uma matriz A com dimensões n x 1 e uma matriz B com dimensões 1 x n, da seguinte forma:

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} * \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix} = R \tag{8.5}$$

Seguindo o conceito da multiplicação entre matrizes, o cálculo realizado no Produto Interno ocorre de acordo com a expressão 8.6.

$$\begin{aligned} R = & \quad (a_1 * b_1) + (a_2 * b_2) + \cdots + (a_n * b_n) \\ R = & \quad \Sigma_{i=1}^{n} \ (a_i * b_i) \end{aligned} \tag{8.6}$$

8.2.1 Algoritmo Sequencial

O Produto Interno é basicamente a soma dos produtos das posições correspondentes dos dois vetores envolvidos. A figura 18 mostra a função do Produto Interno implementada na linguagem C.

```
1  int ProdutoInterno(float *a, float *b, int N)
2  {
3      int i;
4      int R = 0;
5
6      for(i = 0 ; i < N; i++)
7      {
8          R += a[i] * b[i];
9      }
10
11     return R;
12 }
```

Figura 18: Produto Interno na Linguagem C

A linha 8 da figura 18 corresponde à equação 8.6, essa operação será realizada **N** vezes, de acordo com o tamanho dos vetores envolvidos no produto. Desta forma a complexidade do Produto Interno é $\Theta(n)$. Assim, a quantidade de operações realizadas crescerá linearmente de acordo com a quantidade de elementos dos vetores.

8.2.2 Algoritmo Paralelo C+CUDA

O algoritmo paralelo do Produto Interno apresenta uma abordagem diferente do sequencial. Para a implementação da versão paralela C+CUDA foram realizados os seguintes passos:

1. É realizada a multiplicação de cada elemento dos vetores de mesma posição e o produto calculado é armazenado em um vetor de resultados na posição correspondente;

2. Para o cálculo do resultado final será realizada a soma em árvore (*Sum Tree Like Reduction*) do vetor resultado.

Desta forma, o que era feito em uma linha de código no algoritmo sequencial (linha 8 da figura 18) será realizado em 2 passos no algoritmo paralelo C+CUDA.

Cada passo é representado por um *Kernel*. Para execução destes é necessária a definição dos parâmetros de configuração de execução. De acordo com a subseção 7.3.3, o *Kernel* necessita de uma configuração inicial para a sua execução. Sendo obrigatório a passagem do tamanho do *grid* e do bloco. No algoritmo implementado o *grid* e os blocos são unidimensionais, desta forma são passados como parâmetros

de configuração para o *Kernel* variáveis do tipo **int** com a quantidade de blocos e de *threads* por bloco, respectivamente, que são calculados de acordo com o código da figura 19.

```
1    if(N < THREADPERBLOCK )
2     {
3         threadsPerBlock = N;
4         numBlocks       = (N + N - 1)/N;
5     }
6     else
7     {
8     threadsPerBlock = THREADPERBLOCK;
9         numBlocks       = (N + THREADPERBLOCK - 1)/THREADPERBLOCK ;
10    }
```

Figura 19: Cálculo da quantidade de blocos e *threads* por bloco.

A constante ***THREADPERBLOCK*** representa a quantidade máxima de *threads* que o bloco pode conter. O valor dessa constante, durante os testes realizados neste trabalho, varia entre 128, 256 e 512, sendo este último o máximo de *threads* que a arquitetura CUDA, na versão utilizada (vide 9.1), suporta em um bloco. De acordo com a linha 3 e 8, a quantidade de *threads* em um bloco varia de acordo com o tamanho do vetor. Assim se o tamanho do vetor for menor que a quantidade máxima de *threads* definida por bloco, a quantidade de *threads* por bloco será igual ao tamanho do vetor, caso contrário será a quantidade definida pela constante. A quantidade de blocos será calculada de acordo com o tamanho do vetor e a quantidade de *threads* por bloco de forma a gerar um número inteiro e exato para comportar todas as *threads* (linhas 4 e 9).

Após a definição da configuração do *Kernel* é realizado o passo 1 do algoritmo. Na chamada ao *Kernel* para a multiplicação dos elementos dos vetores, figura 20, são passados os parâmetros de configuração entre <<< >>> e os parâmetros de entrada do *Kernel* entre ( ).

```
1    MultiplicaVetor<<<numBlocks, threadsPerBlock >>>(vetorGpuA, vetorGpuB, vetorGpuC, N);
```

Figura 20: Chamada do *Kernel* **ProdutoInterno**.

As variáveis **vetorGpuA** e **vetorGpuB** correspondem aos vetores utilizados nos cálculos que foram previamente alocados e inicializados com os valores do problema. Já a variável **vetorGpuC** representa o vetor de resultados. O parâmetro **N** representa o tamanho dos vetores do problema. Lembrando que os 3 vetores devem apresentar dimensões iguais.

A figura 21 representa a implementação do passo 1 do algoritmo C+CUDA, a multiplicação das posições dos vetores e o armazenamento dos resultados na posição correspondente do vetor de resultados.

```
1  __global__ void MultiplicaVetor(float* A, float* B, float* C, int N)
2  {
3      int i = blockDim.x * blockIdx.x + threadIdx.x;
4
5      if(i<N)
6      {
7          C[i] = A[i] * B[i];
8      }
9  }
```

Figura 21: *Kernel* de multiplicação das posições dos vetores, corresponde ao passo 1 do algoritmo.

Quando o *Kernel* é executado, cada posição do vetor é representada por uma *thread*, que durante a execução irá ajudar na indexação do valor de acordo com a linha 3 da figura 21. O valor de **i** será a posição do dado do vetor para a operação. Essa posição é dada através dos valores da coordenada **x** da dimensão do bloco, **blockDim.x**, do índice do bloco no *grid*, **blockIdx.x**, e do índice da *thread* no bloco, **threadIdx.x**.

Não é necessário nenhum tipo de *loop* para os cálculos, quando o *Kernel* é chamado todas as *threads* dos blocos são executas paralelamente e independentemente nos processadores disponíveis na GPU. O único tratamento realizado é o **if** da linha 5. Essa linha garante que os cálculos serão feitos em posições válidas de acordo com o tamanho dos vetores envolvidos na multiplicação. Desta forma, evita acesso inválido de espaço de memória e a utilização de dados inconsistentes nos cálculos.

Após a execução do *Kernel* **MultiplicaVetor** é executado o passo 2 do algoritmo, a soma do vetor resultado em árvore. Antes de analisar a implementação do *Kernel* de Redução é necessário entender como funciona a soma em árvore (*Sum Tree Like Reduction*).

A ideia principal da soma em árvore é dividir o vetor em duas partes iguais. A segunda metade é somada às posições correspondentes da primeira metade do vetor. Desta forma, na próxima iteração é considerado como vetor a primeira metade da divisão anterior e assim sucessivamente. A figura 22 representa o passo a passo do algoritmo de Soma em Árvore.

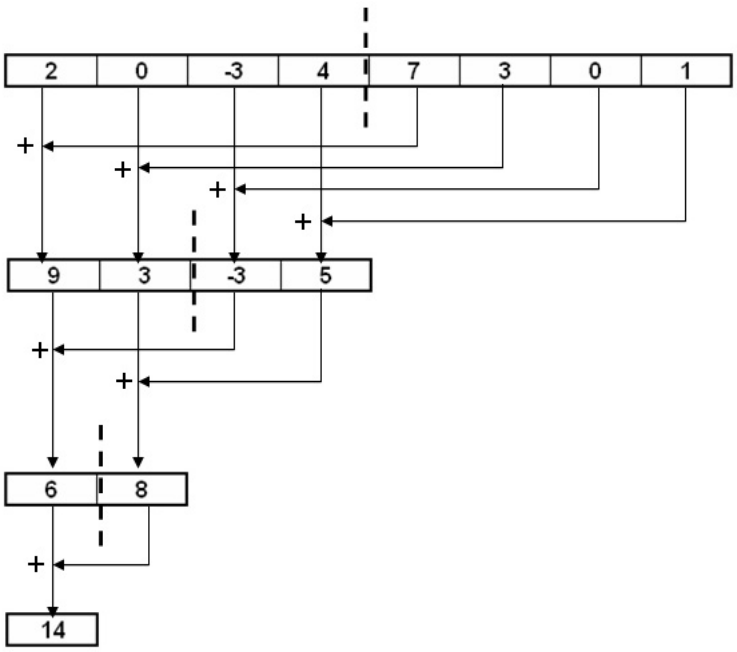


Figura 22: Demonstração do funcionamento do algoritmo de Soma em Árvore.

Na figura 22 é utilizado um vetor de 8 posições. Para realizar o 1º passo da redução, o vetor é dividido em duas partes iguais, ambas com 4 posições. Os elementos 2 e 7 são somados e o resultado é colocado na posição 0; os elementos 0 e 3 são somados e o resultado é colocado na posição 1 e assim sucessivamente para os demais elementos. Na segunda iteração o vetor terá tamanho igual a 4 (considerando apenas a 1º metade do vetor da divisão anterior), na terceira o tamanho será reduzido para 2 e por fim, na última iteração o tamanho será igual a 1. Sendo assim, no final do algoritmo a posição 0 terá a soma de todos os elementos do vetor. Para que o processo de redução ocorra de maneira correta o tamanho do vetor deve corresponder a uma potência de 2, pois a cada iteração é necessário realizar a divisão do vetor em duas partes iguais.

Após o entendimento do algoritmo de soma em árvore é possível analisar a implementação do mesmo em C+CUDA. A figura 23 representa a chamada do *Kernel* que irá realizar a soma em árvore do vetor de resultados.

```
1    for(j = 0, p = 0; j < numBlocks; j++, p += threadsPerBlock)
2     {
3         ProdutoInterno<<<BLOCKREDUCTION, threadsPerBlock >>>(vetorGpuC, threadsPerBlock, p, soma);
4     }
```

Figura 23: Chamada do *Kernel* de soma em árvore.

É necessário realizar um *loop* pela quantidade de blocos para garantir a sincronização da execução entre os blocos. A arquitetura CUDA possibilita a sincroniza-

ção da execução das threads de um bloco, mas não entre blocos diferentes. Assim, se não houver essa sincronização entre os blocos, durante a execução do *Kernel* poderá ocorrer utilização de valores em posições incorretas. Então para garantir o acesso nas posições consistentes e corretas é realizada uma chamada ao *Kernel* do **ProdutoInterno** passando como parâmetros de configuração **BLOCKREDUCTION**, uma constante de valor igual a 1 e a quantidade de *threads* por blocos, representada pela variável **threadsPerBlock**. Assim a cada chamada o *Kernel* executará apenas um bloco.

Como parâmetros de entrada do *Kernel* são passados o vetor de resultados, **vetorGpuC**; a quantidade de *threads* por bloco, **threadsPerBlock**; a variável **p** que irá auxiliar na indexação dos elementos do vetor, indicando em qual bloco os dados estão; e a variável **soma** na qual será atribuído o valor final da soma em árvore. A variável **p** será incrementada a cada iteração do **for**, linha 1 da figura 23, com a quantidade de *threads* por bloco, deslocando assim a indexação dos valores para o bloco seguinte. A figura 24 representa a implementação do *kernel* da soma em árvore em C+CUDA.

```
1  __global__ void ProdutoInterno(float *C, int tamVetor, int p, float *soma)
2  {
3      int j,k;
4
5      for(k = tamVetor/2; k > 0; k >>= 1)
6      {
7          __syncthreads();
8          for(j = threadIdx.x; j < k; j += blockDim.x)
9          {
10             C[j+p] += C[k+j+p];
11         }
12     }
13     __syncthreads();
14
15     if(threadIdx.x==0)
16         *soma += C[p];
17 }
```

Figura 24: *Kernel* da soma em árvore.

Na linha 5 contém o *loop* responsável pela divisão do vetor em duas partes iguais, já o *loop* da linha 8 é responsável pela soma paralela. O **for** da linha 8 apresenta o índice **i** variando do índice **x** da *thread* até o **k** que representa o tamanho do vetor após a divisão, o incremento é o **blockDim.x** que representa a quantidade de *threads* que o bloco contêm. O comando **__syncthreads()** utilizado nas linhas 7 e 13, serve para sincronizar as *threads*, assim as demais linhas do código só serão executadas

quando todas as *threads* em execução chegarem naquele ponto do programa. Desta forma, garante a consistência dos dados. No final do algoritmo apenas a *thread* de índice igual a 0 terá o valor total da soma do vetor de resultados. O comportamento do algoritmo na divisão do vetor é representado pela figura 25.

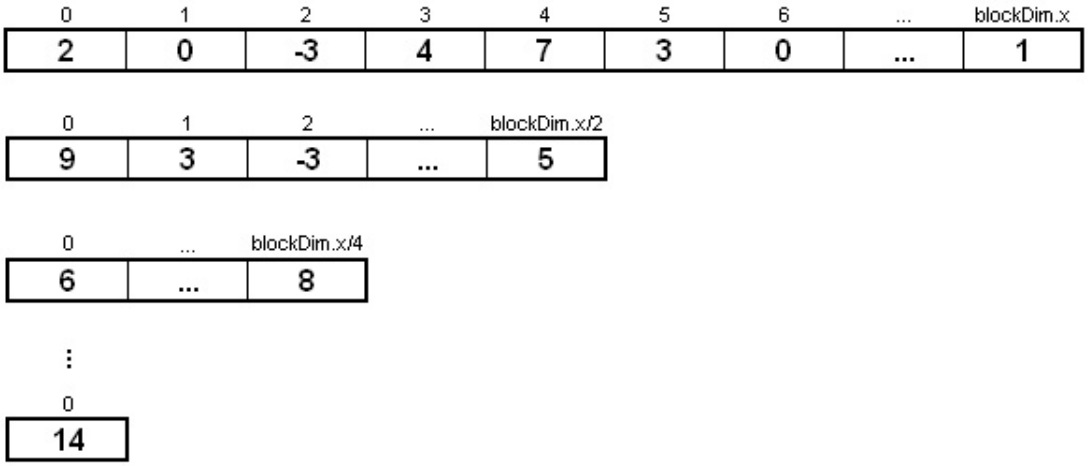


Figura 25: Comportamento do algoritmo C+CUDA na divisão do vetor.

A cada chamada ao *Kernel* será realizada a operação de soma em uma parte do vetor de acordo com a quantidade de *threads* no bloco. Assim, o tamanho do vetor a ser manipulado a cada chamada será igual a quantidade de threads do bloco, **blockDim.x**.

O código completo do programa do Produto Interno implementado em C+CUDA pode ser analisado no Anexo A e a simulação da execução dos *Kernels* no anexo D.

8.3 Produto Matriz Vetor

O Produto Matriz Vetor, como próprio nome diz é a multiplicação entre uma matriz A com dimensões m x n e um vetor b com dimensão n (ou n x 1) que resulta em um vetor r de dimensão m (ou m x 1) de acordo com a equação 8.7.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{bmatrix} \tag{8.7}$$

Analisando essa operação em partes é possível afirmar que cada r_k é calculado

pelo produto interno da linha k da matriz A, que pode ser vista como um vetor, e o vetor b. A equação 8.8 pode ser considerada equivalente a expressão 8.6 do Produto Interno.

$$\begin{aligned} r_k = \quad & (a_{k1} * b_1) + (a_{k2} * b_2) + \cdots + (a_{kn} * b_n) \\ r_k = \quad & \Sigma_{i=1}^{n} \ (a_{ki} * b_i) \end{aligned} \tag{8.8}$$

8.3.1 Algoritmo Sequencial

O algoritmo do Produto Matriz Vetor, implementado na linguagem C, é representado pela figura 26.

```
1  void ProdutoMatrizVetor(float **a, float *b, float *r,int N)
2  {
3      int i, j;
4
5      for(i = 0; i<N; i++)
6      {
7          for(j = 0; j<N; j++)
8          {
9              r[i] += a[i][j]*b[j];
10         }
11     }
12 }
```

Figura 26: Produto Matriz Vetor na Linguagem C

A quantidade de operações realizadas na linha 9 da figura 26 depende das dimensões da matriz a. Desta forma, sendo a matriz a quadrada (NxN), a complexidade da função **ProdutoMatrizVetor** é de $\Theta(n^2)$.

8.3.2 Algoritmo Paralelo C+CUDA

A implementação do algoritmo paralelo C+CUDA do Produto Matriz Vetor apresenta algumas diferenças em relação a estrutura dos dados do algoritmo sequencial. No algoritmo na linguagem C, foram alocados uma matriz de M linhas com N colunas e um vetor com N posições. Já no paralelo a matriz foi alocada como um vetor, desta forma para representar uma matriz de dimensões de 7x8 é alocado um vetor com 56 posições. O motivo da escolha desse tipo de representação para a estrutura da matriz no programa paralelo foi a simplicidade de alocação, cópia e manipulação dos dados.

