UNIVERSIDADE FEDERAL DO ABC
Curso de Pós-Graduação em Ciência da Computação

Dissertação de Mestrado

Cleber Silva Ferreira da Luz

IMPLEMENTAÇÕES DE ALGORITMOS PARALELOS DA SUBSEQUÊNCIA MÁXIMA E DA SUBMATRIZ MÁXIMA EM GPU

Santo André - SP
2013

Curso de Pós-Graduação em Ciência da Computação

Dissertação de Mestrado

Cleber Silva Ferreira da Luz

IMPLEMENTAÇÕES DE ALGORITMOS PARALELOS DA SUBSEQUÊNCIA MÁXIMA E DA SUBMATRIZ MÁXIMA EM GPU

Trabalho apresentado como requisito parcial para obtenção do título de Mestre em Ciência da Computação, sob orientação do Professor Doutor Siang Wun Song e coorientação do Professor Doutor Raphael Yokoingawa de Camargo

Santo André - SP

2013

Este exemplar foi revisado e alterado em relação à versão original, de acordo com as observações levantadas pela banca no dia da defesa, sob responsabilidade única do autor e com a anuência de seu orientador.

Santo André, 17 de maio de 2013.

Assinatura do autor: _____

Assinatura do orientador: _____

Este trabalho contou com auxílio financeiro da Universidade Federadal do ABC - UFABC (bolsa de mestrado, institucional), de fevereiro/2011 a junho/2011, da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - CAPES (bolsa de mestrado, demanda social), de julho/2011 a janeiro/2013.

IMPLEMENTAÇÕES DE ALGORITMOS PARALELOS DA SUBSEQUÊNCIA MÁXIMA E DA SUBMATRIZ MÁXIMA EM GPU.

Cleber Silva Ferreira da Luz

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Sistemas de Computação

Banca Examinadora:

Presidente/Orientador: Prof. Dr. Siang Wun Song

Instituição: Universidade Federal do ABC - UFABC

Prof. Dr. Luiz Carlos da Silva Rozante

Instituição: Universidade Federal do ABC - UFABC

Prof. Dr. Alfredo Goldman vel Lejbman

Instituição: Universidade de São Paulo - USP

Prof. Dr. David Corrêa Martins Júnior (suplente)

Instituição: Universidade Federal do ABC - UFABC

Prof. Dr. Marco Dimas Gubitoso (suplente)

Instituição: Universidade de São Paulo - USP

Agradecimentos

Ao meu orientador, Prof. Dr. Siang Wun Song, agradeço o apoio e o incentivo.

Ao meu coorientador, Prof. Dr. Raphael Y. de Camargo, agradeço o apoio e o incentivo.

A todos os professores da UFABC, em especial aos professores Daniel Martins, Cláudio Meneses, André Balan e o Ronaldo Prati, agradeço pelo apoio e o incentivo dado a mim.

A minha família, em especial a minha querida mãe, Marlucia, pelas orações.

A todos os amigos que conheci nesses dois anos de estudos na UFABC.

A UFABC e a CAPES pelo suporte financeiro que possibilitou o desenvolvimento deste trabalho.

Resumo

Atualmente, GPUs com centenas ou milhares de processadores ou *cores* apresentam um poder computacional muito alto, em comparação com o de uma CPU. Pelo seu benefício-custo, há um grande interesse no seu uso para acelerar as computações de problemas que demandam alto poder computacional. Neste trabalho, são apresentadas implementações paralelas em GPU para dois problemas: subsequência máxima e submatriz máxima. Resultados experimentais obtidos são promissores. Para o primeiro problema, foi obtido um *speedup* de 146 (em relação ao tempo de execução usando apenas a CPU) no caso de entrada de 2.000.000 de elementos, para 4 GPUs. Para o segundo problema, foi obtido um *speedup* de 584, para uma matriz de entrada 15.360×15.360, com 2 GPUs. A nossa implementação se baseia num algoritmo paralelo para o problema da subsequência máxima que não se enquadra no modelo SIMD (*Single Instruction Multiple Data*), que é o modelo da quase totalidade dos algoritmos implementados em GPUs. No modelo SIMD, comandos condicionais podem causar degradação no desempenho, pois parte dos processadores podem executar o ramo *then*, enquanto que outra parte podem executar o ramo *else*. Mostramos neste trabalho que essa restrição é menos acentuada no caso de uma GPU com uma quantidade grande de *streaming multiprocessors* cada um com poucos processadores.

Palavras-chave: CUDA, GPU, subsequência máxima e submatriz máxima.

Abstract

Currently, GPUs with hundreds or thousands of cores present very high computing power, as compared to that of a CPU. Due to the cost-benefit, there is great interest in its use to accelerate the computation of problems that demand high computing power. In this work, parallel implementations on GPU are presented for two problems: maximum subsequence and maximum subarray. Experimental results obtained are promising. For the first problem, we obtained a speedup of 146 (as compared to the execution time of using only the CPU) for an input of 2,000,000 elements, for 4 GPUs. For the second problem, we obtained a speedup of 584, for an input matrix 15.360×15.360*, with 2 GPUs. Our implementation is based on a parallel algorithm for maximum subsequence that does not fit into the SIMD model (Single Instruction Multiple Data), that is the model of almost all the algorithms implemented on GPUs. In the SIMD model, conditional commands may cause performance degradation, since part of the processors may execute the then branck, while another part may execute the else branch. We show in this work that this restriction is less serious in case of a GPU with a large quantity of streaming multiprocessors, each with few processors.*

Keywords: CUDA, GPU, maximum subsequence and maximum subarray.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Os avanços na área de Biotecnologia, em especial no desenvolvimento de técnicas de sequenciamento de DNA (*Deoxyribonucleic Acid*), têm produzido uma gigantesca massa de dados biológicos, traduzidos em sequências de DNA e de proteínas. O grande desafio criado a partir da geração desses dados é a tarefa de analisá-los e transformá-los em informações biológicas relevantes, capazes de proporcionar aos pesquisadores novos conhecimentos e habilidades [18]. Essas habilidades incluem, por exemplo, novas técnicas para diagnóstico e tratamento de doenças genéticas.

A utilização da Ciência da Computação no tratamento dessas informações é imprescindível, não somente pela grande quantidade de dados gerados ou pelo tamanho das sequências, mas também pela possibilidade de se desenvolverem novas técnicas computacionais para resolver problemas de Biologia Molecular [18]. Essas novas técnicas deram origem a uma nova área de pesquisa, denominada Bioinformática.

Este trabalho de pesquisa apresenta duas implementações paralelas. A primeira implementação paralela lida com o problema da subsequência máxima. Essa implementação paralela pode ser utilizada para acelerar o processamento de várias aplicações da Biologia Computacional, tais como, análise de proteínas e sequências de DNA, identificação de genes, comparação de bases de DNA ou aminoácidos, entre outras aplicações. A segunda implementação paralela lida com o problema da submatriz máxima. Ambas visam o aproveitamento do poder computacional das placas gráficas GPUs (*Graphics Processing Units*) para obter soluções eficientes para os dois problemas.

1.1 Oportunidades oferecidas pelo uso de GPUs e CUDA

Atualmente, GPUs utilizam centenas ou milhares de processadores ou núcleos (*cores*) [31]. Impulsionadas pela crescente complexidade no processamento gráfico, em especial nas aplicações de jogos, as GPUs passaram por uma grande revitalização tecnológica. Em comparação com uma CPU, uma GPU contém um poder computacional muito alto [25]. A Figura 1.1 exibe uma comparação entre a capacidade de processamento das GPUs da NVIDIA e dos processadores da Intel, no que diz respeito às operações de ponto flutuante. Pode-se observar que as GPUs da NVIDIA já possuem um poder computacional maior em relação às CPUs da Intel [25].

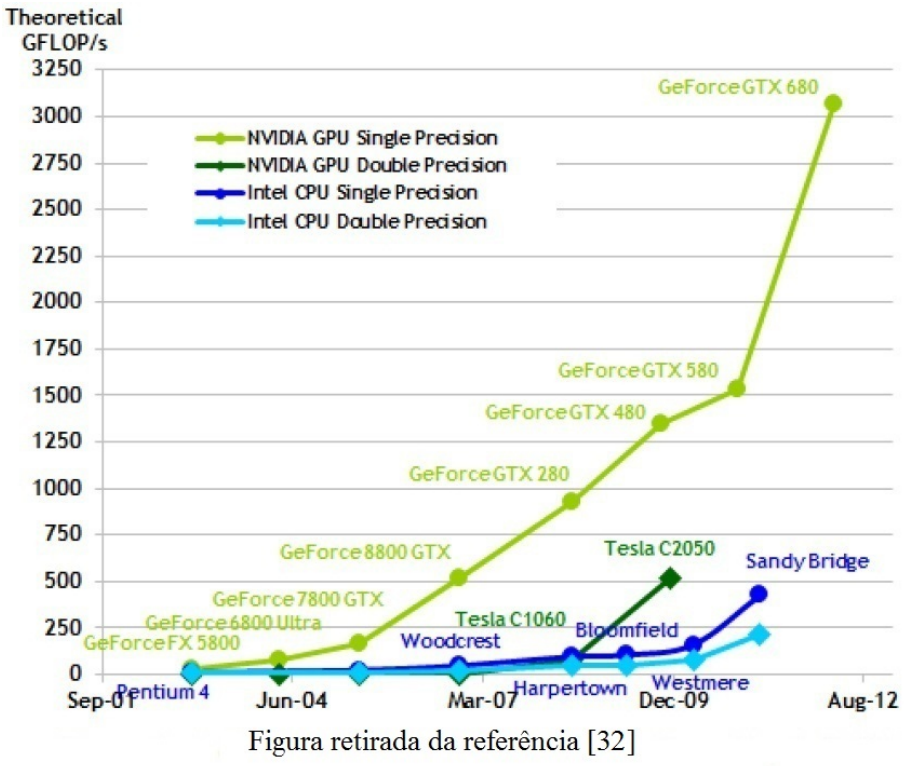


Figura 1.1: Evolução da velocidade em GFLOPS das GPUs e CPUs.

Tendo em vista esse enorme potencial computacional oferecido pelas GPUs da NVIDIA, essa empresa, em 15 de fevereiro de 2007, tornou público uma nova plataforma de *software* denominado CUDA (*Compute Unified Device Architecture*) [35]. Com CUDA é possível explorar o potencial computacional das GPUs, antes restrita à computação gráfica, para computação em geral, gerando o conceito de GPGPU (*General Purpose Graphics Processing Units*) [11].

1.2 Problema da subsequência máxima

O problema da subsequência máxima é definida como se segue. Dada uma sequência de n números $(x_1, x_2, ..., x_n)$, deseja-se determinar uma subsequência (contígua) cujos valores numéricos tenham soma máxima.

Por exemplo, dada a sequência $(3, 5, 10, -5, -30, 5, 7, 2, -3, 10, -7, 5)$, a subsequência máxima é $(5, 7, 2, -3, 10)$ e a soma máxima é 21.

O problema da subsequência máxima aparece em várias aplicações da Biologia Computacional [7]. Algumas delas são:

- Identificação de domínios transmembranas em proteínas;
- Análise de proteínas e sequências de DNA;
- Identificação de genes;
- Comparação de bases de DNA ou aminoácidos;
- Identificação de regiões hidrofóbicas.

Esse trabalho de pesquisa apresenta uma adaptação do algoritmo paralelo de Alves, Cáceres e Song [1] para implementação em GPU. Essa adaptação foi implementada em uma máquina composta por 2 placas gráficas GTX295 [19] da NVIDIA. Cada placa gráfica GTX295 possui 2 GPUs. No artigo de Alves, Cáceres e Song [1], publicado em 2004, utiliza-se o modelo CGM (*Coarse Grained Multicomputer*) para obter a subsequência máxima de uma sequência dada. Nesse artigo, o algoritmo paralelo de Alves, Cáceres e Song apresenta um *speedup* de 40, obtido através de um *cluster* composto por 62 nós.

O desafio aqui é que o algoritmo paralelo de Alves, Cáceres e Song [1] não se encaixa no paradigma SIMD (*Single Instruction Multiple Data*) [44], que é o modelo da quase totalidade dos algoritmos implementados em GPUs. Em uma GPU, comandos condicionais do tipo *if-then-else* podem causar degradação no desempenho, uma vez que parte dos processadores (núcleos) podem executar o ramo *then*, enquanto que outra parte pode executar o ramo *else*.

Os resultados experimentais obtidos neste trabalho, entretanto, foram promissores. Conforme será visto nas seções seguintes, com a implementação paralela em placas GPUs, foi possível obter um *speedup* da ordem de 140 (em relação ao tempo de execução somente

pela CPU), para sequências com 2.000.000 elementos. Esse *speedup* foi obtido através da utilização de 2 placas gráficas GPU GTX 295. Cada placa gráfica GPU GTX 295 contém 480 núcleos (*cores*), totalizando assim, 960 núcleos no processamento. Comparado com o resultado de Alves, Cáceres e Song [1], a implementação em GPUs é portanto superior em termos de benefício-custo. Essa é uma contribuição deste trabalho.