Diferentemente da implementação do Produto Interno da seção 8.2.2 o Produto Matriz Vetor é composto por uma *Kernel* chamado no programa principal apenas uma vez, sem realização de *loops*, figura 27.

```
1   MultiplicaMatriz<<< numBlocks, threadsPerBlock  >>>(matrizGpu, L, C, vetorGpu, vetorResulGpu);
```

Figura 27: Chamada ao *Kernel* **MultiplicaMatriz**

Como parâmetros de entrada são passadas **matrizGpu** vetor com os dados da matriz do problema, **L** a quantidade de linhas da matriz, **C** a quantidade de colunas da matriz, **vetorGpu** vetor com os dados do problema e **vetorResulGpu** vetor no qual serão armazenados os valores resultantes da multiplicação entre a matriz e o vetor. Os parâmetros de configuração, **numBlocks** e **threadsPerBlock** seguem o mesmo cálculo da figura 19.

A implementação do *Kernel* do Produto Matriz Vetor é representado pela figura 28.

```
1   __global__ void MultiplicaMatriz(float *A, int numLinhas, int numColunas, float *R, float *X)
2   {
3       __shared__ float T[THREADPERBLOCK];
4       int ini, fim, i, j, k;
5
6       for(i = blockIdx.x; i < numLinhas; i += gridDim.x)
7       {
8           ini = i * numColunas;
9           fim = ini + numColunas;
10
11          T[threadIdx.x] = 0.0;
12
13          for(j = threadIdx.x + ini, k = threadIdx.x; j < fim; j += blockDim.x, k += blockDim.x)
14          {
15              T[threadIdx.x] += R[k] * A[j];
16          }
17
18          __syncthreads();
19
20          ProdutoInterno(T, THREADPERBLOCK, &X[i]);
21      }
22  }
```

Figura 28: *Kernel* **MultiplicaMatriz**

Para compreender a paralelização do Produto Matriz Vetor implementado em C+CUDA é necessário entender o deslocamento que o algoritmo realiza pelas *threads* dos blocos. Suponha uma matriz **A** com 7 linhas e 8 colunas, que podem ser representadas na configuração do *Kernel* por 4 blocos contendo cada um 16 *threads*. Mas na prática esses blocos serão trabalhados como uma matriz de 7x8, desta forma são utilizados 2 blocos de 16 *threads* e 2 blocos com 12 *threads*, representada graficamente pela figura 29. As demais *threads* não serão utilizadas por não apresentarem associação com os

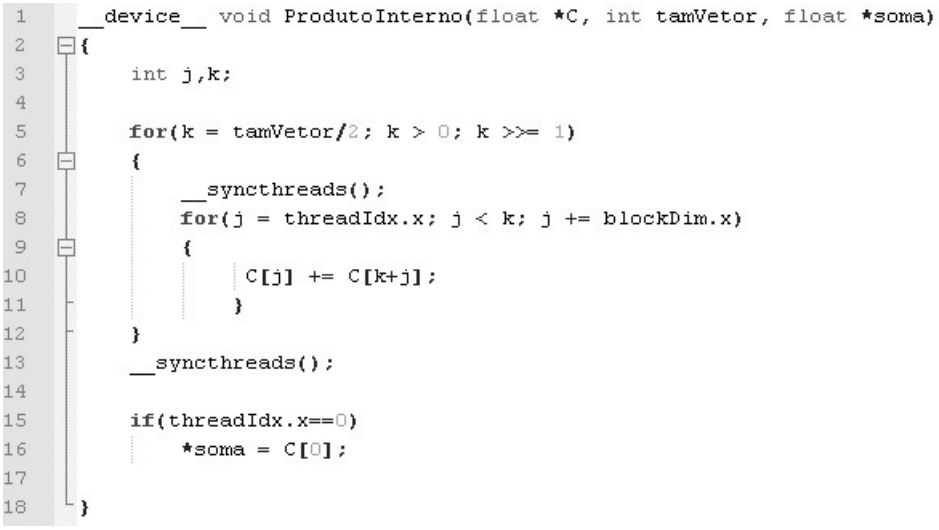
dados da matriz representada. O algoritmo irá começar pelo primeiro bloco, área cinza da figura 29. A cada passo do algoritmo da figura 28 essa área cinza será deslocada para a direita. Este deslocamento será realizado pelo **for** da linha 13 do algoritmo. Quando todos os elementos equivalentes a uma linha da matriz forem multiplicados pelas posições correspondentes do vetor **R** e acumulados na variável local **T**, linha 15, será realizada a soma em árvore implementada pela função **ProdutoInterno**.

Após o cálculo da soma em árvore do vetor **T** é realizado o deslocamento da área cinza para esquerda e para baixo, operação realizada pelo **for** da linha 6. As variáveis **ini** e **fim** são responsáveis, respectivamente, pelo armazenamento dos índices de início e final das linhas da matriz **A**. Desta forma, é possível ter uma quantidade total de *threads* não necessariamente múltiplas da quantidade total de elementos contidos na matriz.

(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)
(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)

Figura 29: Exemplo do deslocamento do algoritmo Produto Matriz Vetor pelas *threads* dos blocos.

A função **ProdutoInterno** chamada dentro do *Kernel* **MultiplicaMatriz** na linha 20 do código da figura 28 segue a mesma lógica do algoritmo da figura 24. A função da figura 30 utilizada no *Kernel* do produto matriz vetor é do tipo ***__device__***, isto é, uma função que só poderá ser chamada dentro de um *Kernel* e executada dentro da GPU.



```
1   __device__ void ProdutoInterno(float *C, int tamVetor, float *soma)
2   {
3       int j,k;
4
5       for(k = tamVetor/2; k > 0; k >>= 1)
6       {
7           __syncthreads();
8           for(j = threadIdx.x; j < k; j += blockDim.x)
9           {
10              C[j] += C[k+j];
11          }
12      }
13      __syncthreads();
14
15      if(threadIdx.x==0)
16          *soma = C[0];
17
18  }
```

Figura 30: Código da função ***__device__*** **ProdutoInterno**.

Os parâmetros de entrada da função **ProdutoInterno** são o vetor **C**, o tamanho do vetor a ser somado **tamVetor** e a variável **soma** que armazenará o valor final da soma de todas as posições do vetor. Não é necessário nenhuma variável auxiliar para controlar a indexação, pois o vetor passado como parâmetro será manipulado por completo de acordo com o valor da variável **tamVetor**.

O código completo do programa do Produto Matriz Vetor implementado em C+CUDA pode ser analisado no Anexo B.

9 Teste de Desempenho

Neste capítulo serão apresentadas as análises de desempenho das duas operações da Álgebra Linear que são básicas na resolução de Sistemas Lineares, Produto Interno e Produto Matriz Vetor apresentadas no capitulo 8. Inicialmente será apresentada a configuração técnica do ambiente, no qual foram realizados todos os testes. Posteriormente será apresenta toda a análise de desempenho obtido pelos algoritmos implementados em C+CUDA e na Linguagem C. Por fim será apresentado um estudo e projeção da paralelização destas operações em Métodos iterativos para soluções de Sistemas Lineares.

9.1 Ambiente de Testes

Todos os testes foram realizados no LCAD23 do Laboratório de Computação de Alto Desempenho do Centro Tecnológico da Universidade Federal do Espírito Santo (LCAD - CT - UFES). Os experimentos C+CUDA e os sequenciais, na linguagem C, correspondentes ao paralelo em LCAD's BOXX *Personal Supercomputer* que é um *quad-core* AMD Phenon X4 9950 de 2,6 GB máquina com 2MB de L2 e 8GB de DRAM e 4 GPU's NVIDIA Tesla C1060 placas PCI *Express* 1.3GHz com 240 *stream processors* e 4 GB de DRAM cada. A versão do compilador CUDA foi o nvcc 2.1.

Os códigos sequenciais relativos ao Produto Interno e Produto Matriz Vetor foram compilados utilizando *"gcc file.c -o file"*. O código do Método dos Gradientes Conjugados foi compilado utilizando *"gcc -gp file.c -o file"* para gerar o arquivo **gmon.out** para permitir a análise das funções do programa. Já os códigos paralelos C+CUDA foram compilados em duas etapas, primeiramente o código **.cu** deve ser transformado em **.cpp** pelo comando *"nvcc -cuda file.cu"*, em seguida para gerar um programa executável é usado o comando *"nvcc file.cu.cpp -o file"*.

O acesso aos recursos computacionais do LCAD pode ser feito de duas formas

principais: acesso direto às máquinas no Laboratório ou através de acesso remoto. O acesso remoto é obtido através da máquina lcad1.lcad.inf.ufes.br. Após a conexão com a máquina lcad1 o usuário poderá acessar as demais máquinas através de **SSH** (*Secure Shell*). Os testes foram realizados na máquina lcad23, na qual o kit de desenvolvimento CUDA está instalado. Os arquivos podem ser transferidos para o LCAD através de SPC (*Secure Copy*) para as máquinas.

9.2 Análise de Desempenho

Nesta seção serão realizadas as análises do desempenho das implementações sequencial e paralela dos algoritmos das operações de Produto Interno e Produto Matriz Vetor apresentadas no capítulo 8.

9.2.1 Produto Interno

Os testes de desempenho dos algoritmos sequencial na linguagem C e o paralelo em C+CUDA do Produto Interno foram realizados no ambiente descrito na seção 9.1. Os tempos de execução obtidos são referentes apenas a execução da função do cálculo do Produto Interno, sendo desprezados os tempos de leitura dos dados e alocação das variáveis do problema.

Os valores dos vetores de entrada são do tipo *float* variando entre 0 e 1, utilizando 4 casas decimais de precisão. A escolha do tipo e a precisão dos dados foram escolhidos de acordo com o suporte da arquitetura CUDA, desta forma é possível obter maior precisão dos dados calculados. Os dados foram salvos em arquivos binários, para otimização do espaço físico, gravação e leitura dos dados pelo programa. Os mesmos arquivos de entrada foram utilizados em ambos os programas, para que sejam analisados nas mesmas condições.

A obtenção dos tempos, em milissegundos, dos algoritmos sequencial em C e paralelo em C+CUDA apresentados nas tabelas 2, 3 e 4 foi feita através da média aritmética dos tempos de execução dos mesmos. Ambos os programas foram executados 5 vezes com a mesma entrada e configuração. Foi definida uma quantidade relativamente pequena de testes, pois o comportamento dos tempos de execução não apresentam grande variação. Destas 5 execuções foram retirados o melhor e pior tempo. Com os demais tempos foi realizado o cálculo da média aritmética simples.

Como os dados no algoritmo paralelo são representados por *threads* e estas estão divididas em blocos, foram realizadas diferentes divisões destas nos blocos.

Os dados no algoritmo paralelo CUDA+C são representados por *threads* que são divididas igualmente entre os blocos na configuração do *kernel*. Com isso a forma de organização dessas *threads* pode afetar o desempenho do algoritmo. Para identificar essa influência durante os testes de desempenho a quantidade de *threads* por bloco foi variada assumindo os valores 128, 256 e 512. Estes valores são potências de 2 devido as características do algoritmo implementado.

As tabelas 2, 3 e 4 mostram a relação da quantidade de dados com os tempos obtidos em cada algoritmo e o *speed-up* calculado através da divisão do tempo de execução do programa sequencial pelo tempo de execução do programa paralelo.

Tabela 2: Média e *Speed-up* dos tempos de execução do algoritmo do Produto Interno sequencial em C e paralelo C+CUDA com blocos de 128 *threads*

Qtd. de Dados	C(ms)	C+CUDA(ms)	*Speed-up*
1024	18,6667	54,6667	0,3415
2048	36,6667	74,3333	0,4933
4096	72,0000	111,0000	0,6486
8192	145,0000	184,6667	0,7852
16384	289,6667	332,3333	0,8716
32768	579,0000	626,3333	0,9244
65539	1324,6670	1204,0000	1,1002

Tabela 3: Média e *Speed-up* dos tempos de execução do algoritmo do Produto Interno sequencial em C e paralelo C+CUDA com blocos de 256 *threads*

Qtd. de Dados	C(ms)	C+CUDA(ms)	*Speed-up*
1024	18,6667	44,6667	0,4179
2048	36,6667	73,6667	0,4977
4096	72,0000	74,0000	0,9730
8192	145,0000	110,3333	1,3142
16384	289,6667	184,6667	1,5686
32768	579,0000	341,3333	1,6963
65539	1324,6670	663,6667	1,9960

Tabela 4: Média e *Speed-up* dos tempos de execução do algoritmo do Produto Interno sequencial em C e paralelo C+CUDA com blocos de 512 *threads*

Qtd. de Dados	C(ms)	C+CUDA(ms)	*Speed-up*
1024	18,6667	39,3333	0,4746
2048	36,6667	45,0000	0,8148
4096	72,0000	54,0000	1,3333
8192	145,0000	73,6667	1,9683
16384	289,6667	113,0000	2,5634
32768	579,0000	184,6667	3,1354
65539	1324,6670	331,3333	3,9980

Analisando os tempos de execução dos algoritmos paralelo e sequencial apresentados nas tabelas anteriores, é possível notar a importância da quantidade de *threads* nos blocos. Com 128 *threads* por bloco o algoritmo paralelo consegue apresentar um desempenho melhor que o sequencial a partir do arquivo com 65539 dados. Ao alterar para 256 *threads* o ganho de desempenho começa ocorrer com 8192 dados. Já com 512 o algoritmo sequencial ultrapassa o tempo de execução do paralelo com 2048 dados. Esta diferença nos tempos é melhor visualizada no gráfico da figura 31. O programa C+CUDA com configuração de 512 *threads* por bloco apresenta o melhor desempenho dentre as demais configurações.

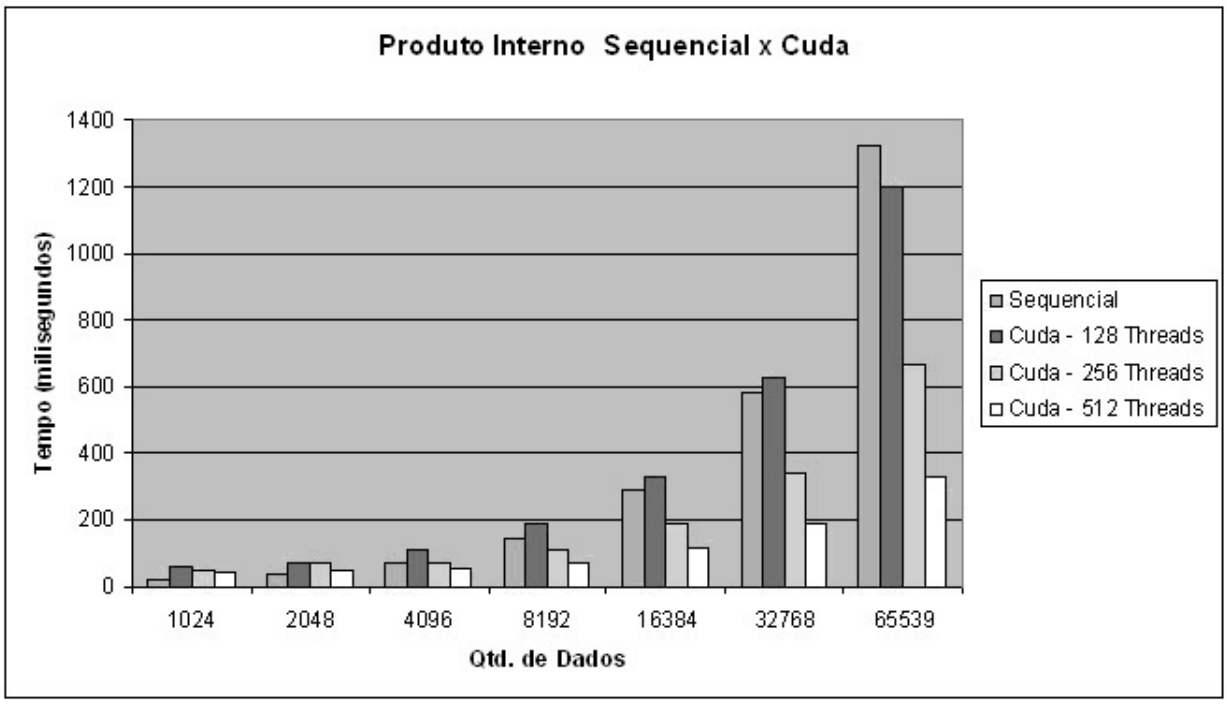


Figura 31: Gráfico em barras de comparação dos tempos de execução do Produto Interno Sequencial em C e paralelo C+CUDA.

O gráfico da figura 31 mostra que quando a quantidade de dados é pequena a paralelização com CUDA dos dados não é eficiente, sendo melhor a utilização do algoritmo sequencial. Mas à medida que a quantidade de dados cresce o algoritmo CUDA apresenta pequeno crescimento no tempo de execução comparado com o sequencial. Com isso, é possível concluir que a paralelização do Produto Interno é válida para grandes massas de dados, sendo dispensável para pequenas quantidades.

O desempenho expressivo do programa C+CUDA com 512 *threads* em relação aos demais se dá pela quantidade de blocos gerados para execução. Por exemplo, quando a massa de dados é de 65536 o algoritmo com 512 *threads* será executado com 128 blocos, o com 256 *threads* terá 256 blocos e o com 128 *threads* terá 512 blocos. O algoritmo apresenta um *loop* entre os blocos, chamadas ao *Kernel*, sendo este trecho considerado o gargalo do programa paralelo. Desta forma quanto maior a quantidade de blocos, mais chamadas serão feitas ao *Kernel*, realizando assim mais operações em relação a uma quantidade menor de blocos, implicando em um maior tempo de execução do programa.