1.3 Problema da submatriz máxima

O problema de submatriz máxima é definido assim. Seja dada uma matriz K, de n linhas e m colunas. Deseja-se encontrar uma submatriz de K tal que a soma de seus elementos seja máxima [3].

O problema de submatriz máxima aparece em reconhecimento de padrão bidimensional, e corresponde a um estimador de semelhança de padrão em uma imagem digital [43].

Como exemplo, seja a matriz de entrada

$$K = \begin{pmatrix} -10 & -10 & 20 & -20 & -30 \\ 5 & -20 & 10 & 40 & 10 \\ 30 & -40 & 20 & -10 & -15 \\ -20 & 4 & -5 & 50 & 10 \\ 10 & -20 & 10 & -40 & 10 \end{pmatrix}.$$

A submatriz máxima de K é

$$\begin{pmatrix} 10 & 40 & 10 \\ 20 & -10 & -15 \\ -5 & 50 & 10 \end{pmatrix}$$

A soma dos elementos da submatriz máxima é 110.

Algoritmos paralelos foram apresentados por Perumulla e Deo [43] para o modelo PRAM, e por Alves, Cáceres e Song [3], para o modelo CGM, tendo este obtido um *speedup* de 23 para um *cluster* com 62 nós [3].

O presente trabalho apresenta uma adaptação do algoritmo paralelo de Perumulla e Deo [43] e de Alves, Cáceres e Song [3] para implementação em GPU. Experimentos realizados em uma máquina com duas placas GTX 680 [20] apresentaram um *speedup* de 584 (*speedup* obtido com 3.072 cores), considerando matriz quadrada K de n linhas, onde

$n = 15.360$.

Para conseguir este desempenho, foi necessário distribuir as computações de forma eficiente e independente entre as *threads*. Como será detalhado nas seções seguintes, cada *thread* fica responsável pelo processamento de uma submatriz de K, constituída das colunas $g, g+1, \ldots, h$ de K, com $0 \leq g \leq h \leq m-1$. Uma contribuição desse trabalho é a dedução de uma fórmula que associa um par de índices (g, h) a cada *thread*. Ver mais detalhes na Seção 5.3.

1.4 Organização da dissertação

Esta dissertação encontra-se organizada da seguinte maneira. No Capítulo 2 é apresentada uma descrição sobre GPU. O Capítulo 3 descreve o modelo de programação paralela (CUDA) utilizado em GPUs. O Capítulo 4 detalha o problema da subsequência máxima abordado nessa pesquisa, juntamente com as soluções paralelas e a implementação em GPU. O Capítulo 5 apresenta o algoritmo paralelo e a implementação em GPU para o problema da submatriz máxima. Já o Capítulo 6 apresenta os resultados experimentais obtidos nessa pesquisa. Finalmente conclusões são dadas no Capítulo 7.

Capítulo 2

GPU

GPU é uma unidade especializada em processar imagens. Impulsionadas pelo crescente mercado de processamento gráfico, as GPUs têm obtido grande visibilidade no mercado financeiro e na área acadêmica [32]. As GPUs têm conquistado cada vez mais espaço no mercado eletrônico, em especial em jogos. A necessidade de cálculos rápidos para renderizar imagens fez com que as GPUs ganhassem grande visibilidade no mercado de jogos. Sua visibilidade na área acadêmica aconteceu após o surgimento do conceito de GPGPU (*General Purpose Computing on Graphics Processing Units*) [11] também chamado de *GPU Computing*. Esse conceito permite o uso de GPUs para computação de propósito geral. A partir desse princípio, podemos explorar o grande poder computacional das GPUs para realizar computação de propósito geral. Com isso, é possível obter uma maior velocidade no processamento de dados.

Entretanto, nem todas aplicações têm um bom desempenho em GPUs. Uma aplicação ideal para GPU consegue obter um desempenho de até várias centenas de vezes mais rápida sobre a sua versão executada em uma CPU [32]. Tendo em vista esse enorme poder computacional oferecido pelas GPUs, desenvolvedores de *softwares*, cientistas e pesquisadores estão descobrindo usos amplamente variados para a computação com GPUs. Alguns deles são: identificação de placas ocultas em artérias [9, 32], análise do fluxo de tráfego aéreo [9, 32] e visualização de moléculas [9, 32].

Sendo composta por centenas de núcleos (*cores*), uma GPU consegue obter um desempenho significativo [28]. GPUs trabalham muito bem no processamento de imagens [32], cálculos aritméticos [32], simulações físicas [32], processamento de sinais [32], computação científica [32] e problemas que envolvem álgebra linear [32].

Um aspecto importante na análise do poder de processamento de uma GPU é a largura de banda da memória (*memory bandwidth*). Em termos de largura de banda, as GPUs

da NVIDIA também ultrapassaram as CPUs da Intel, conforme pode ser visto na Figura 2.1.

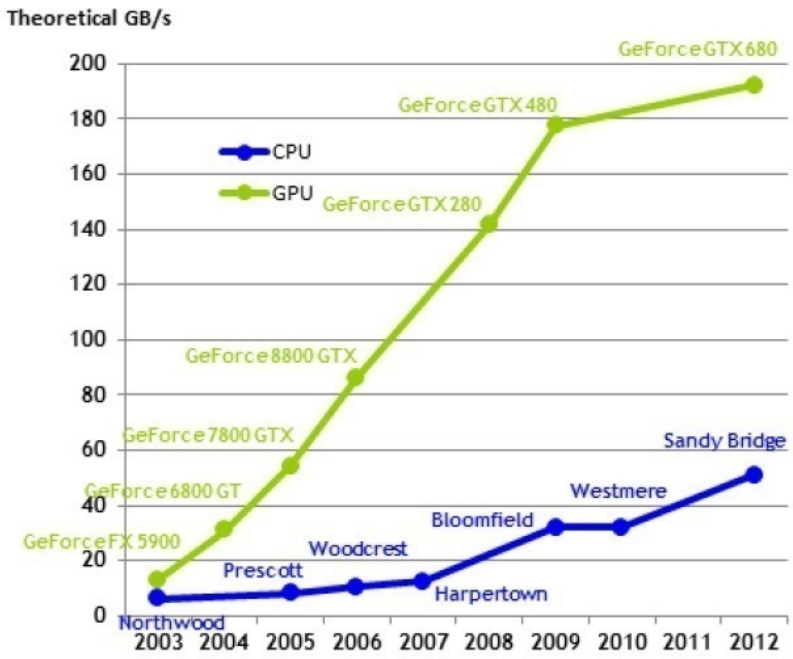


Figura retirada da referência [32].

Figura 2.1: Evolução da largura de banda das GPUs e CPUs.