No gráfico da figura 32 é realizado um comparativo entre os *speed-ups* obtidos nas diferentes massas de dados e configuração de *threads* por bloco.

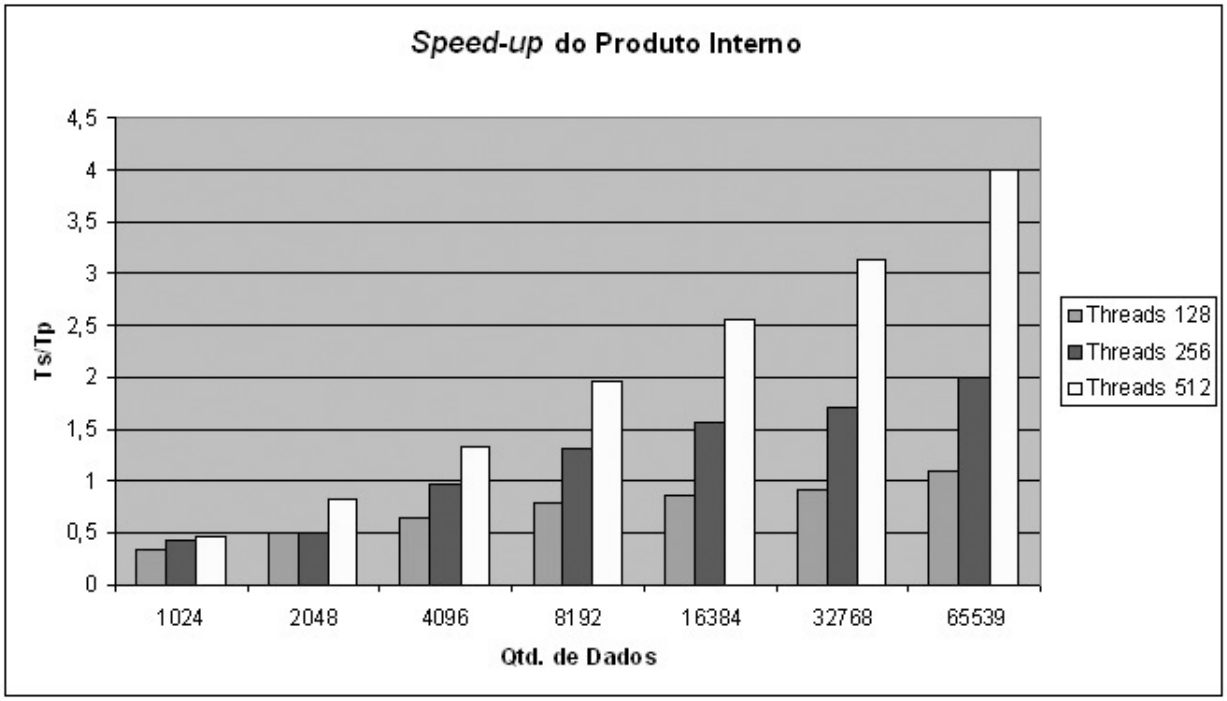


Figura 32: Gráfico de barras comparando o *speed-up* em cada configuração de *thread*.

É possível analisar que o ganho de desempenho é muito expressivo com a configuração de 512 *threads*. Após todos os testes realizados no algoritmo implementado em C+CUDA do Produto Interno foi possível conseguir um *speed-up* de quase 4 na configuração de 512 *threads* com uma massa de dados igual a 65539, isto é, o programa paralelo C+CUDA conseguiu ser quase 4 vezes mais rápido que o seu equivalente sequencial.

Com a pequena amostra utilizada para testes de desempenho do algoritmo CUDA do Produto Interno foi possível conseguir e mostrar um ganho considerável de desempenho. Com essa análise é possível perceber que algoritmos que utilizam a operação de Produto Interno, podem ser otimizadas com a paralelização desta função. Este estudo será realizado na seção 9.3.

9.2.2 Produto Matriz Vetor

Os testes de desempenho dos algoritmos sequencial na linguagem C e o paralelo em C+CUDA do Produto Matriz Vetor foram realizados no ambiente descrito na seção 9.1. Os tempos de execução obtidos são referentes apenas a execução da função do cálculo do Produto Matriz Vetor, sendo desprezados os tempos de leitura dos dados e alocação das variáveis do problema.

Os valores dos vetores de entrada são do tipo *float* variando entre 0 e 1, utilizando 4 casas decimais de precisão. A escolha do tipo e a precisão dos dados foram escolhidos de acordo com o suporte da arquitetura CUDA, desta forma é possível obter maior precisão dos dados calculados. Os dados foram salvos em arquivos binários, para a otimização do espaço físico, gravação e leitura dos dados pelo programa. Os arquivos referentes as matrizes do problema foram gerados de formas diferentes para os tipos de algoritmo. Para a implementação sequencial os dados foram gravados no formato de uma matriz, já para o paralelo os dados foram gravados no formato de um vetor. Essa diferença ocorre devido a diferença de tratamento da matriz nos algoritmos implementados. Assim uma matriz de dimensões 1024x128 utilizada no programa sequencial é equivalente a um vetor de 131072 no programa paralelo. Mesmo com a diferenciação na forma de gravação dos dados, em ambos os arquivos os valores dos dados são os mesmos.

A obtenção dos tempos de execução dos algoritmos e a variação da quantidade de *threads* por blocos segue o padrão aplicado no teste de desempenho do Produto Interno da seção 9.2.1.

As tabelas 5, 6 e 7 mostram a relação da quantidade de dados com os tempos obtidos em cada algoritmo e o *speed-up* calculado através da divisão do tempo de execução do programa sequencial pelo tempo de execução do programa paralelo. A coluna *"Qtd. de Dados na Matriz"* refere-se ao tamanho do vetor alocado para o algoritmo paralelo, esse valor corresponde a multiplicação da quantidade de linhas pela quantidade de colunas da matriz.

Tabela 5: Média e *Speed-up* dos tempos de execução do algoritmo do Produto Matriz Vetor sequencial em C e paralelo C+CUDA com blocos de 128 *threads*

Matriz x Vetor	Qtd. de Dados na Matriz	C(ms)	C+CUDA(ms)	*Speed-up*
128x1024 x 1024	131072	1986,3333	37,3333	53,2054
1024x128 x 128	131072	2007,0000	37,6667	53,2832
256x1024 x 1024	262144	4255,3333	42,3333	100,5197
1024x256 x 256	262144	4253,0000	42,0000	101,2619
512x1024 x 1024	524288	8271,3333	44,6667	185,1791
1024x512 x 512	524288	8281,6667	44,3333	186,8045
128x2048 x 2048	262144	6423,6667	42,0000	152,9444
2048x128 x 128	262144	6509,6667	42,3333	153,7716
256x2048 x 2048	524288	8217,6667	45,6667	179,9489
2048x256 x 256	524288	8358,6667	44,6667	187,1343
512x2048 x 2048	1048576	13247,6667	51,0000	259,7582
2048x512 x 512	1048576	13723,6667	51,3333	267,3441

Tabela 6: Média e *Speed-up* dos tempos de execução do algoritmo do Produto Matriz Vetor sequencial em C e paralelo C+CUDA com blocos de 256 *threads*

Matriz x Vetor	Qtd. de Dados na Matriz	C(ms)	C+CUDA(ms)	*Speed-up*
128x1024 x 1024	131072	1986,3333	38,0000	52,2719
1024x128 x 128	131072	2007,0000	39,0000	51,4615
256x1024 x 1024	262144	4255,3333	42,3333	100,5197
1024x256 x 256	262144	4253,0000	41,6667	102,0720
512x1024 x 1024	524288	8271,3333	45,0000	183,8074
1024x512 x 512	524288	8281,6667	45,0000	184,0370
128x2048 x 2048	262144	6423,6667	43,3333	148,2385
2048x128 x 128	262144	6509,6667	42,3333	153,7716
256x2048 x 2048	524288	8217,6667	45,3333	181,2720
2048x256 x 256	524288	8358,6667	45,0000	185,7481
512x2048 x 2048	1048576	13247,6667	51,3333	258,0714
2048x512 x 512	1048576	13723,6667	50,6667	270,8618

Tabela 7: Média e *Speed-up* dos tempos de execução do algoritmo do Produto Matriz Vetor sequencial em C e paralelo C+CUDA com blocos de 512 *threads*

Matriz x Vetor	Qtd. de Dados na Matriz	C(ms)	C+CUDA(ms)	*Speed-up*
128x1024 x 1024	131072	1986,3333	37,6667	52,7345
1024x128 x 128	131072	2007,0000	37,6667	53,2832
256x1024 x 1024	262144	4255,3333	42,3333	100,5197
1024x256 x 256	262144	4253,0000	42,3333	100,4646
512x1024 x 1024	524288	8271,3333	45,0000	183,8074
1024x512 x 512	524288	8281,6667	45,0000	184,0370
128x2048 x 2048	262144	6423,6667	42,0000	152,9444
2048x128 x 128	262144	6509,6667	41,3333	157,4919
256x2048 x 2048	524288	8217,6667	45,0000	182,6148
2048x256 x 256	524288	8358,6667	45,0000	185,7481
512x2048 x 2048	1048576	13247,6667	50,6667	261,4671
2048x512 x 512	1048576	13723,6667	51,0000	269,0915

Após a analise das tabelas 5, 6 e 7 é possível perceber a diferença expressiva entre os tempos de execução dos algoritmos sequencial e paralelo C+CUDA. Como houve uma diferença muito grande entre o sequencial e paralelo, para melhor visualização dos tempos do gráfico da figura 33, o eixo y está representado na escala logarítmica. Além da grande diferença nos tempos de execução dos algoritmos é possível perceber que não houve diferença entre as configurações da quantidade de *threads* por bloco, como ocorreu no algoritmo do Produto Interno. As execuções das matrizes MxN e NxM não apresentaram diferenças no desempenho em ambos os algoritmos implementados.

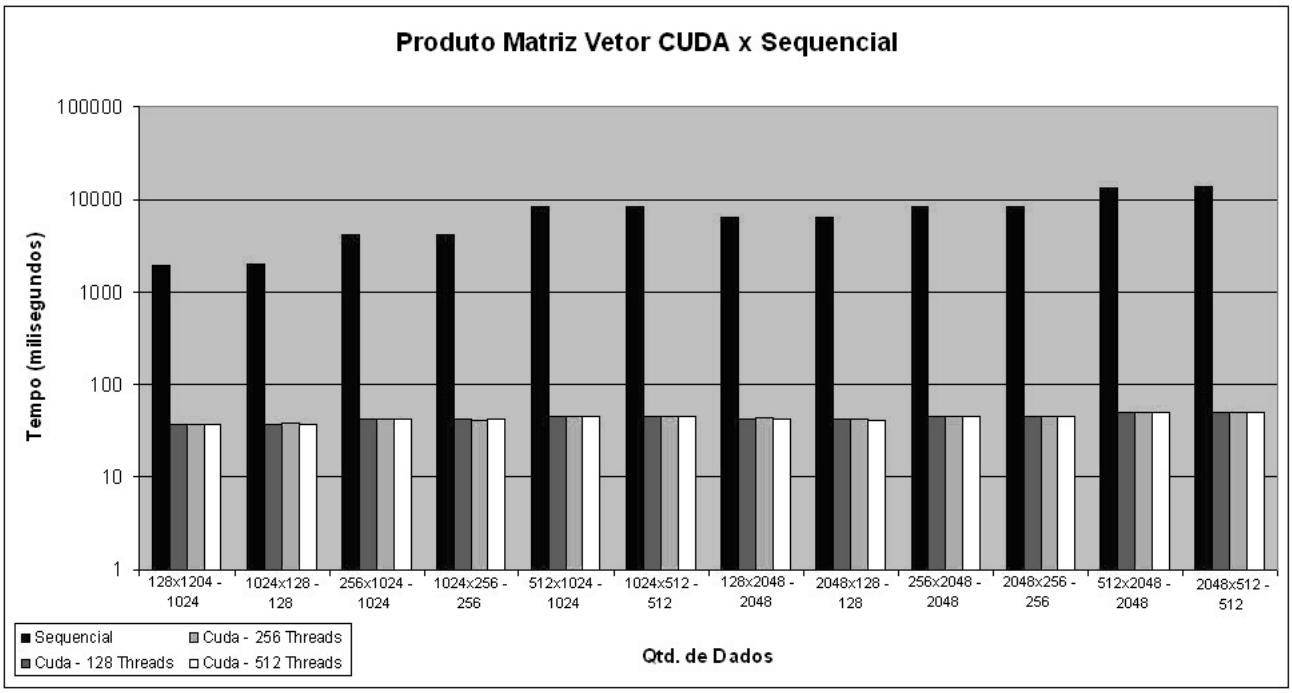


Figura 33: Gráfico de barras representando os tempos de execução dos algoritmos paralelos C+CUDA e sequencial na linguagem C

Logo no começo dos testes o algoritmo paralelo foi mais rápido que o sequencial. Com uma matriz de dimensões 128x1024 e um vetor com 1024 elementos o algoritmo paralelo com 128 *threads* chegou a ser mais de 53 vezes mais rápido que o sequencial.

No gráfico da figura 34 é representada as curvas do *speed-up* calculado entre os algoritmos paralelo e sequencial. Em todos os casos de testes o algoritmo C+CUDA apresentou grande ganho de desempenho. A configuração mais lenta, com 256 *threads* com 128x1024 x 1024, conseguiu ser mais de 52 vezes mais rápido que o sequencial. Já a configuração mais rápida, com 256 *threads* com 2048x512 x 512, chegou a ser mais de 270 vezes mais rápido que o sequencial.

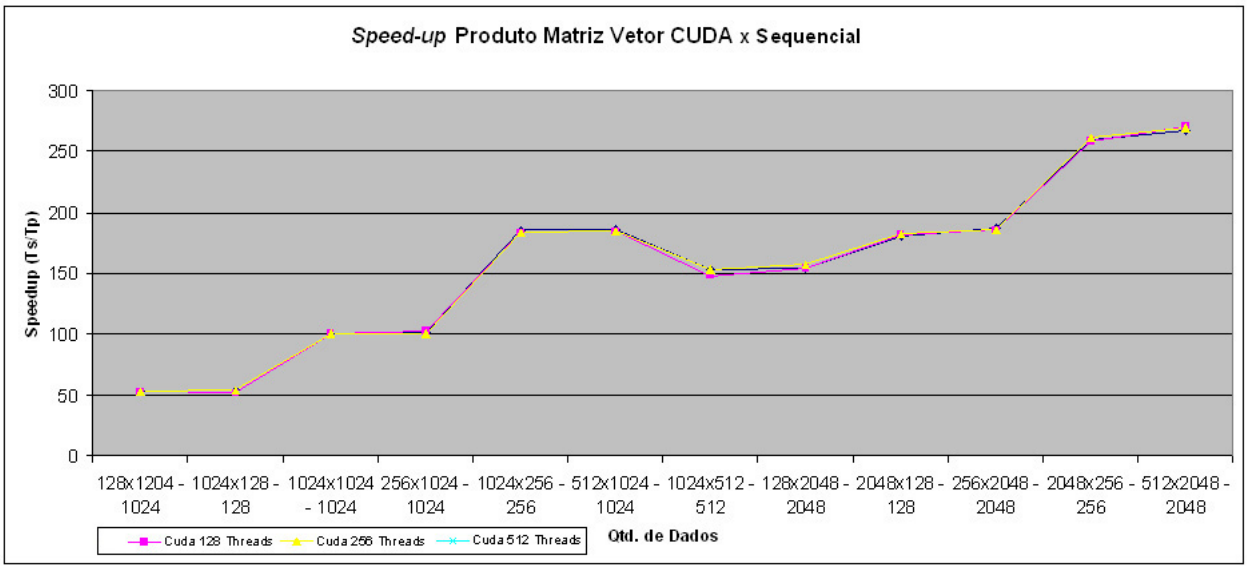


Figura 34: Gráfico de linhas representando o *Speed-up* obtidos nos testes do Produto Matriz Vetor.

Com uma pequena amostra utilizada para testes de desempenho do algoritmo CUDA do Produto Matriz Vetor foi possível conseguir e mostrar um ganho considerável de desempenho. Com essa análise é possível perceber que algoritmos que utilizam a operação de Produto Matriz Vetor, podem ser otimizadas com a paralelização desta função. Este estudo será realizado na seção 9.3.

9.3 Estudo de Caso

Com a analise dos testes de desempenho das operações de Produto Interno e Produto Matriz Vetor realizada nas subseções 9.2.1 e 9.2.2, respectivamente, é possível notar que a paralelização destas funções utilizando CUDA+C apresentam resultados expressivos nos tempos de execução em relação aos seu equivalente sequencial na linguagem C. Assim estes resultados juntamente com o breve estudo sobre sistemas lineares realizado na seção 8.1 serão utilizados na próxima subseção para a elaboração da projeção da paralelização das operações utilizadas na resolução do Método dos Gradientes Conjugados.