Para aproveitar toda capacidade de processamento de uma GPU é preciso manter o *hardware* mais ocupado possível [24]. GPUs foram projetadas para o processamento de dados em grandes quantidades. Ao se processar uma quantidade pequena de dados em uma GPU, provavelmente não haverá um grande desempenho. Em alguns casos, quando a quantidade de dados for pequena, é aconselhável processar na própria CPU ao invés de enviar para processar na GPU. Agindo dessa formar provavelmente o processamento será mais rápido [30].

2.1 A capacidade de processamento de uma GPU

Em comparação com as CPUs, as GPUs possuem um grande poder computacional. Uma GPU consegue atingir valores de pico teóricos em processamento de ponto flutuante superiores a qualquer processador atualmente disponível [30]. Em comparação com um processador Xeon [15] da família Intel, analisando suas especificações, um processador Xeon da última geração consegue obter 48 GFLOPS (*Giga Floating Point Operations Per Second*) em seu processamento. Presume-se que este terá dificuldade em superar uma NVIDIA 9800 GTX em capacidades aritméticas que apresenta 648 GFLOPS como máximo teórico, mesmo com um número semelhante de transistores (820 milhões no primeiro vs

750 milhões no segundo) [15].

Os valores máximos teóricos apresentados no parágrafo anterior foram calculados considerando que cada núcleo do Xeon é capaz de executar quatro instruções de ponto flutuante em cada ciclo de relógio. Com quatro núcleos funcionando a 3 GHz obtemos um total de 48 GFLOPS [15]. Para a GPU os cálculos são mais complexos, na medida em que há potencial de cálculo nas diversas unidades, eventualmente diferentes, do processador.

A definição de uma métrica como o FLOPS (*Floating Point Operations Per Second*) apresenta algumas vantagens importantes. Naturalmente os valores de desempenho efetivo dependem da aplicação e podem ser muito inferiores aos valores de pico. Perante esta relação entre o potencial das GPUs e CPUs, passa a ser responsabilidade do programador otimizar as aplicações em ambos os contextos para utilizá-las devidamente.

2.2 Arquitetura de uma GPU

Placas gráficas GPUs fornecem um poder computacional muito elevado em comparação com uma CPU. Essa desigualdade se dá através da diferença entre as suas arquiteturas [10]. Uma CPU tem a sua arquitetura otimizada para a execução de programas que são processados por uma pequena quantidade de *threads*. Devido a esse fato, uma CPU é otimizada para executar várias instruções em paralelo de uma única *thread* e, com isso, boa parte de sua arquitetura é dedicada para o controle de instruções e para memória *cache* [31]. A arquitetura de uma GPU é otimizada para a execução de programas paralelos, programas que são processados por milhares de *threads* ao mesmo tempo. Com essa finalidade, sua arquitetura contém um número muito maior de processadores, e possui uma menor quantidade de controle e *cache* reduzido [31].

A Figura 2.2 ilustra a diferença entre as arquiteturas.

Apesar de suas arquiteturas diferenciadas, uma arquitetura completa a outra. Utilizando as duas arquiteturas em conjunto, podemos fazer uso da programação heterogênea. Com a programação heterogênea é possível processar aplicações que tenham partes sequenciais e partes paralelas. Partes sequenciais são processadas na própria CPU e as partes paralelas são enviadas para serem processadas na GPU.

É importante ressaltar que nesse cenário a CPU tem um papel fundamental, pois sem uma CPU fica impossível de utilizar uma GPU. Não é possível enviar dados para serem processados em uma GPU sem uma CPU. Uma GPU pode ser vista como um co-processador que ajuda a obter um maior desempenho no processamento dos dados [21].

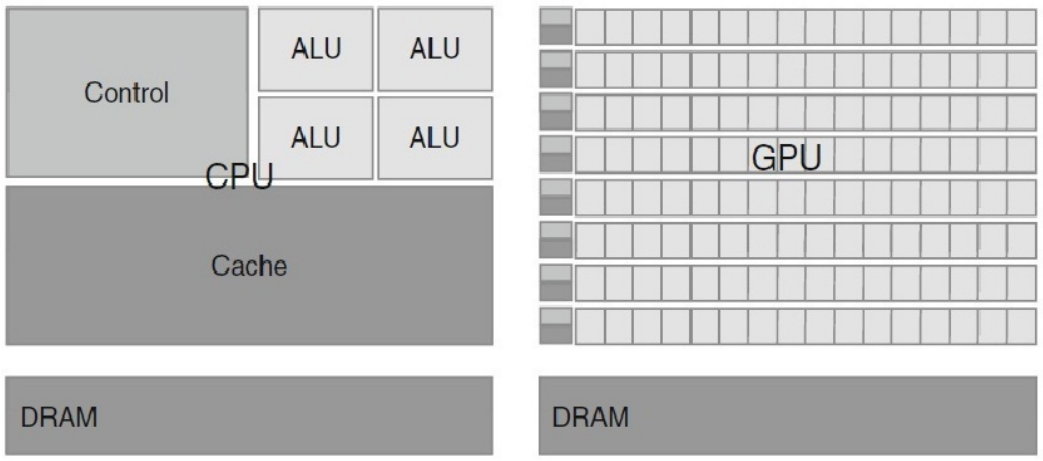


Figura retirada da referência [26].

Figura 2.2: Arquitetura de uma CPU e uma GPU.

Uma GPU é composta por centenas de unidades de processamento ou núcleos (*cores*), que são organizadas dentro de um vetor chamado *Streaming Multiprocessors* (SMs) [31]. A quantidade de *Streaming Multiprocessors* presentes em uma GPU varia de placa para placa. Um SM contém uma região de memória chamada de memória compartilhada (a ser explicada nas próximas seções). A finalidade dessa memória é realizar a troca de mensagens e informações entre as *threads* que são executadas dentro do SM. Além da memória compartilhada, um SM contém uma região de memória chamada registradores. Essa região de memória auxiliam as *threads* a executar as suas tarefas. Cada SM contém uma quantidade certa de *Streaming Processors* (SPs), os SPs são responsáveis por controlar as instruções compartilhadas e o *cache* de instrução. A quantidade de SPs presentes em um SM também varia de placa para placa. Uma placa gráfica G80, por exemplo, contém 16 SMs, com 8 SPs em cada SM, totalizando 128 SPs [16]. Na Figura 2.3 é possível observar como um SM é organizado.

GPUs processam seus dados utilizando o modelo de programação paralela SIMD (*Single Instruction Multiple Data*) [44]. No modelo SIMD uma única instrução é executada por diferentes processadores. Utilizando SIMD no processamento, as GPUs permitem que vários dados sejam processados de forma igual e em paralelo. Quando *n threads* são criadas para serem executadas na GPU, essa quantidade de *threads* é dividida em grupo de uma certa quantidade de *threads* denominado *warp* [31] (e.g. 32 *threads*. Cada grupo de *threads* de um *warp* processam o mesmo código e são executadas em um único SM. A quantidade de *threads* que compõem um *warp* pode variar de GPU para GPU.

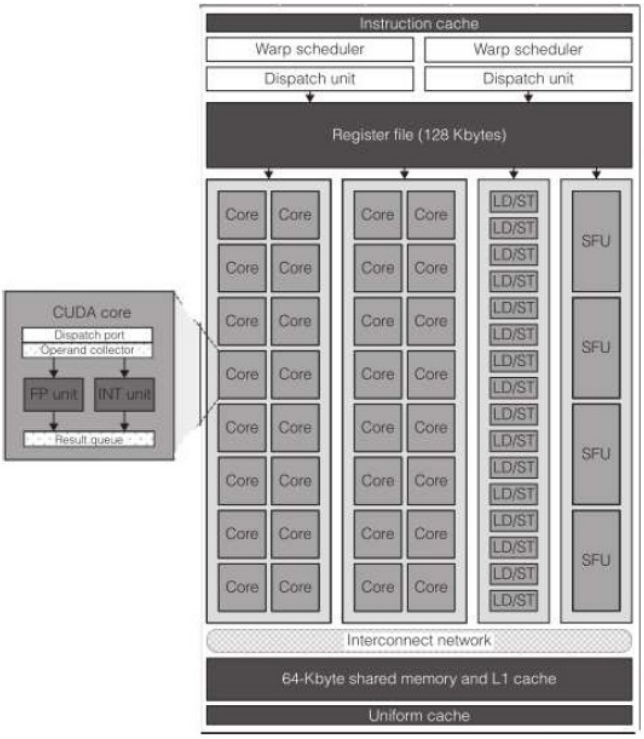


Figura retirada da referência [26].

Figura 2.3: Organização de um Streaming Multiprocessor.

2.3 Níveis de memórias de uma GPU

Uma GPU possui diferentes níveis de memória, cada nível com propósito, tamanho e acesso diferenciado. Quando se programa em GPU com o propósito de GPGPU, o programador deve conhecer e se preocupar com no mínimo 3 níveis de memória, que são:

- Memória global;
- Memória compartilhada;
- Registradores.

A Figura 2.4 mostra como essas memórias são acessadas. Além desses três níveis de memórias, uma GPU ainda possui memória de textura, memória local e a memória de constante.

2.3.1 Memória global

Quando os dados são enviados para serem processados em uma GPU, utilizamos a memória global para armazená-los. A memória global é relativamente grande, porém, seu

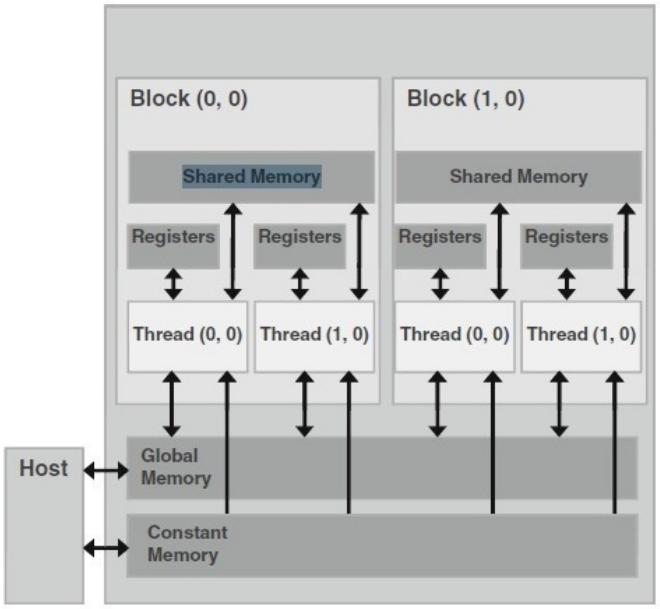


Figura retirada da referência [26]

Figura 2.4: Organização dos tipos de memórias mais importantes de uma GPU.

tamanho varia de placa para placa. Pelo fato de ser uma memória *off-chip*, seu acesso é lento [30]. Todas *threads* criadas dentro de uma GPU têm acesso de leitura e escrita na memória global.

Esse nível de memória tem o seu lado positivo e o seu lado negativo. O lado positivo é que esse nível de memória em relação ao tamanho é superior a todos os outros níveis. O lado negativo é que o seu acesso é muito custoso e lento. Quando o programador faz uso dessa região de memória ele dever estar bem ciente desses dois aspectos.

Para enviar dados para serem processados em uma GPU e consequentemente armazenados na memória global, é necessário utilizar a função *cudaMalloc()* [31] e a função *cudaMemcpy()*[31] (ambas são funções da linguagem CUDA). Mais detalhes dessas duas funções encontram-se na Seção 3.2.

2.3.2 Memória compartilhada

Cada multiprocessador (SM) de uma GPU possui uma quantidade pequena de memória chamada de memória compartilhada. Essa memória é utilizada para as *threads* de um mesmo bloco trocarem informações [30]. Cada multiprocessador tem a sua própria memória compartilhada. Nessa região de memória as *threads* têm acesso de leitura e escrita.

Semelhante à memória global, a memória compartilhada também tem um lado positivo e um negativo. Seu lado positivo é que ela é uma memória *on-chip* e, com isso, o seu acesso é rápido. Seu lado negativo é que o seu tamanho é relativamente pequeno. Seu tamanho varia de placa para placa. Para utilizar esse nível de memória é necessário utilizar uma palavra reservada chamada *Shared* [31].

2.3.3 Registradores

Registradores são espaços de memórias reservadas para auxiliarem as *threads* na execução de suas tarefas. Cada *thread* possui uma certa quantidade de registradores, e seu acesso é privado no sentido de uma *thread* não poder acessar os registradores de outra *thread*. No geral, toda vez que encontramos declarações como *int x* em um código CUDA, estamos utilizando registradores [30].

2.4 Arquitetura Fermi

A arquitetura Fermi é uma das mais importantes arquitetura das GPUs da NVIDIA. Ela possui 16 multiprocessadores, cada um com 32 núcleos, totalizando 512 núcleos por chip [34]. Seu gerenciador de *threads GigaThread* distribui o trabalho para os multiprocessadores de acordo com a carga dinâmica através de agrupamentos de *threads*, podendo executar mais de um programa simultaneamente se apropriado. O gerenciador interno do multiprocessador gerencia *threads* individuais, podendo executar até 1536 *threads* concorrentes.