9.3.1 Análise e Projeção de Execução

O método dos gradientes conjugados será foco de análise neste trabalho devido a complexidade e as operações envolvidas na sua utilização na resolução de proble-

mas lineares de forma iterativa. O gradiente conjugado é basicamente o método das direções conjugadas que consiste na seleção de sucessivos vetores de direção como uma versão conjugada dos sucessivos gradientes encontrados ao longo do processo de solução. A cada iteração do método são realizados dois produtos internos para a obtenção de dois escalares definidos de forma que a sequência obedeça condições de ortogonalidade estabelecidas pelo próprio método. Além da operação do produto interno são realizadas multiplicações entre matrizes e vetores buscando a anulação do resíduo para a obtenção do resultado aproximado do sistema [3].

A ideia da análise do programa é verificar quais das funções representa o gargalo para a execução, isto é, em qual das funções o programa gasta mais tempo durante a execução. Desta forma, é possível identificar qual ponto do programa é candidato a ser paralelizado utilizando CUDA. Para essa análise foi utilizada a ferramenta **GProf** nativa do compilador **gcc**. Antes de realizar a análise dos dados coletados sobre a execução do programa é necessário realizar um breve estudo sobre a ferramenta utilizada para a coleta dos dados de desempenho do algoritmo implementado.

O **GProf** é uma ferramenta de ***Profile*** que permite a geração e análise de perfis de execução de programas. Com a sua utilização é possível obter uma análise da desempenho do algoritmo exibindo os resultados na forma de grafo. Com os resultados gerados é possível conhecer a quantidade de funções existentes no programa, a quantidade de vezes que cada uma dessas funções são chamadas e a porcentagem de tempo gasto em cada uma dessas chamadas. A grande vantagem do **GProf** é a geração de uma visão do comportamento de todo o sistema. Realiza a coleta das informações durante a execução real do programa, sendo de grande utilidade em programas muito grandes ou complexos para se analisar pelo código-fonte [4] [36]. A utilização do **GProf** mais comum é na análise de programas para otimização, incluindo técnicas de paralelização como é o caso deste trabalho.

O **GProf** faz parte do **gcc** e para a sua utilização é necessário realizar os seguintes passos de acordo com [4]:

1. O código deve ser compilado com o parâmetro **-pg**, o qual irá habilitar o compilador para executar o código de forma a realizar as medições. A linha de comando utilizada para a compilação do código do método dos gradientes conjugados foi *"gcc -pg MGC.c -o MGC.exe"* (vide código completo no anexo C);

2. Ao executar o programa é gerado um arquivo **gmon.out**, no qual contêm todas

as informações da execução do programa. Essas informações são interpretadas pelo **GProf**. Para visualizar os dados gerados da execução do programa MGC.c foi executado o comando *"gprof MGC.exe gmon.out> analise.txt"*. A saída foi desviada para o arquivo **analise.txt** para melhor visualização dos dados.

Após o entendimento do funcionamento da ferramenta **GProf** é possível realizar a análise do comportamento do programa do método dos gradientes conjugados. Como parâmetros de entrada foram utilizados uma matriz simétrica e definida positiva ([43]) de dimensões 1024x1024 e um vetor de dimensão 1024. Os dados contidos em ambas as estruturas são do tipo **float** com valores variando entre 0 e 1 com precisão de 4 casas decimais. O resultado da análise está representado na figura 35.

```
1  Each sample counts as 0.01 seconds.
2    %   cumulative   self              self     total
3   time   seconds   seconds    calls  ms/call  ms/call  name
4  100.19      0.04     0.04        2    20.04    20.04  ProdutoMatrizVetor
5    0.00      0.04     0.00        5     0.00     0.00  ProdutoInterno
6    0.00      0.04     0.00        3     0.00     0.00  abrirArquivo
7    0.00      0.04     0.00        1     0.00     0.00  lerDadosArquivoMatriz
```

Figura 35: Resultado da análise de desempenho do programa MGC

A coluna *"% Time"* apresenta os percentuais do tempo total correspondente a cada função listada na coluna *"name"*. As colunas *"self seconds"* e *"cumulative seconds"* apresentam, respectivamente, o tempo total gasto na execução da função e o tempo acumulado pela função em questão e por todas as demais que estão acima desta. A coluna *"calls"* mostra a quantidade de chamadas realizadas a cada função ao longo da execução do programa. A coluna *"self ms/call"* informa o tempo médio, em milissegundos, gastos em cada uma das chamadas à função. Já a coluna *"total ms/call"* informa o tempo médio, em milissegundos, gastos na execução da função e das funções chamadas no decorrer da sua execução [21]. Após a obtenção das informações da execução do programa MCG.c é possível realizar a análise e projeção da utilização de uma das funções, produto interno ou produto matriz vetor, implementadas em C+CUDA.

Analisando os resultados apresentados na figura 35 é possível destacar as funções **ProdutoMatrizVetor** e **ProdutoInterno** que são responsáveis diretas pelo processo de resolução do Método dos Gradientes Conjugados. É possível identificar na coluna *"call"* que ao longo de toda a execução do programa MCG (vide Anexo C) as funções foram chamadas 2 e 5 vezes, respectivamente. No caso a quantidade de

chamadas não é um ponto relevante na analise, pois a função **ProdutoMatrizVetor** foi a menos utilizada e a que mais consumiu o tempo de execução do programa. Isso ocorreu pois a operação do Produto Matriz Vetor apresenta complexidade maior que a do Produto Interno (vide subseções 8.2 e 8.3), além de envolver uma maior quantidade de dados na operação. O consumo médio do Produto Matriz Vetor ultrapassou os 100% devido a erros de cálculos e aproximações da máquina, mas fica bem claro que a função **ProdutoMatrizVetor** é uma forte candidata a ser paralelizada. Essa escolha ocorre pelo fato desta função ser o gargalo do programa MGC, isto é, praticamente todo o tempo de execução do programa é consumido pela função prejudicando o desempenho.

Desta forma, de acordo com os testes realizados na seção 9.2.2 a paralelização do Produto Matriz Vetor utilizando C+CUDA atinge grandes *speed-ups* podendo ser utilizado na paralelização desta função no programa MGC. Assim levando em consideração fatores que virão a interferir no ganho de desempenho (chamadas ao sistema, acesso a disco dentre outros), é possível considerar que, com a paralelização da função **ProdutoMatrizVetor**, o programa fique até 50 vezes mais rápido que o seu equivalente sequencial.

Esta projeção pode ser entendida aos demais algoritmos dos métodos que tenham como base de resolução as operações de Produto Interno e Produto Matriz Vetor. Com isso é possível identificar o grande potencial que a GPU tem para proporcionar um ganho significativo de desempenho na execução de programas que necessitem destes tipos de algoritmos [33].

10 CONCLUSÃO

Neste trabalho foi realizado um estudo da nova tecnologia para desenvolvimento de aplicações maciçamente paralelas que utilizam a Unidade de Processamento Gráfico (GPU), o CUDA. A utilização desta arquitetura para a realização dessas aplicações não necessita de nenhum conhecimento, pelo menos inicialmente, de como funciona o processamento gráfico. Desta forma, todas as questões gráficas envolvidas no processo são abstraídas e, além disso, o processo de comunicação entre os dispositivos (CPU e GPU) envolvidos é transparente para o programador.

Realmente a utilização desta arquitetura faz com que a curva de aprendizagem seja mais amena em relação às linguagens de programação na GPU anteriores, mas isso não quer dizer que não existam dificuldades. Questões como indexação de *threads*, construção de blocos, a forma como o programa é executado, preocupações com desempenho entre outras, são bem complexas. A programação paralela de uma GPU não se assemelha a Programação Paralela em CPU. Em uma CPU o programa é dividido em partes distintas que são independentes entre si. Após a execução destas partes, o resultado é unido e a execução continua. No caso da GPU, uma chamada gera várias *threads* que executarão as mesmas instruções diferenciando apenas nos dados manipulados. Por isso, questões de acesso inválido de memória, principalmente quando os dados se diferem apenas de posições distintas de um vetor, é uma grande preocupação. Outra grande dificuldade é a falta de sincronização entre *threads* de blocos distintos. Não existem formas simples de realizar esta sincronização através, por exemplo, de alguma diretiva da própria arquitetura CUDA, esse tipo de sincronização deve ser contemplada na implementação do algoritmo paralelo.

A falta de precisão é outro ponto negativo da execução em uma GPU. Dados com mais de sete dígitos não puderam ser utilizados nos teste, pois faziam com que a GPU, após os cálculos, perdesse a precisão e gerasse uma diferença de resultados comparados a execução pela CPU.

Apesar das dificuldades e dos pontos negativos, é certo que há um ganho de desempenho significativo entre a execução de um algoritmo sequencial e um executado paralelamente pela GPU com a utilização da arquitetura CUDA. Os teste apontaram em um *speed-up* máximo de 270.8618 na implementação do Produto Matriz Vetor, com uma matriz de dimensões 2048x512 e um vetor de 512 posições, isto significa que o algoritmo paralelo nesta situação conseguiu ser mais de 270 vezes mais rápido que o seu equivalente sequencial. Um fato a ser considerado é que houve ganho de desempenho apenas quando a quantidade de dados utilizados era grande. Prova-se, assim, que paralelizar algoritmos a serem executados pela GPU que utilizam pouca quantidade de dados não é uma boa alternativa, por apresentar desempenho inferior ao equivalente sequencial.

Além do ganho desempenho, outra prova de que a utilização da GPU para processamento de dados gerais está em expansão no mercado é o fato de já serem utilizadas na construção de supercomputadores, alguns deles muito bem colocados no Top500. Vale lembrar, que a utilização da GPU não descarta a CPU. A própria arquitetura CUDA só funciona com a utilização dos dois dispositivos.

Para trabalhos futuros, o método utilizado neste trabalho pode ser de fato paralelizado e testado a fim de provar as projeções aqui realizadas. Outro ponto muito interessante de pesquisa é a verificação da precisão das novas GPUs lançadas no mercado. Além disso, pode ser feito um estudo sobre como a configuração dos blocos de *threads* pode afetar o desempenho dos Algoritmos . Indo mais além, poderia ser verificado como GPUs podem trabalhar em conjunto com outras GPUs.

REFERÊNCIAS

[1] Adrenaline: Tecnologia - **NVIDIA auxilia a desvendar os mistérios do universo**. Disponível em: <http://www.adrenaline.com.br/tecnologia/noticias/3717/nvidia-auxilia-a-desvendar-os-misterios-do-universo.html>. Acesso em: JUN/2010

[2] ASSOCIATION, IEEE Standards. **IEEE 754:** Standard for Binary Floating-Point Arithmetic. Disponível em: <http://grouper.ieee.org/groups/754/>. Acesso em: MAIO/2010

[3] BECKER, Camila; PAZOS, Ruben Edgardo Panta; CROSSETTI, Geraldo Lopes. **Método de Gradiente Conjugado na otimização de problemas modelados na catalização de polímeros**. Universidade de Santa Cruz do Sul, UNISC. Santa Cruz do Sul, RS. 2006

[4] CHAVES, Luciano. **Discas para facilitar a depuração de programas**. Instituto de Computação, Universidade Estadual de Campinhas. 6 se setembro de 2010.

[5] CORPORATION, N. **CUDA - CUBLAS Library**, 2007.

[6] CORPORATION, N. **NVIDIA CUDA Programming Guide**, 20 de fevereiro de 2010. Versão 3.0.

[7] CORPORATION, N. **CUDA - CUFFT Lbrary**, 2007.

[8] CORPORATION, N. **NVIDIA CUDA Architecture**, 2009.

[9] CORPORATION, N. **NVIDIA CUDA Compute Unified Device Architecture:** Manual de Referencia. 2008.

[10] CORPOTATION N. **NVIDIA's Next Generation CUDA™ Compute Architecture:** Fermi™, 2009.

[11] CORPORATION, N. **NVIDIA Tesla Personal Supercomputer**, 2009.

[12] CORPORATION, N. **CUDA ZONE**.
Disponível em: <http://www.nvidia.com/object/cuda_home_new.html>. Acesso em: MAR/2010

[13] DANTAS, Daniel. **As GPU no processamento de vídeo em tempo real**. Universidade de São Paulo - USP, SP, 2006.

[14] DA PENHA, Dulcinéia Oliveira. **Análise Comparativa do Uso de Multi-Thread e OpenMP Aplicados a Operações de Convolução de Imagem**. Pontifícia Universidade Católica de Minas Gerais, BH, 200.

[15] DE PERNI, Paulo Henrique Ribeiro; **Um Estudo sobre Métodos Iterativos na Solução de Sistemas de Equações Provenientes de Métodos dos Elementos de Contorno**. 2002.Tese de Engenharia Civil, Universidade Federal do Rio de Janeiro - UFRJ. p. 35-46.

[16] DE ROSE, César A. F.; NAVAUX, Philippe O. A. Arquiteturas Paralelas. [S.l.]: Sagra Luzzatto, 2003.

[17] DONGARRA, Jack et al; **High Performance Computing Clusters, Constellations, MPPs, and Future Directions**. 2003. University of Tennessee.

[18] EPCC. **MSc in High Performance Computing**. Disponível em: <http://www.epcc.ed.ac.uk/msc>. Acesso em: NOVEMBRO/2010

[19] FERRAZ, Samuel B. **GPUs**. 2009. Faculdade de Computação, Universidade Federal de Mato Grosso do Sul - UFMS, Campo Grande.

[20] FISCHBORN, Marcos. **Computação de Alto Desempenho Aplicada à análise de Dispositivos Eletromagnéticos**. Tese (Doutorado em Engenharia Elétrica) - Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina - UFSC, Florianópolis, 2006.

[21] GOMES, André S. P.; MARTINS, Lucimara R.; VAZQUEZ, Pedro A. M. **Técnicas de Análise do Perfil de Execução e Otimização de Programas em Química Computacional**. Instituto de Química, Universidade Estadual de Campinas. 2001

[22] HALFHILL, T. R. **Parallel Processing With CUDA:** Nvidia's High-Performance Computing Platform Uses Massive Multithreading. [S.l.]: Microprocessor Report, Janeiro 2008.

[23] HESTENES, Magnus R.; STIEFEL Eduard.***Methods of conjugate gradients for solving linear systems***. *Journal of Research of the National Bureau of Standards*, Vol. 49, No. 6. (Dezembro 1952), pp. 409-436.

[24] KIPFER P., SEGAL M., WESTERMANN R.**UberFlow: A GPU-based particle engine**. Graphics Hardware 2004.

[25] L. Catabriga; A. M. P. Valli, B. Z. Melotti; et al. ***Performance of LCD Interative Method in the Finite Element and Finete Diference Solution of Convection-Diffusion Equations***. Departamento de Ciência da Computação da Unifersidade Federal do Espírito Santo - UFES e Centro de Computação Paralela e Departamento de Engenharia Civil (COPPE) da Universidade Federal do Rio de Janeiro - UFRJ. 2005.

[26] LARSEN E. S., MCALLISTER D.**Fast matrix multiplies using graphics hardware**. Anais da Conferência ACM/IEEE em Supercomputação. 2001.

[27] LOPES, Bruno Cardoso; AZEVEDO, Rodolfo Jardim de. **Computação de Alto Desempenho utilizando CUDA**. Universidade Estadual de Campinas - Unicamp, Campinas, 2008.

[28] MENENGUCI, Wesley dos Santos. **Computação de Alto Desempenho envolvendo clusters e métodos numéricos**. Centro Universitário de Vila Velha - UVV, 2008.

[29] MITTMANN, Adiel; COMUNELLO, Eros; WANGENHEIM, Aldo von. **Precisão e Exatidão Numéricas em Aplicações Médicas Baseadas em GPU**. Universidade Federal de Santa Catarina - UFSC, 2008.

[30] MOCELIN, Charlan Luís; SILVA, Eduardo Fialho Barcellos da; TOBALDINI, Geison Igor. **GPU:** Aspectos Gerais. Universidade Regional Integrada Campus de Erechim, Erechim, RS, 2009.

[31] NICKOLLS, John et al. **Scalable Parallel Programming**. [S.l]. 2008.

[32] NVIDIA GPU coding e Computing. [S.l]. [s.d.].

[33] OWENS, John D. et al. **A Survey of General-Purpose Computation on Graphics Hardware**. Dublin, Irlanda. 2005

[34] Painel do Hardware: Tecnologia em suas mãos - **Kaspersky utiliza tecnologia CUDA para acelerar verificação de vírus**. Disponível em: <http://paineldohardware.com/blog/2009/12/kaspersky-utiliza-tecnologia-cuda-para-acelerar-verificacao-de-virus/>. Acesso em: JUN/2010.

[35] PAMPLONA, Vitor. **Análise de Performance:** C vs Cuda. UFRGS, 2008. Disponível em: <http://vitorpamplona.com/wiki/An% C3% A1lise% 20de% 20Performance:% 20C% 20vs% 20Cuda>. Acesso em: MAIO/2010

[36] PERON, Rafael de Vasconcellos; MARQUES, Eduardo. **Otimização de código fonte C para o processador Nios II**. Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo - USP. 2007. p. 4-5

[37] SAAD, Youcef; SCHULTZ, Martin H. ***GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems***. Publicado no Jornal *SIAM Journal on Scientific and Statistical Computing*, volume 7 questão 3. Julho de 1986.

[38] TOP500. **TOP 500 Supercomputers sites**. Disponível em: <http://www.top500.org/>. Acesso em: NOVEMBRO/2010

[39] TRIOLET, Damien. **Nvidia CUDA:** preview - CUDA's API, 2007. Disnível em: <http://www.behardware.com/articles/659-4/nvidia-cuda-preview.html>. Acesso em: ABR/2010

[40] VASCONCELLOS, Felipe Brito. **Programando com GPUs:** Paralelizando o Método Lattice-Boltzmann com CUDA. Universidade Federal do Rio Grande do Sul - UFRGS, Porto Alegre, 2009.

[41] VERONESEM, Lucas et al. **Implementação Paralela em C+CUDA de uma Rede Neural Probabilística**. Universidade Federal do Espiríto Santo - UFES, Vitória, 2009.

[42] VIANA, José Ricardo. **Programação em GPU:** Passado, presente e futuro.[S.l]. [s.d.].

[43] Wikipédia - A enciclopédia Livre. **Matriz Positica Definida**. Disponível em: <http://pt.wikipedia.org/wiki/Matriz_positiva_definida>. Acesso em: NOV/2010

ANEXO A - Código desenvolvido em C+CUDA do Produto Interno

ProdutoInternoCuda.cu

```
♯include <stdio.h>
♯include <stdlib.h>

♯include <sys/time.h>

♯define THREADPERBLOCK 512
 ♯define BLOCKREDUCTION 1

/**** Protótipos das Funções da CPU ****************************/
FILE* abrirArquivo(char *nome_arquivo, char *modo_leitura);
/*************************************************************/

/**** Kernel ProdutoInterno - Executada na GPU ******************
Parâmetros:
<float *C : vetor contendo os elementos para serem somados>
<int tamVetor: tamanho do vetor a ser somado>
<int p: variával de controle da posição a ser somada do vetor>
<float *soma : variável para o armazenamento do resultado final da soma>
**/
__global__ void ProdutoInterno(float *C, int tamVetor, int p, float *soma)
{
        int j,k;

/*****************************************************************************************
```

*Este primeiro **for** é responsável por dividir vetor resultado **C** em duas partes iguais. E*

*a cada iteração o índice **k** será reduzido pela metade com o deslocamento do bit para direita. Esse processo de quebra do vetor resultado em duas partes iguais ocorrerá até chegar a um vetor de uma posição.*
*********/

```
for(k = tamVetor/2; k >0; k >>= 1)
{
        __syncthreads();
```

/**********

*Este segundo**for** será responsável pela soma das posições correspondentes dos vetores gerados pela divisão realizada no primeiro **for**. Só entrarão neste **for** as threads que apresentarem a coordenada x (índice) com valor menor que o tamanho do vetor após a divisão. Desta forma a indexação da segunda metade do vetor será realizada utilizando o índice da thread para garantir a correspondencia das posições das duas metades dos vetores mais a posição da divisão do vetor dada pela variável **k** e para realizar a localização dos dados no bloco é somada a variável de entrada do kernel **p**. Ao final da iteração a variável **j** será incrementada com a dimensão do bloco.*
*********/

```
        for(j = threadIdx.x; j <k; j += blockDim.x)
        {
                C[j+p] += C[k+j+p];
        }
}

__syncthreads();
```

/**********

*No fina da soma em árvore do vetor resultado a thread de índice 0 irá conter o resultado final desse somatório. Com o kernel pode ser chamado várias vezes de acordo com a quantidade de blocos a cada chamada será armazenado o valor parcial do produto interno na variável acumulativa **soma**. A variável **p** irá indicar em qual posição a redução começou, pois será esta posição que conterá o resultado parcial da redução.*
*********/

```
if(threadIdx.x==0)
```

```
        {
                *soma += C[p];
        }
}

/**** Kernel MultiplicaVetor - Executada na GPU ****************
Parâmetros:
<float *A : vetor contendo os elementos do problema>
<float *C : vetor contendo os elementos do problema>
<int N: tamanho dos vetores do problema>
**/
__global__ void MultiplicaVetor(float* A, float* B, float* C, int N)
{
 /*********************************************************************************************
A indexação das threads será dada pelo seguinte cálculo. No qual somente a coordenada x é considerada. É realizada a multiplicação da coordenada x da dimensão do bloco com a coordenada x do bloco em relação ao grid. Esse produto será somado com a coordenada x da thread que representa a sua localização em relação ao bloco. Com isso será possível acessar as threads dos blocos.
*********************************************************************************************/
        int i = blockDim.x * blockIdx.x + threadIdx.x;

/*********************************************************************************************
O controle realizado pelo if é para garantir que só serão utilizadas posições válidas na multiplicações dos vetores.
*********************************************************************************************/
        if(i<N)
        {
                C[i] = A[i] * B[i];
        }
}

/**** Função Principal ******************************************
Argumentos:
./ProdutoInternoCuda <vetorA_N.dat> <vetorB_N.dat>
```

```
**/
int main(int argc, char **argv)
{
        if(argc != 3) return 0;

       //Declaração das variáveis do problema do Host(CPU)
        float *vetorHostA;
        float *vetorHostB;
        float *vetorHostC;
        float *resul;
        float *resul2;

       //Declaração das variáveis do problema do Device(GPU)
        float *vetorGpuA;
        float *vetorGpuB;
        float *vetorGpuC;
        float *soma;

       //Variável para armazenar o tamanho dos vetores
        int N;

       //Variáveis correspondentes aos parâmetros de execução do kernel
        int threadsPerBlock;
        int numBlocks;

       //Variáveis índices
        int j, p, k;

       //Variáveis relativas ao cálculo do tempo de execução do Kernel
        struct timeval start, end;

       //Ponteiros para os arquivos utilizados no programa
        FILE *inA;
```

```
 FILE *inB;

//Abrindo os arquivos do programa
 inA = abrirArquivo(argv[1], "rb");
 inB = abrirArquivo(argv[2], "rb");

//Leitura dos parâmetros referentes aos dados lidos
 sscanf(argv[1], "%*[^_]_% d.dat", &N);

//Cálculo do tamanho dos dados
 size_t size = N * sizeof(float);
 size_t sizeF = sizeof(float);

//Alocação das variáveis na CPU
 vetorHostA = (float*)malloc(size);
 vetorHostB = (float*)malloc(size);
 vetorHostC = (float*)malloc(size);
 resul = (float*)malloc(sizeF);
 resul2 = (float*)malloc(sizeF);
 *resul = 0;

//Alocação das variáveis na GPU
 cudaMalloc((void**)&vetorGpuA, size);
 cudaMalloc((void**)&vetorGpuB, size);
 cudaMalloc((void**)&vetorGpuC, size);
 cudaMalloc((void**)&soma, sizeF);

//Preenchimento das variaveis da CPU
 fread(vetorHostA, sizeof(float), N, inA);
 fread(vetorHostB, sizeof(float), N, inB);

//Preenchimento das variaveis da GPU
 cudaMemcpy(vetorGpuA, vetorHostA, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(vetorGpuB, vetorHostB, size, cudaMemcpyHostToDevice);
cudaMemcpy(soma, resul, sizeF, cudaMemcpyHostToDevice);

//Cálculo da quantidade de blocos e de threads por blocos
if(N < THREADPERBLOCK )
{
        threadsPerBlock = N;
        numBlocks = (N + N - 1)/N;
}
else
{
        threadsPerBlock = THREADPERBLOCK;
        numBlocks = (N + THREADPERBLOCK - 1)/THREADPERBLOCK;
}

/**** Start do tempo ***************************************/

//Start do tempo de execução
gettimeofday(&start, NULL);

/*****************************************************************************************
Chamada ao kernel para multiplicar os elementos do vetores. Esse kernel é responsável pela geração do vetor resultado que conterá em cada posição a multiplicação das posições correspondentes dos vetores do problema.
*****************************************************************************************/
MultiplicaVetor<<< numBlocks, threadsPerBlock >>>
        (vetorGpuA, vetorGpuB, vetorGpuC, N);

/*****************************************************************************************
Como a arquitetura CUDA disponibiliza somente a sincronização entre threads de mesmo bloco, para garantir a sincronização das threads de blocos diferentes na realização do processo de redução é passada para o kernel um bloco de cada vez. Assim a cada iteração será obtido o resultado parcila do produto interno. Desta forma será garantida a integridade e consistencia dos dados e a sincronização das threads
```

*de blocos diferentes. Para localizar a posição a ser utiliza no processo de redução é contabilizada uma variável **p** que indicará em qual bloco as operações devem ser feitas. Para isso a cada iteração essa variável é incrementada com a quantidade de threads contidas em cada bloco.*

```
**********************************************************************************************/
        for(j = 0, p = 0; j < numBlocks; j++, p += threadsPerBlock)
        {
                ProdutoInterno<<< BLOCKREDUCTION, threadsPerBlock >>>
                        (vetorGpuC, threadsPerBlock, p, soma);
        }

       //End do tempo de execução
        gettimeofday(&end, NULL);

       /**** End do tempo ************************************/

       //Cópia do resultado da GPU para a CPU
        cudaMemcpy(resul2, soma, sizeF, cudaMemcpyDeviceToHost);

       //Exibe o resultado calculado
        printf("Resultado: % f \n \n", *resul2);

       //Cálculo do tempo de execução do Kernel
        printf("Tempo: % ld \n \n",
                        ((end.tv_sec * 1000000 + end.tv_usec) -
                        (start.tv_sec * 1000000 + start.tv_usec)));

       //Libera recursos do Host
        free(vetorHostA);
        free(vetorHostB);
        free(vetorHostC);
        free(resul);
        free(resul2);
```

```
            //Fecha os arquivos utilizados
             fclose(inA);
             fclose(inB);

            //Libera recursos da Gpu
             cudaFree(vetorGpuA);
             cudaFree(vetorGpuB);
             cudaFree(vetorGpuC);
             cudaFree(soma);

            return 0;
}

/**** Função abrirArquivo - Executada na CPU ****************
  Parâmetros:
   <char *nome_arquivo : Nome do arquivo a ser aberto>
  <char *modo_leitura : Modo de leitura do arquivo>
 Retorno:
<FILE* : ponteiro para o arquivo aberto>
**/
 FILE* abrirArquivo(char *nome_arquivo, char *modo_leitura)
 {              FILE *fp;

            if( (fp = fopen(nome_arquivo, modo_leitura)) == NULL)
             {
                        printf("Erro no arquivo!\n");
                        exit(1);
             }
            return fp;
 }
```

ANEXO B - Código desenvolvido em C+CUDA do Produto Matriz Vetor

ProdutoMatrizVetorCuda.cu

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

#include <sys/time.h>

#define THREADPERBLOCK 512

/**** Protótipos das Funções da CPU ************************/
FILE* abrirArquivo(char *nome_arquivo, char *modo_leitura);
void lerDadosArquivoMatriz(FILE *fp, float **matr, int m, int n);
/*****************************************************************/

/**** Função ProdutoInterno - Executada na GPU ****************
Parâmetros:
<float *C : vetor contendo os elementos para serem somados>
<int tamVetor: tamanho do vetor a ser somado>
<float *soma : variável para o armazenamento do resultado final da soma>
**/
__device__ void ProdutoInterno(float *C, int tamVetor, float *soma)
{
        int j,k;

        for(k = tamVetor/2; k >0; k >> = 1)
        {
```

```
                __syncthreads();

                for(j = threadIdx.x; j <  k; j += blockDim.x)
                {
                        C[j] += C[k+j];
                }
        }

        __syncthreads();

        if(threadIdx.x==0)
        {
                *soma = C[0];
        }
}

/**** Kernel MultiplicaMatriz - Executada na GPU ****************
Parâmetros: // <float *A : vetor que representa a matriz do problema>
<int numLinhas : quantidade de linhas da matriz A>
<int numColunas: quantidade de colunas da matriz A>
<float *R : vetor do problema>
<float *X : vetor que irá armazenar o resultado do produto entre matriz A e vetor R>
**/
__global__ void MultiplicaMatriz(float *A, int numLinhas, int numColunas, float *R, float
*X)
{
 /*************************************************************************************
Alocação do vetor resultado a ser utilizado para armazenar as multiplizações da linha
da matriz pelo vetor do problema. O vetor está sendo alocado na memória compartilha
e apresentará como tamanho a quantidade de threads de um bloco, com isso todos
as threads do bloco poderão acessar os seus dados e representá-los nas operações.
*************************************************************************************/
        __shared__ float T[THREADPERBLOCK];
        int ini, fim, i, j, k;
```

```
        for(i = blockIdx.x; i <numLinhas; i += gridDim.x)
        {
                ini = i * numColunas;
                fim = ini + numColunas;
                T[threadIdx.x] = 0.0;

                for(j = threadIdx.x + ini, k = threadIdx.x;
                        j <fim; j += blockDim.x, k += blockDim.x)
                {
                        T[threadIdx.x] += R[k] * A[j];
                }
                __syncthreads();

                ProdutoInterno(T, THREADPERBLOCK, & X[i]);
        }
}

/**** Função Principal ******************************************
Argumentos:
./ProdutoMatrizVetorCuda <matriz_M.dat> <vetor_N.dat>
**/
int main(int argc, char **argv)
{
        if(argc != 3) return 0;

       //Declaração das variáveis do problema do Host(CPU)
        float *matrizHost;
        float *vetorHost;
        float *vetorResulHost;

       //Declaração das variáveis do problema do Device(GPU)
        float *matrizGpu;
        float *vetorGpu;
        float *vetorResulGpu;
```

```
//Variáveis índices
 int i;
 int C, L, M;

//Variáveis correspondentes aos parâmetros de execução do kernel
 int threadsPerBlock;
 int numBlocks;

//Variáveis relativas ao cálculo do tempo de execução do Kernel
 struct timeval start, end;

//Ponteiros para os arquivos utilizados no programa
 FILE *inA;
 FILE *inB;

//Abrindo os arquivos do programa
 inA = abrirArquivo(argv[1], "rb");
 inB = abrirArquivo(argv[2], "rb");

//Leitura dos parâmetros referentes aos dados lidos
 sscanf(argv[1], "%*[^_]_% d.dat", &M);
 sscanf(argv[2], "%*[^_]_% d.dat", &C);

//Cálculo da quantidade de linhas da matriz
 L = M/C;

//Cálculo do tamanho dos dados
 size_t sizeV = C * sizeof(float);
 size_t sizeM = M * sizeof(float);

//Alocação das variáveis na CPU
```

```
 vetorHost = (float*)malloc(sizeV);
 vetorResulHost = (float*)calloc(C, sizeof(float));
 matrizHost = (float*)malloc(sizeM);

//Alocação das variáveis na GPU
 cudaMalloc((void**)&vetorGpu, sizeV);
 cudaMalloc((void**)&vetorResulGpu, sizeV);
 cudaMalloc((void**)&matrizGpu, sizeM);

//Preenchimento das variaveis da CPU
 fread(vetorHost, sizeof(float), C, inB);
 fread(matrizHost, sizeof(float), M, inA);

//Preenchimento das variaveis da GPU
 cudaMemcpy(vetorGpu, vetorHost, sizeV, cudaMemcpyHostToDevice);
 cudaMemcpy(matrizGpu, matrizHost, sizeM, cudaMemcpyHostToDevice);
 cudaMemcpy(vetorResulGpu, vetorResulHost, sizeV,
                cudaMemcpyHostToDevice);

//Determinando a quantidade de blocos e threads por blocos
 int qtdMaior = L<C? C : L;

//Cálculo da quantidade de blocos e de threads por blocos
 if(qtdMaior <THREADPERBLOCK )
 {
            threadsPerBlock = qtdMaior;
            numBlocks = (qtdMaior + qtdMaior - 1)/C;
 }
 else
 {
            threadsPerBlock = THREADPERBLOCK;
           numBlocks = (qtdMaior+THREADPERBLOCK-1)/THREADPERBLOCK;
 }
```

```
//Start do tempo de execução
 gettimeofday(&start, NULL);

//Chamada ao Kernel
 MultiplicaMatriz<<< numBlocks, threadsPerBlock >>>
                    (matrizGpu, L, C, vetorGpu, vetorResulGpu);

//End do tempo de execução
 gettimeofday(&end, NULL);

//Cópia do resultado da GPU para a CPU
 cudaMemcpy(vetorResulHost, vetorResulGpu, sizeV, )
                    cudaMemcpyDeviceToHost);

//Cálculo do tempo de execução do Kernel
 printf("Tempo: % ld \n \n",
             ((end.tv_sec * 1000000 + end.tv_usec) -
             (start.tv_sec * 1000000 + start.tv_usec)));

/**** Armazenamento do resultado ****************************/
 FILE *out = fopen("testeCUDA2.txt", "wt");

for(i = 0; i<L; i++)
 {
             fprintf(out, "% f \n", vetorResulHost[i]);
 }
 fclose(out);
 /***************************************************************/

//Libera recursos do Host(CPU)
 free(vetorHost);
 free(vetorResulHost);
 free(matrizHost);
```

```
            //Fecha os arquivos utilizados
             fclose(inA);
             fclose(inB);