Essa arquitetura possui espaço de endereçamento físico de 40 bits e virtual de 64 bits e uma *cache* L2 unificada entre os multiprocessadores de 768 kbytes, que está conectada a seis interfaces de DRAM e a interface PCIe. Os multiprocessadores possuem *cache* L1 de dados e memória local configuráveis, totalizando 64 Kbytes, podendo ser configuradas para uma de 16 Kbytes e a outra 48 Kbytes ou vice-versa, banco de registradores de 128 kbytes, *cache* de instrução, *cache* e memória de texturas e 2 grupos gerenciadores de *thread* e unidades de despacho de instruções [34]. Além disso, uma GPU possui também um protetor de memória para aumentar a integridade dos dados, podendo corrigir erros de 1 bit e detectar erros de 2 bits na memória, *caches* ou registradores [41]. Isso prolonga muito o tempo de vida dos sistemas, o que motiva ainda mais sua utilização para servidores.

Essa arquitetura é composta por 32 núcleos, que são capazes de executar até 32 instruções aritméticas por ciclo de relógio. Cada núcleo possui uma unidade inteira e uma de ponto flutuante [37]. Sua unidade de ponto flutuante inclui, além dos tipos das operações

básicas, instrução FMA *(Fused Multiply-Add)* que possuem o formato $D = A \times B + C$ sem perda de precisão. Sua velocidade em operações de 64 bits de ponto flutuante é metade da velocidade em comparação com as de 32 bits [39]. A Figura 2.5 mostra a organização da arquitetura Fermi.

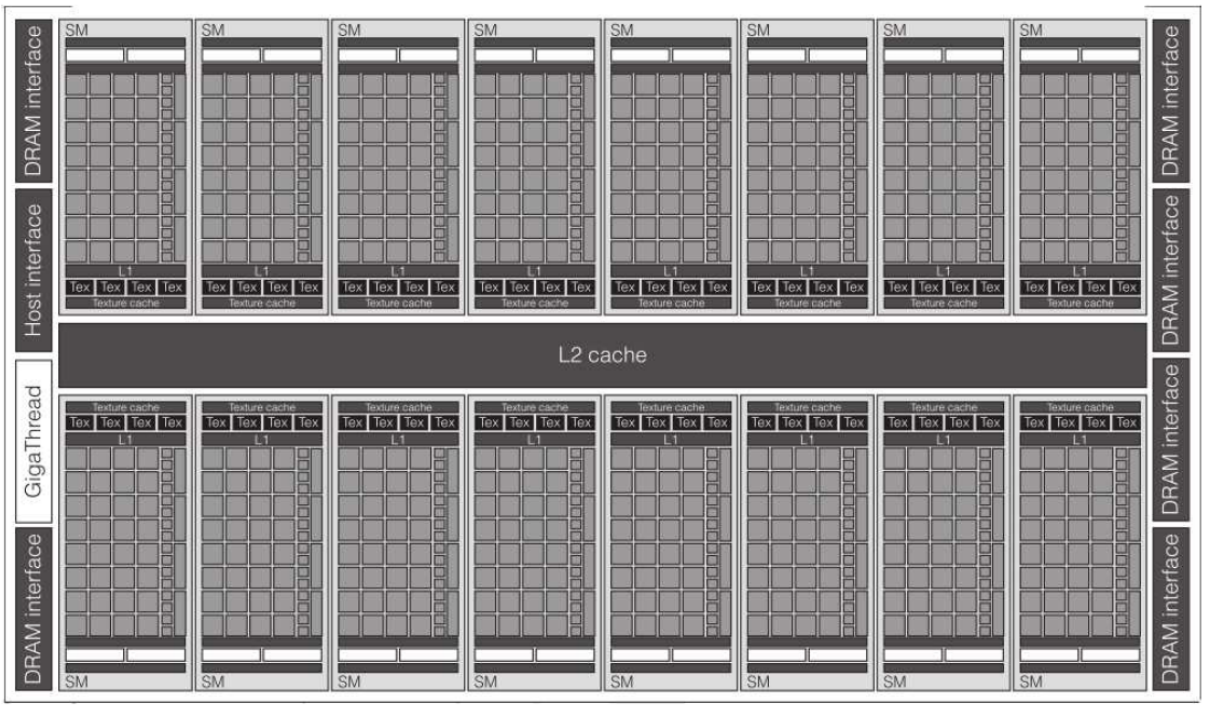


Figura retirada da referência [36]

Figura 2.5: Arquitetura Fermi.

2.5 Arquitetura Kepler

A arquitetura *Kepler*, sucessora da arquitetura *Fermi*, é a ultima arquitetura projetada pela NVIDIA. Seu foco, ao ser projetada, foi conseguir o maior desempenho por *watt* [33]. As primeiras placas a utilizar essa tecnologia são a GeForce GTX 680, para *desktops* e a GeForce GT 640M, para *notebooks*.

A *Kepler* é uma *Fermi* remodelada, porém com algumas diferenças fundamentais. Cada SM contém agora 192 CUDA *cores*, para um total de 1536 *cores* em toda a GPU *Kepler* [36]. Essas mudanças fizeram com que o desempenho aumentasse consideravelmente, se comparado ao da *Fermi*. Com isso, a arquitetura *Kepler* usa menos energia ao mesmo tempo em que proporciona mais desempenho. Além disso, foram feitas alterações no sistema de memória e na estrutura de processamento. Um *cache* L2 de 512 KB é compartilhado por toda GPU, proporcionando espaço de *buffer* para várias unidades do *chip*,

a sua frequência da largura de banda de *cache* e a operação atômica foram aumentadas para fornecer um maior suporte aos CUDA *cores*. A arquitetura *Kepler* conta com quatro controladores de memória de 64 bits, operando a uma taxa de 6008 MHZ [38], quase o dobro de sua antecessora, que contava com seis controladores de 64 bits.

Foi implementado na *Kepler* um novo motor de *display*, capaz de mostrar até 4 monitores ao mesmo tempo com apenas uma GPU, inclusive com a opção de 3D em todos os monitores. A codificação de vídeo deixou de ser feita pelos CUDA *cores*, e passou a ser feito por um *software* implementado direto no *hardware*, chamado NVENC [36]. Tal mudança fez com que a GPU consuma menos energia e deixasse a codificação quatro vezes mais rápida. Todas as mudanças resultaram no que é a GPU mais rápida e com menos consumo de energia já vista. Porém isso tudo não veio sem um sacrifício. As GPUs *Fermi* tem uma habilidade de fazer cálculos muito superiores se comparadas a *Kepler* [14], pois a NVIDIA tirou algumas dessas habilidades para melhorar a eficiência por energia.

A economia de energia é algo que realmente impressiona na *Kepler*. Comparada à antecessora, a energia ativa foi reduzida em 15%, e o desperdício em 50%, o que resultou em uma melhora global de 35% [22]. Isso não é de se surpreender, pois a maioria das mudanças tinha esse propósito. O consumo de energia de processadores *stream* e controles lógicos foram reduzidos, agora ao contrário da *Fermi*, as GPUs *Kepler* têm *clock* na mesma frequência. Os geradores de *clock* e os processadores de *stream* requerem menos energia à frequência mais baixa. Com isso, o consumo total de energia é reduzido, embora o desempenho máximo teórico permaneça o mesmo.

Outra modificação que contribui com a eficiência é o novo *design* do SM (Streaming Multiprocessor) que recebeu o nome de "SMX"[36] e foi especialmente redesenhado para obter a maior eficiência em desempenho por *watt*. Para melhorar a eficiência, o SMX roda com *clock* gráfico ao invés de utilizar o *shader core clock*, mas com 1536 CUDA *core*s em GK104, a GeForce GTX 680 SMX oferece duas vezes mais desempenho por *watt* que sua antecessora, a Fermi SM (GF110) [38]. Isso permite que a nova GTX 680 [20] proporcione um desempenho revolucionário por *watt* em comparação a GTX 580.

Capítulo 3

CUDA

CUDA é uma plataforma de *software* para computação paralela de propósito geral criada pela NVIDIA em 2007. CUDA pode ser aplicado em diversos campos tais como matemática, computações científicas, biomedicina e engenharia [32]. Com essa tecnologia inovadora é possível utilizar os recursos das unidades de processamento gráfico (GPUs) da NVIDIA. Utilizando tais recursos é possível acelerar o processamento das aplicações de forma a permitir a resolução de muitos problemas computacionais complexos em um tempo menor que utilizando somente uma CPU.

Programadores de C e C++ conseguem programar facilmente em CUDA. Sua sintaxe é semelhante à da linguagem C. CUDA é disponibilizado gratuitamente para as plataformas Windows, Linux e Mac e possui versão estável e bem documentada.

3.1 Compilação de um código CUDA

Uma aplicação CUDA consiste de uma ou mais fases que são executadas na CPU ou na GPU. As fases que apresentam partes sequenciais são processadas na CPU, e as fases que apresentam partes paralelizáveis são processadas na GPU. Uma aplicação CUDA contém um código unificado que permite executar instruções na CPU e na GPU ao mesmo tempo. CUDA tem o seu próprio compilador. Esse compilador é chamado de NVCC [30]. Dentro do NVCC existe um compilador padrão para a linguagem C e outro para a linguagem CUDA. O NVCC separa a compilação do código, para que partes do código escritas em C sejam compiladas utilizando o compilador da linguagem C, e as partes do código escritas em CUDA sejam compiladas com o NVCC. Nesse trabalho de pesquisa o código do programa CUDA foi compilado com o NVCC e o código do programa sequencial foi compilado com o GCC.

Podemos abstrair esse procedimento a partir da Figura 3.1.

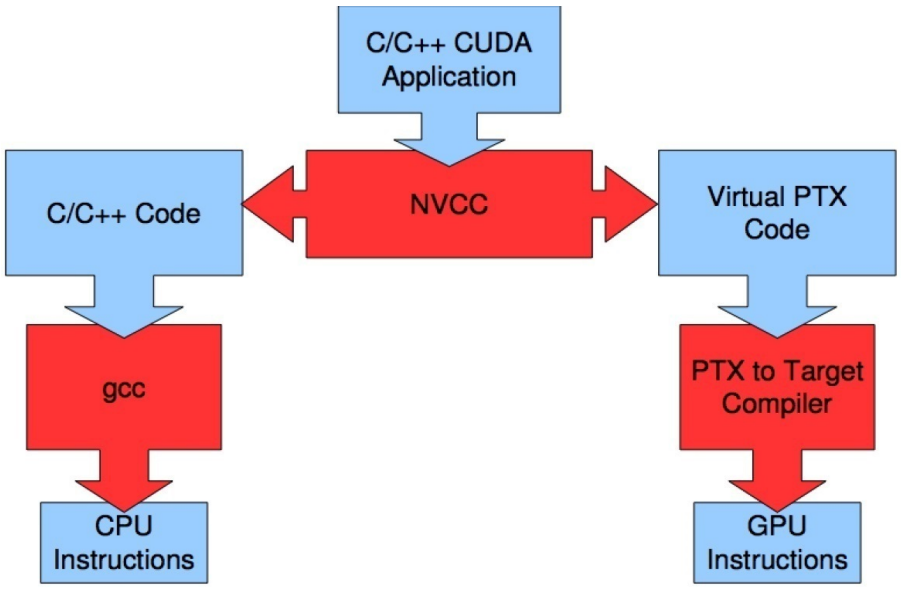


Figura retirada da referência [32].

Figura 3.1: Compilação de um código CUDA.

3.2 Execução de um código CUDA

O processamento de dados realizado em uma GPU é feito através de funções paralelas. Essas funções paralelas recebem o nome de *kernel* [29]. Quando um *kernel* é lançado para ser processado dentro de uma GPU centenas de *threads* são criadas. Em outras palavras, um *kernel* controla um conjunto de *threads* que são executadas dentro da GPU. As *threads* são organizadas dentro de blocos. Esses blocos são organizados dentro de um *grid* [29]. Mais detalhes sobre blocos e *grid* encontram-se na Seção 3.4.

A definição de um *kernel* é semelhante à declaração de uma função em C, a única diferença é a necessidade do uso de uma palavra reservada chamada *global*. Quando um *kernel* é chamado para ser executado, é necessário especificar a quantidade de *threads* e de blocos que serão executados. A Figura 3.2 exibe uma execução de uma aplicação em CUDA.

Quando um *kernel* é lançado o processamento é movido para a GPU e, nesse momento as *threads* são criadas e entram em execução. Quando todo o processamento termina, o *kernel* é finalizado e o processamento volta para a CPU. Em uma aplicação CUDA o processamento dos dados é realizado fazendo alternâncias entre CPU e GPU [30].