            //Libera recursos da Device(GPU)
             cudaFree(matrizGpu);
             cudaFree(vetorGpu);
             cudaFree(vetorResulGpu);

            return 0;
}

/**** Função abrirArquivo - Executada na CPU ****************
  Parâmetros:
  <char *nome_arquivo : Nome do arquivo a ser aberto>
  <char *modo_leitura : Modo de leitura do arquivo>
 Retorno:
<FILE* : ponteiro para o arquivo aberto>
**/
 FILE* abrirArquivo(char *nome_arquivo, char *modo_leitura)
 {              FILE *fp;

            if( (fp = fopen(nome_arquivo, modo_leitura)) == NULL)
             {
                        printf("Erro no arquivo!\n");
                        exit(1);
             }
            return fp;
}
```

ANEXO C - Código desenvolvido na linguagem C do Método dos Gradientes Conjugados

MGC.c

```
♯include <stdio.h>
♯include <stdlib.h>

♯include <sys/time.h>

/**** Protótipos das Funções da CPU ***************************/
void lerDadosArquivoMatriz(FILE *, float **, int, int);
FILE* abrirArquivo(char *, char *);
float ProdutoInterno(float *, float *, float *, int);
void ProdutoMatrizVetor(float **, float *, float *, int);
/*****************************************************************/

/**** Função Principal ****************************************
Argumentos:
./MGC <matriz_M_N.dat> <vetor_N.dat>
**/
int main(int argc, char **argv)
{
        if(argv != 3)  return 0;


       //Variáveis índices
        int j, k;
```

```
//Variável para armazenar o tamanho do vetor
 int N;

//Variáveis do problema
 float **A;
 float *b;
 float *x;
 float *v;
 float *r;
 float *p;

//Declaração e inicialização das variáveis do método
 float deltaNew = 0.0;
 float deltaAux = 0.0;
 float alfa = 0.0;
 float deltaOld = 0.0;
 float beta = 0.0;
 float deltaIni = 0.0;

//Variáveis índices de iterações
 int i;
 int iter;

//Variável auxiliar
 float aux;

//Variáveis de controle das iterações do método
 float tol;
 int iteMax;

//Variáveis relativas ao cálculo do tempo de execução do Kernel
 struct timeval start, end;
```

```
//Ponteiros para os arquivos utilizados no programa
 FILE *inA;
 FILE *inB;
 FILE *out;

//Abrindo os arquivos do programa
 inA = abrirArquivo(argv[1], "rb");
 inB = abrirArquivo(argv[2], "rb");
 out = abrirArquivo("Resposta.txt", "wt");

//Leitura dos parâmetros referentes aos dados lidos
 sscanf(argv[2], "% *[^_]_% d.dat", &N);

//Alocação das variáveis
 A = (float**)malloc(N*sizeof(float*));

for(j = 0; j<N; j++)
        A[j] = (float*)malloc(N*sizeof(float));
b = (float*)malloc(N*sizeof(float));
 x = (float*)calloc(N,sizeof(float));
 v = (float*)calloc(N,sizeof(float));
 r = (float*)calloc(N,sizeof(float));
 p = (float*)calloc(N,sizeof(float));

//Definição da tolerância e da quantidade máxima de
 //iterações do método
 tol = 0.1;
 iteMax = 3;

//Preenchimento das variaveis
 fread(b, sizeof(float), N, inB);
 lerDadosArquivoMatriz(inA, A, N, N);
```

```
//Inicialização dos índices de iterações
 i = 1;
 iter = 0;

//Start do tempo de execução
 gettimeofday(&start, NULL);

/**** Início do Método do Gradiente Conjugado ********/
 for(j=0; j<N; j++)
 {
            v[j] = r[j] = b[j];
 }

ProdutoInterno(r, r, &deltaNew, N);

deltaIni = deltaNew;

while((i<iteMax) && (deltaNew>(tol*tol)*deltaIni))
 {
            ProdutoMatrizVetor(A, v, p, N);

           aux = 0.0;

           ProdutoInterno(v, p, &aux, N);
            alfa = deltaNew/aux;

           for(j=0; j<N; j++)
            {
                     x[j] = x[j] + (alfa*v[j]);
            }

           for(j=0; j<N; j++)
```

```
                {
                        r[j] = r[j] - (alfa*p[j]);
                }

                deltaOld = deltaNew;

                aux = 0.0;
                 ProdutoInterno(r, r, &aux, N);
                 deltaNew = aux;

                beta = deltaNew/deltaOld;

                for(j=0; j<N; j++)
                 {
                        v[j] = r[j] + (beta*v[j]);
                 }

                i++;
                 iter++;
 }

//End do tempo de execução
 gettimeofday(&end, NULL);
 /**** Fim do Método do Gradiente Conjugado **********/

//Cálculo do tempo de execução do Kernel
 printf("Tempo: % ld\n\n", ((end.tv_sec*1000000+end.tv_usec) -
 (start.tv_sec * 1000000 + start.tv_usec)));

//Armazenamento do resultado
 for(j=0;j<N;j++)
 {
                 fprintf(out, "% f \n", x[j]);
```

```
		}

		//Fecha os arquivos utilizados
		fclose(inA);
		fclose(inB);
		fclose(out);

		//Libera recursos
		for(j = 0; j<N; j++)
			free(A[jj]);
		free(A);
		free(b);
		free(x);
		free(v);
		free(r);
		free(p);

		return 0;
}

/**** Função ProdutoMatrizVetor ****************
Parâmetros:
<float **a : Matriz quadrada do problema>
<float *b : Vetor do problema>
<float *c : Vetor resultado do produto matriz vetor>
<int N : Representa a dimensão da matriz e do vetore>
**/
void ProdutoMatrizVetor(float **a, float *b, float *c, int N)
{
		int i, j;

		for(i = 0; i<N; i++)
		{
			for(j = 0; j<N; j++)
```

```
                {
                        c[i] += a[i][j]*b[j];
                }
        }
}

/**** Função ProdutoInterno ****************
Parâmetros:
<float *a : Vetor do problema>
<float *b : Vetor do problema>
<float *c : Variável com o resultado do produto interno>
<int N : Tamanho dos vetores>
**/
void ProdutoInterno(float *a, float *b, float *c, int N)
}
        int i;

        for(i = 0; i<N; i++)
        {
                *c += a[i]*b[i];
        }
}

/**** Função ProdutoInterno ****************
Parâmetros:
<FILE *fp : Ponteiro para o arquivo com os dados a serem lidos>
<float *matr : Ponteiro para matriz a ser preenchida>
<int m : Quantidade de linhas da matriz>
<int n : Quantidade de colunas da matriz>
Retorno:
<FILE* : ponteiro para o arquivo aberto>
**/
void lerDadosArquivoMatriz(FILE *fp, float **matr, int m, int n)
{
        int i;
```

```
        for(i = 0; i<m; i++)
                fread(matr[i], sizeof(float), n, fp);
}

/**** Função abrirArquivo - Executada na CPU ***************
 Parâmetros:
 <char *nome_arquivo : Nome do arquivo a ser aberto>
 <char *modo_leitura : Modo de leitura do arquivo>
**/
FILE* abrirArquivo(char *nome_arquivo, char *modo_leitura)
{       FILE *fp;

        if( (fp = fopen(nome_arquivo, modo_leitura)) == NULL)
        {
                printf("Erro no arquivo!\n");
                exit(1);
        }
        return fp;
}
```

ANEXO D - Simulação da Execução dos *Kernels* Responsáveis pelo Cálculo do Produto Interno CUDA+C

Para melhor entendimento da execução de um programa paralelo na plataforma CUDA, será realizada uma simulação da execução do Produto Interno. A simulação apresenta a seguinte configuração:

- Serão utilizados os seguintes vetores de entrada:

  vetorGpuA

3	1	5	1	2	1	0	1

  vetorGpuB

3	3	1	-1	2	2	0	1

- Número de blocos no grid (**numBlocks**): 1 blocos;
- Quantide de *threads* por bloco (**threadsPerBlock**): 8 *threds*.

O algoritmo paralelo do Produto Interno foi abordado em duas etapas:

1º Passo: Formação do vetor resultado a partir da multiplicação das posições correspondentes dos vetores do problema;

2º Passo: Realizar a Soma em Árvore do vetor resultado.

Para o 1º passo são realizadas as seguintes operações:

1.a O programa chama o *kernel*, (figura 20), responsável pela multiplicação das posições dos vetores para montar o vetor resultado (figura 21).
Assim temos a seguinte situação na figura 36, os vetores **vetorGpuA** e **vetorGpuB** contendo os dados do problema e o vetor resultado vazio para armazenar o resultado das multiplicações.

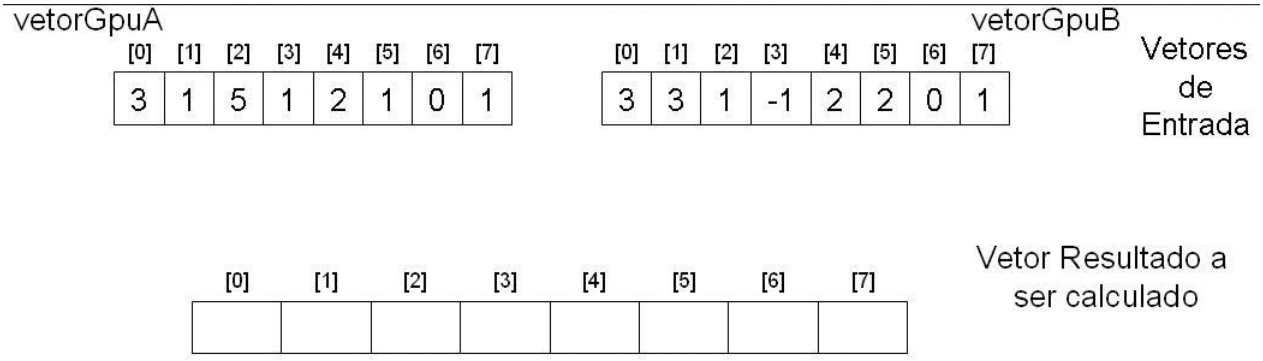


Figura 36: Vetores de entrada do *Kernel* **ProdutoInterno**.

1.b Na execução do *Kernel* cada posição do vetor é representada por uma *thread*. A indexação das posições dos vetores é realizada a partir das seguintes variáveis:

- **blockDim.x**: Corresponde a coordenada x da dimensão do bloco. Como o bloco apresenta dimensão 1x8 o valor dessa variável será igual a 0 (zero) seguindo o padrão de indexação da linguagem C;
- **blockIdx.x**: Corresponde a coordenada x da posição do bloco em relação ao grid. Como o grid contém um bloco, então as coordenadas do bloco em execução são (0,0);
- **threadIdx.x**: Corresponde a coordenada x da posição da *thread* em relação ao bloco.

As *threads* ilustradas na figura 37 apresentam apenas a coordenada [x] para simplificar a visualização da indexação dos vetores.

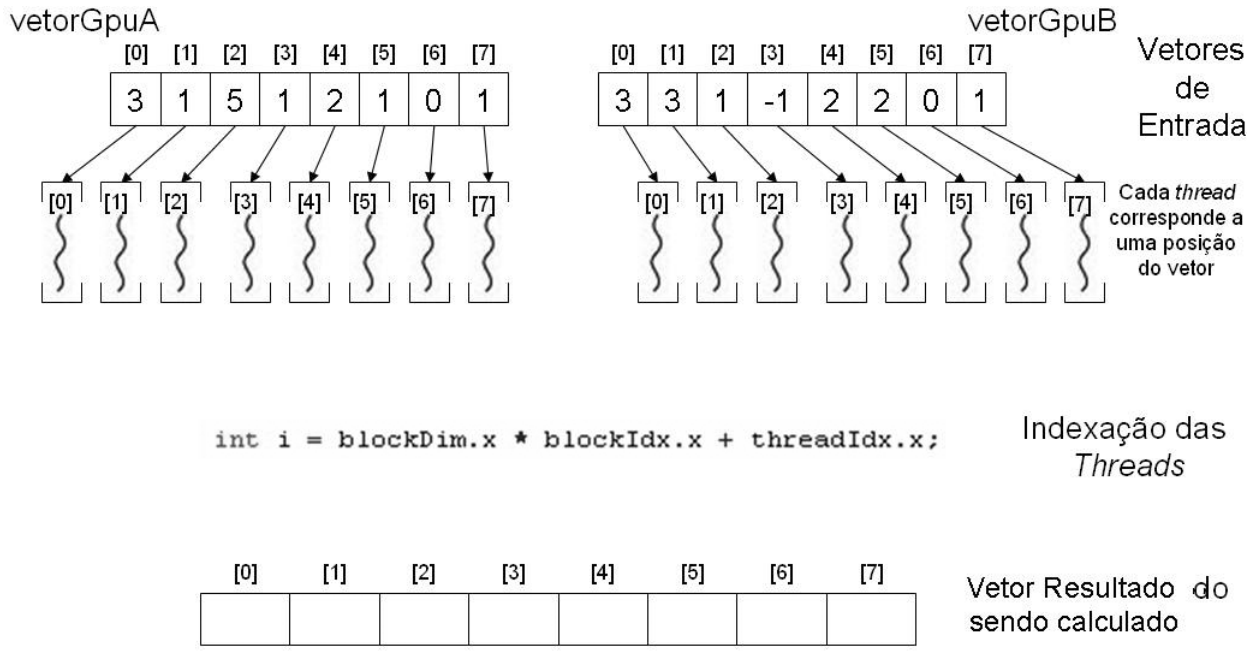


Figura 37: As posições dos vetores são representados por *threads*.

1.c A indexação das posições dos vetores ocorrerão de forma paralela seguindo o padrão das figuras abaixo:

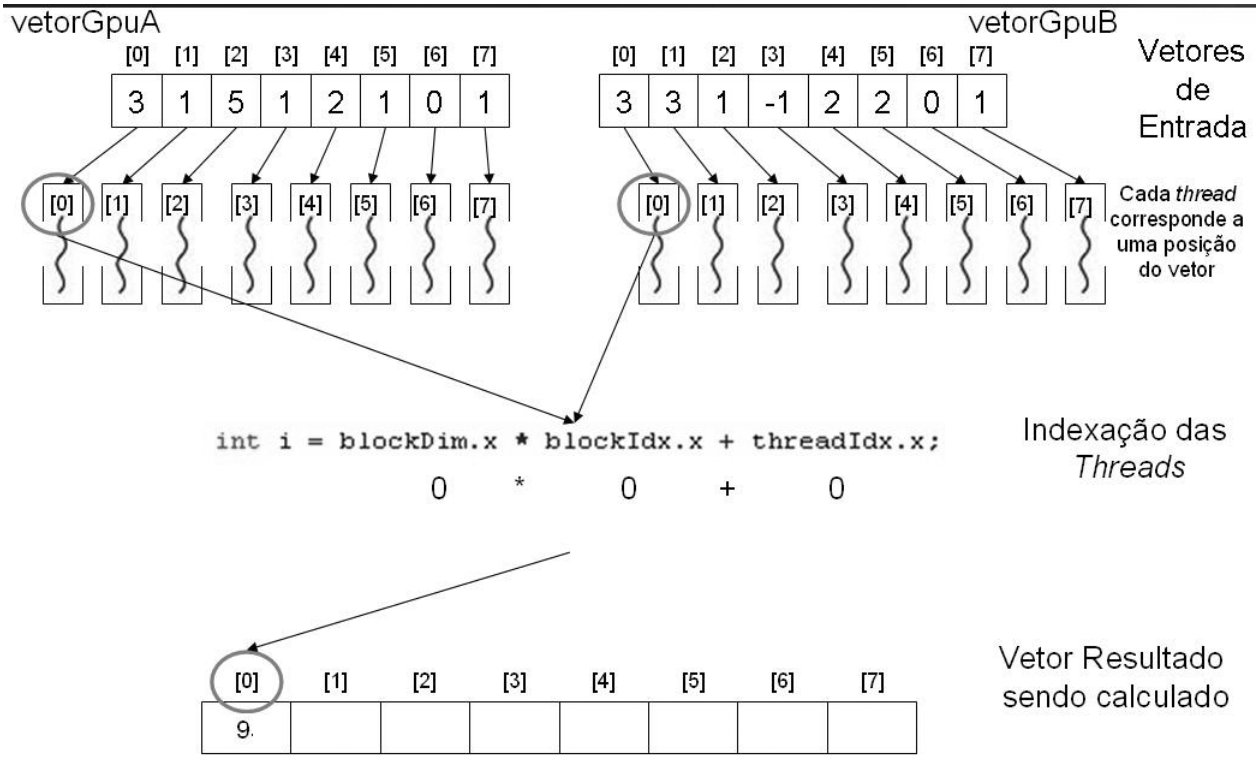


Figura 38: Multiplicação dos dados da posição 0 dos vetores de acordo com a indexação.

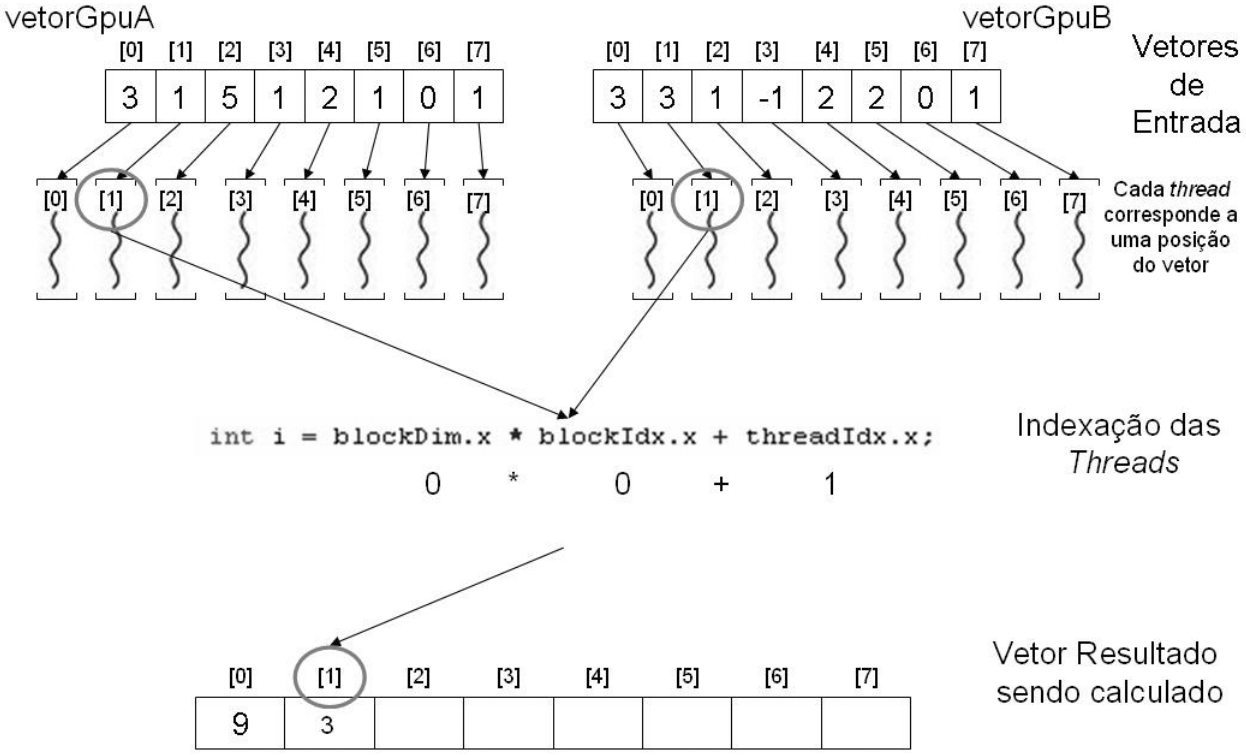


Figura 39: Multiplicação dos dados da posição 1 dos vetores de acordo com a indexação.

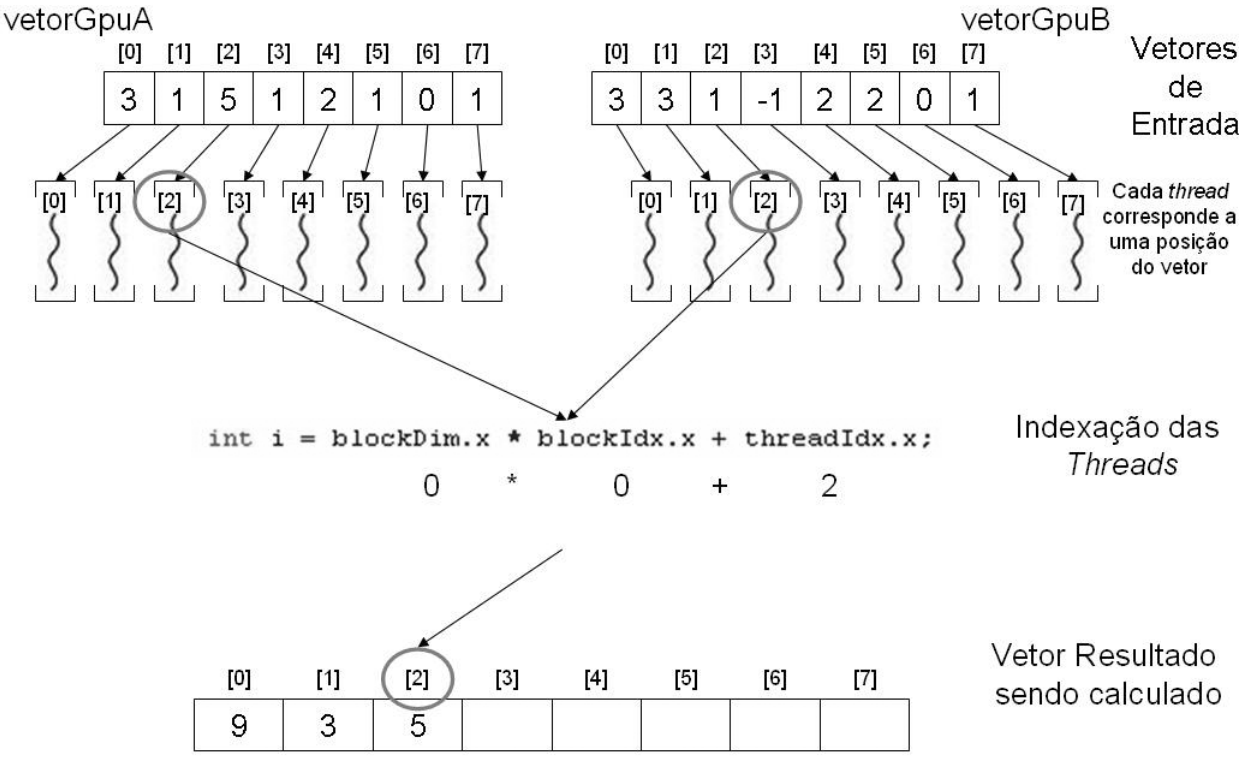


Figura 40: Multiplicação dos dados da posição 2 dos vetores de acordo com a indexação.

Como é possível perceber nas figuras 38, 39 e 40 a indexação é determinada

pela coordenada da *thread* em questão.

1.d Esse processo se repete até chegar a posição de índice 7 dos vetores, neste momento o vetor resultado estará formado:

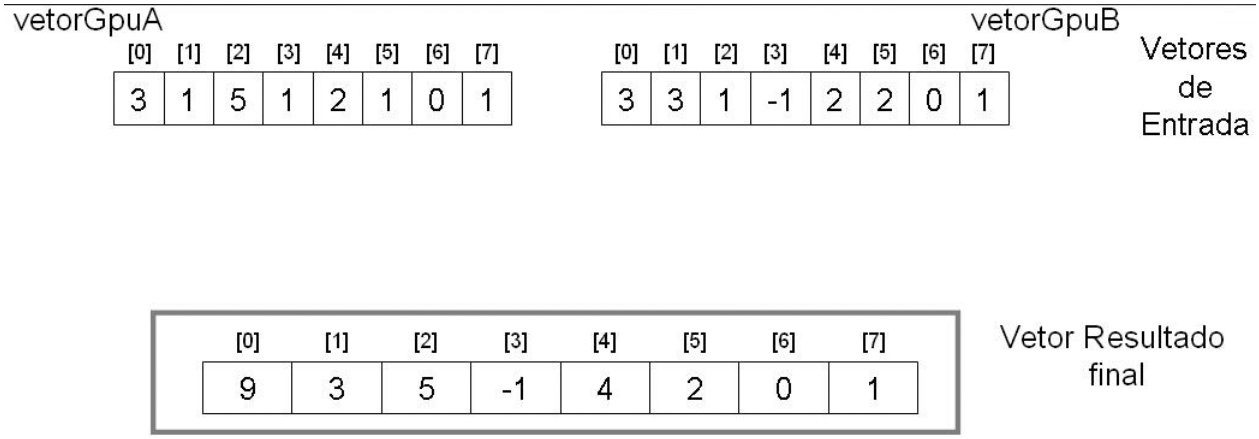


Figura 41: Vetor resultado ao final do processo de multiplicação dos vetores.

Para o 2º passo são realizados as seguintes operações:

2.a Após o cálculo do vetor resultado o programa realiza a chamada do *Kernel* **ProdutoInterno**, figura 23, que executa a soma em árvore dos dados do vetor resultado (figura 24).

2.b Na primeira iteração o vetor resultado conterá 8 posições. Seguindo o algoritmo da figura 24 o **for** da linha 5 realizará a divisão do vetor em duas partes iguais, figura 42.

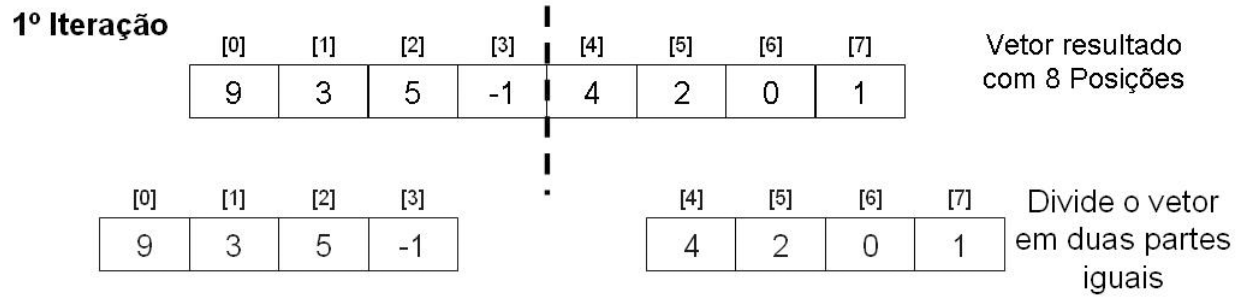


Figura 42: Divisão do vetor resultado na 1º iteração.

2.c Assim cada posição do vetor é representa por uma *thread*, dentro do **for** da linha 8 da figura 24 só entrarão as *threads* que apresentam a coordenada menor que a posição da divisão do vetor (figura 43).

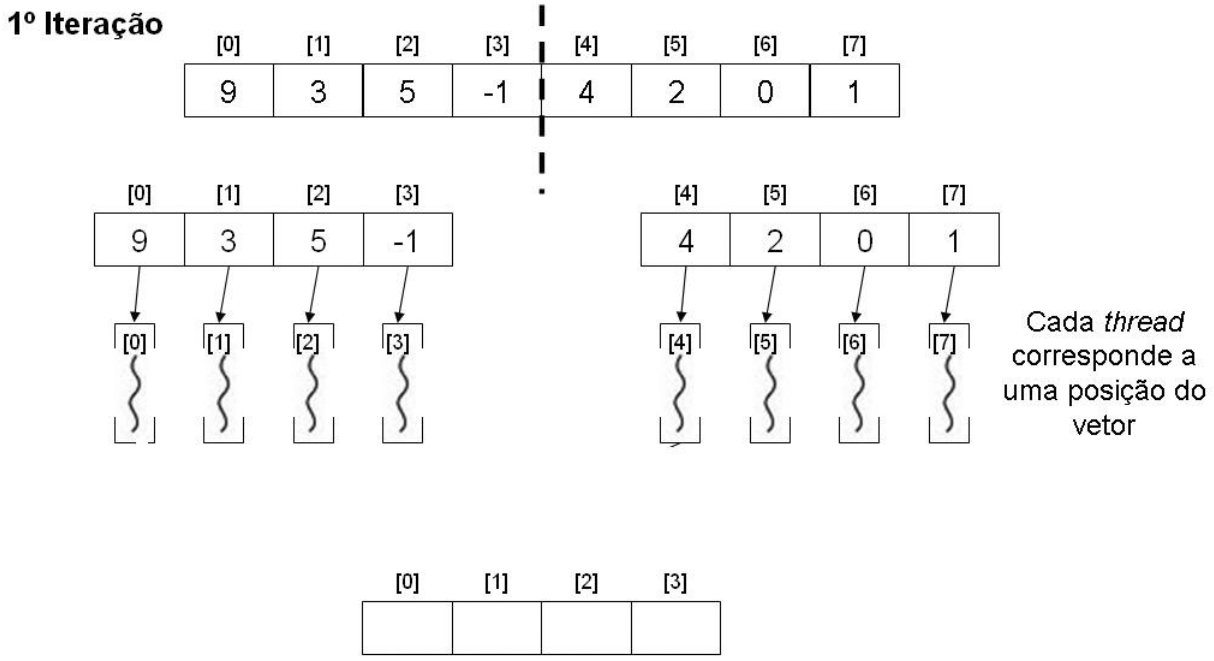


Figura 43: Cada *thread* representa uma posição do vetor.

2.d Na linha 10 da figura 24 é realizada a soma das posições correpondentes das partes do vetor. A ideia é somar as posições correpondentes entre as duas partes do vetor, figura 44 . A indexação da segunda metade é feita utilizando a posição da divisão somando a coordenada da *thread* reponsável pela posição. A variável **p** é usada para posicionar a indexação no bloco correto.

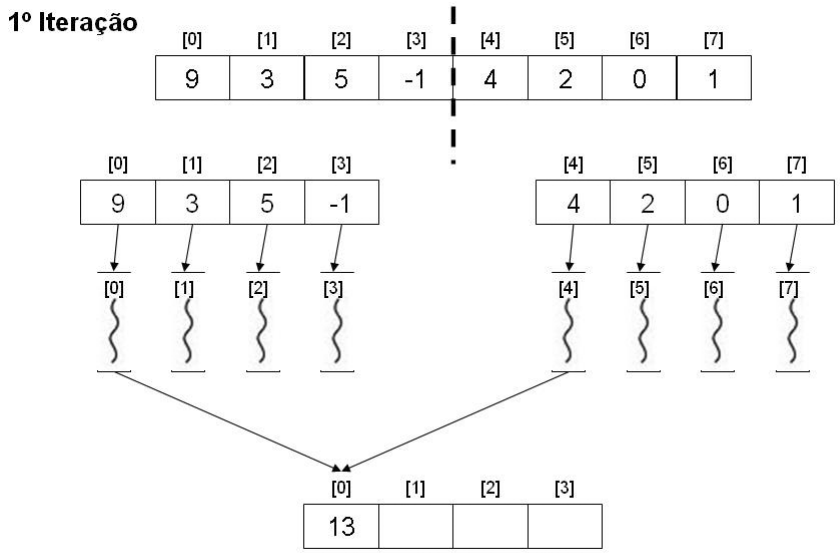


Figura 44: Soma das posições correspondentes das metades do vetor resultado.

2.e No final da 1º iteração a primeira metade do vetor resultado conterá os valores da segunda metade somados a suas posições correspondentes, figura 45.

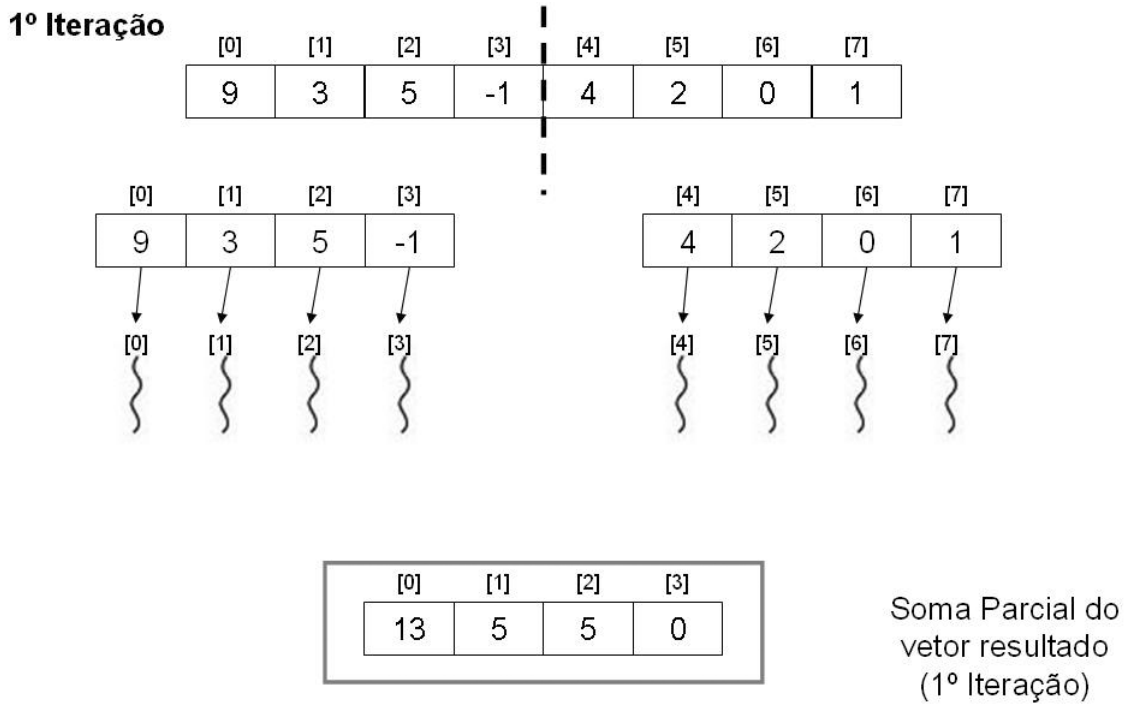


Figura 45: Final da 1º iteração.

2.f Para a 2º iteração o vetor obtido na iteração anterior será considerado o vetor principal. Sendo desta forma dividido em duas partes iguais, figura 46.

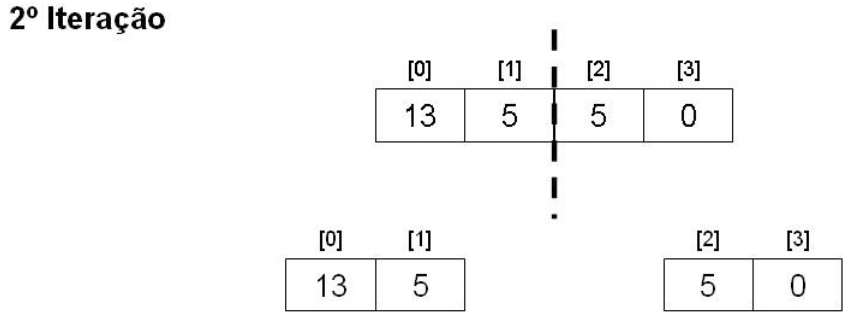


Figura 46: Início da 2º iteração com o vetor obtido da iteração anterior.

2.g O processo se repete, cada posição do vetor é representada por uma *thread*, figura 47.

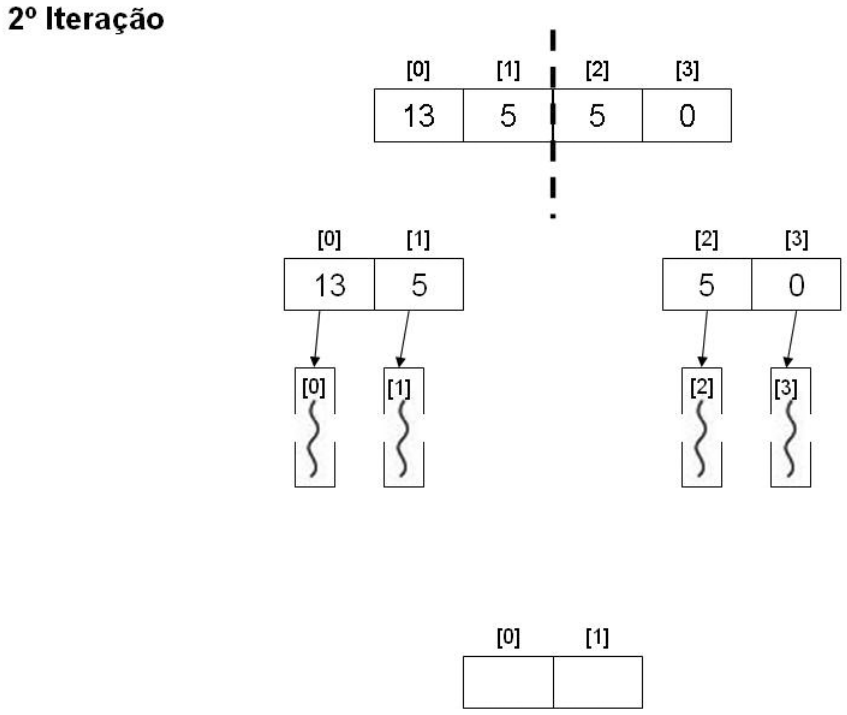


Figura 47: Posições dos vetores representadas por *threads*.

2.h Os dados da segunda metade do vetor serão somados nas posições correspondentes da primeira metade, figuras 48 e 49.

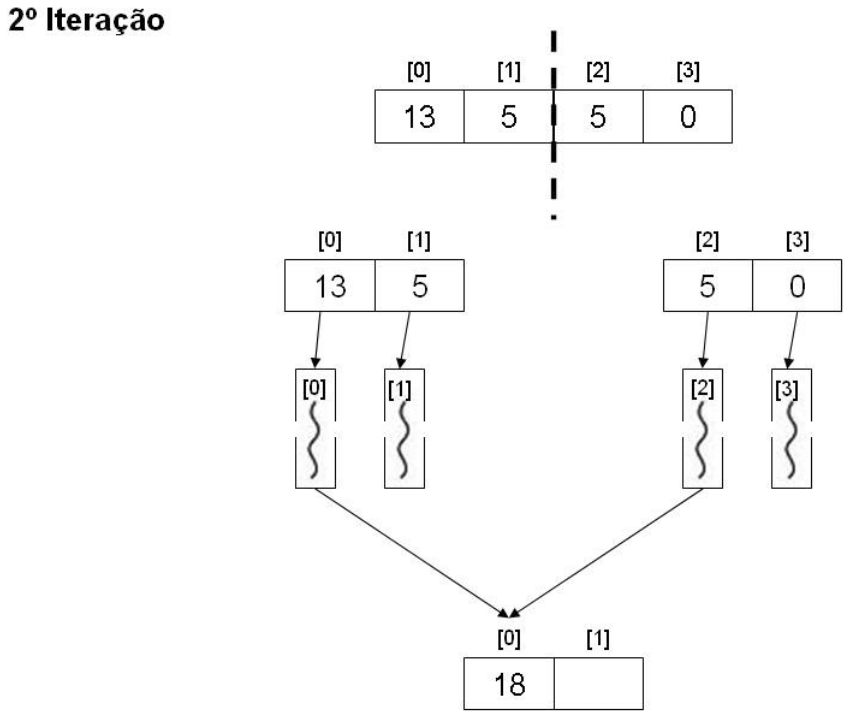


Figura 48: Adição do dado da posição 2 na posição 0 do vetor.

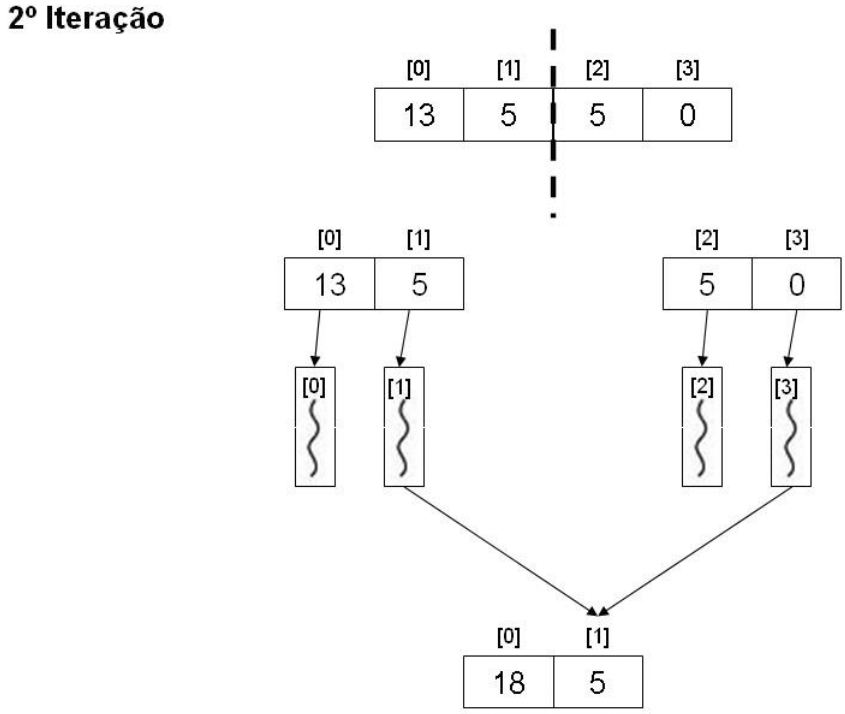


Figura 49: Adição do dado da posição 3 na posição 1 do vetor.

2.i No final da 2º iteração todos os dados da segunda metade do vetor estão somados nas posições correspondentes da primeira metade do vetor, figura 50.

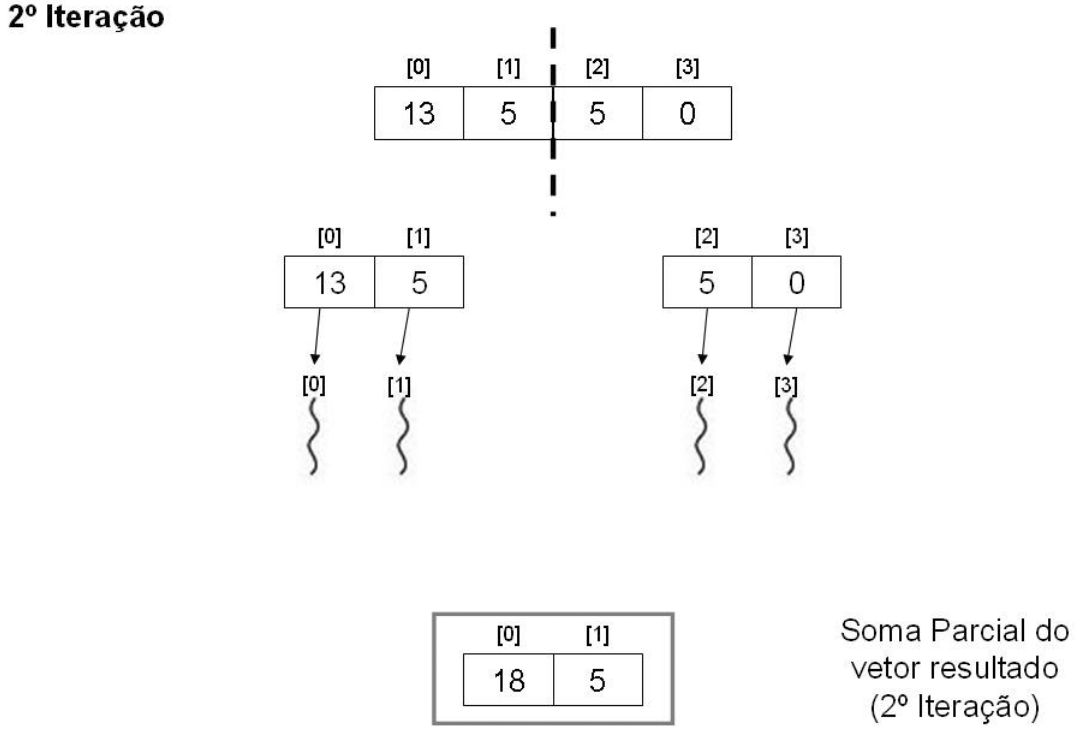


Figura 50: Resultado parcial no final da 2º iteração.

2.j Na 3º e última iteração o processo se repete nas figuras 51, 52 e 53.

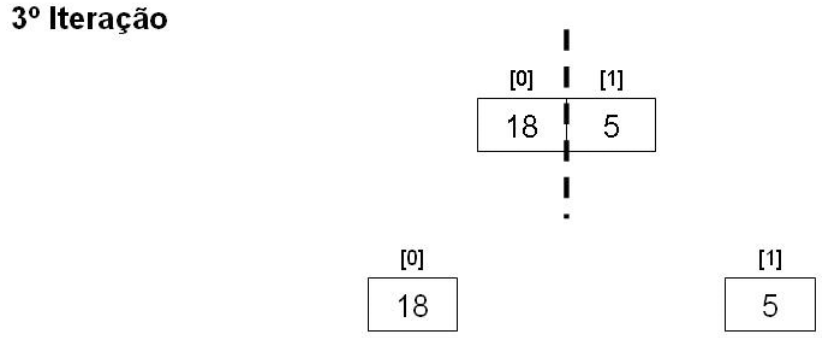


Figura 51: Divisão do vetor em duas partes iguais.

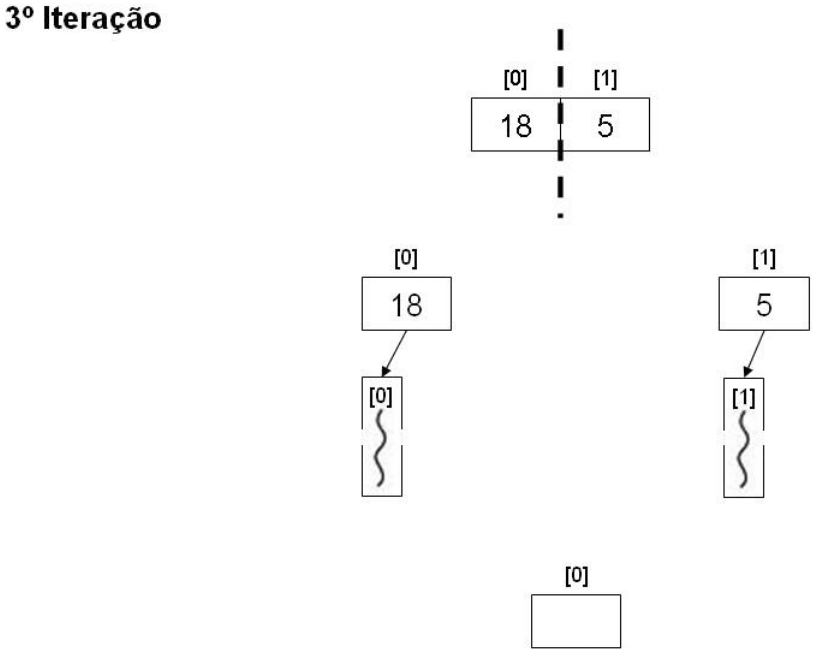


Figura 52: Cada posição do vetor é representada por uma *thread*.

3º Iteração

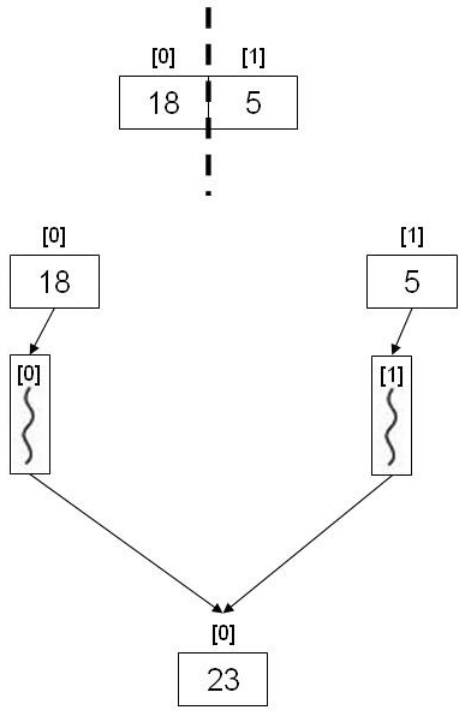


Figura 53: Os dados das posições 0 e 1 são somadas.

2.k No final da execução do *kernel* a *thread* 0 terá o resultado final da soma em árvore, isto é, terá o resultado do produto interno, figura 54. Lembrando que se existissem mais blocos este resultado seria acumulado em uma variável **soma**, até que todos os blocos da aplicação sejam executados.

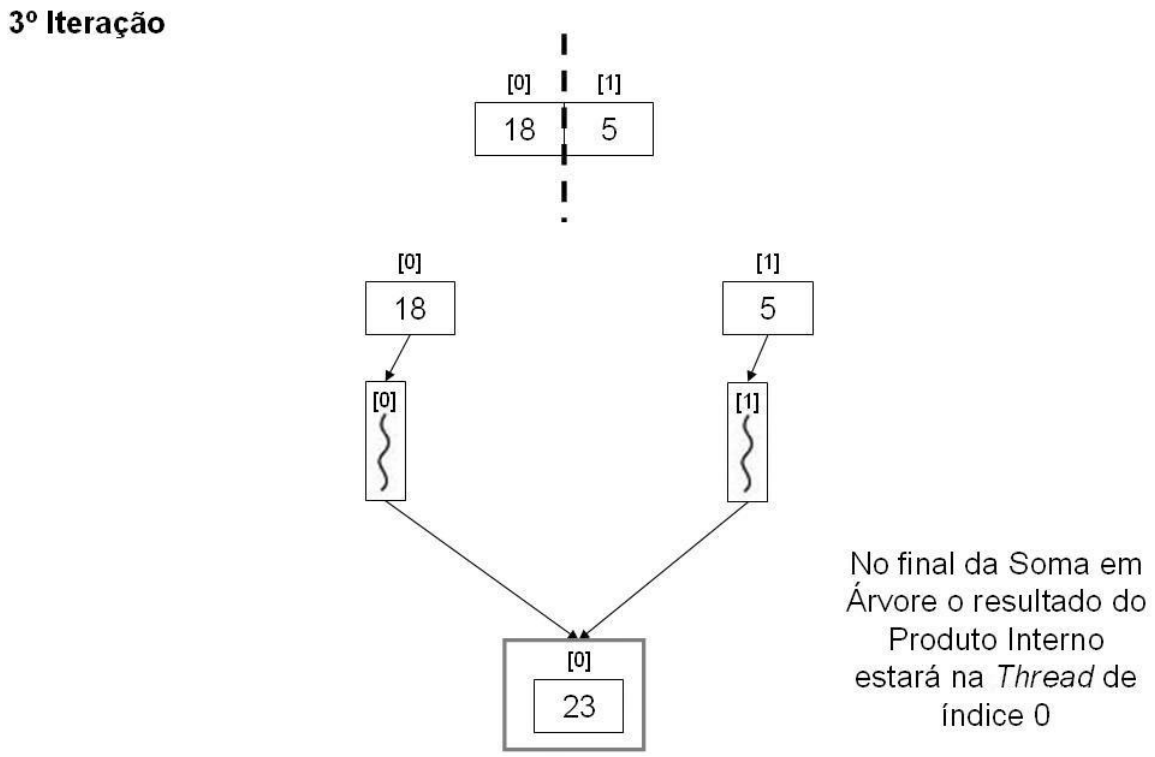


Figura 54: Resultado final da execução do *Kernel*. A *thread* 0 terá o resultado do Produto Interno.