Abaixo seguem dois exemplos de códigos. O código 1 mostra como um *kernel* é decla-

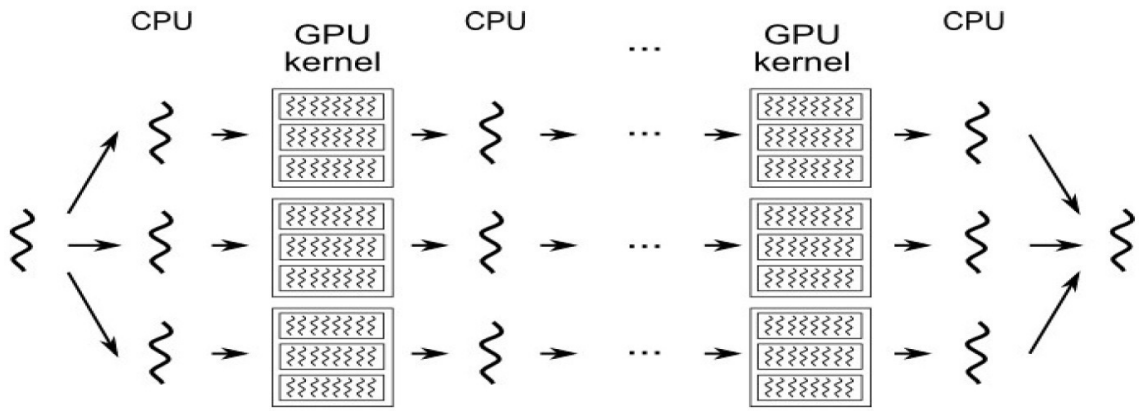


Figura retirada da referência [32].

Figura 3.2: Execução de código em CUDA.

rado. O código 2 ilustra como um *kernel* é chamado.

Código 1: *Kernel*

```
//funcao kernel
__global__ void nomeDoKernel (lista_de_parametros)
{
  ...
}
```

Código 2: Chamada de um *kernel*

```
nomeDoKernel<<<gridDim, blockDim>>>(lista_de_parametros);
```

Figura retirada da referência [25].

Figura 3.3: Exemplo de declaração e chamada de um *kernel*.

3.3 Estrutura de uma aplicação CUDA

Como descrito na Seção 3.2 o processamento de uma aplicação CUDA inicia na CPU e quando um *kernel* é lançado o processamento passa para a GPU. Após a finalização do *kernel* o processamento volta para a CPU. Para enviar dados para serem processados em uma GPU é necessária a realização de alguns passos, tais como:

1. Reservar espaço de memória na GPU;
2. Copiar os dados para a GPU;

3. Processar os dados na GPU;

4. Copiar os dados de volta para a CPU;

5. Liberar o espaço de memória reservado na GPU.

A reserva de memória em uma GPU é realizada através da função *cudaMalloc()* [45]. Sua sintaxe é semelhante a função *malloc()* da linguagem C. Após feita a reserva de memória é necessário realizar a cópia de dados para serem processados na GPU. Essa cópia é realizada através da função *cudaMemcpy()* [45]. Sua sintaxe é semelhante a função *memcpy()* da linguagem C.

A etapa de processamento dos dados é realizada com a chamada do *kernel*, após a finalização do *kernel* é necessário copiar os dados de volta para a CPU. A última etapa consiste em liberar o espaço de memória reservado na GPU. Essa liberação é realizada através da função *cudaFree()* [45] que é equivalente a função *free()* da linguagem C [30, 45]. A Figura 3.4 exibe um exemplo da organização dessas etapas.

A reserva de memória e a cópia dos dados para a GPU são realizadas na própria CPU [23]. A única responsabilidade da GPU é o processamento dos dados.

```
2  main(){
3
4  //Processamento na CPU
5          .
6          .
7          .
8
9  //Reservando espaço de memoria na GPU
10 cudaMalloc((void**)&vet_d, N * sizeof(int));
11
12 //Copiando os dados para a GPU
13 cudaMemcpy(vet_d, vet_h, N * sizeof(int), cudaMemcpyHostToDevice);
14
15 //Lançado o kernel para ser processado na GPU
16 subSeqMax<<<BLOCK_SIZE, nThreadsPerBlock>>>(vet_d, N);
17
18 //Copiando os dados de volta para a CPU
19 cudaMemcpy(vetFinal_h, vetFinal_d, N * sizeof(int), cudaMemcpyDeviceToHost);
20
21 //Processamento contínua na CPU
22         .
23         .
24         .
25
26 cudaFree(vetFinal_d);
27
28 return 0;
29 }
```

Figura 3.4: Exemplo de um código em CUDA.

3.4 Grid, blocos e threads

Quando lançamos um *kernel* para ser processado em uma GPU, uma certa quantidade de *threads* são criadas. Essas *threads* são divididas e organizadas dentro de blocos. Os blocos, por sua vez, são organizados dentro de um *grid*. Através dessa organização as *threads* conseguem distribuir dados e trabalhos. A execução de um *grid* consiste em processar diversas *threads* agrupadas em blocos, e um *grid* pode possuir uma, duas ou três dimensões [25].

Um bloco é a forma com que as *threads* estão organizadas para serem executadas dentro de um SM. As *threads* dentro de um bloco conseguem trocar dados e conseguem se sincronizar em um determinado momento. Essa sincronização é realizada através do *syncthreads()*, mecanismo de sincronização utilizado na linguagem CUDA. Quando um *kernel* chama o *syncthreads()*, é formada uma barreira de sincronização. Com essa barreira as *threads* não continuam o processamento até que todas elas tenham chegado à barreira. Depois que todas *threads* tenham chegado à barreira elas continuam o processamento.

Semelhante ao *grid*, os blocos também possuem uma, duas ou três dimensões. Cada bloco possui um identificador único (*blockId*) que permite descobrir qual é a sua posição dentro do *grid*. É possível lançar 65.535 blocos para serem processados em uma GPU e cada bloco suporta no máximo 512 *threads* (pela arquitetura *FERMI*). Quando se enviam muitos blocos para serem processados em uma GPU eles são processados em ordem pelos *Streaming Multiprocessors*.

Cada *Streaming Multiprocessor* tem a capacidade de processar pelo menos um bloco de cada vez. Dependendo da quantidade da recursos que os blocos necessitam, o SM consegue executar mais de um bloco simultaneamente [29]. A quantidade de blocos lançados são divididos entre os SMs. Cada SM processa uma fração da quantidade de blocos lançados [30]. A Figura 3.5 mostra um exemplo de processamento de blocos.

As *threads* também possuem um identificador único (*threadId*) que permite descobrir qual a sua posição dentro de um bloco [29]. É possível descobrir a localização das *threads* em uma dimensão usando *threadIdx.x*, em duas dimensões usando o *threadIdx.y* e na terceira dimensão usando *threadIdx.z*. As *threads* são criadas no momento em que o *kernel* é lançado, e são destruídas quando o *kernel* é finalizado.

As *threads* conseguem trabalhar em conjunto, uma ajudando a outra, para que todas juntas consigam realizar o processamento dos dados. Entretanto as *threads* só conseguem se comunicar e trabalhar em conjunto com as *threads* do mesmo bloco. Pelo fato de a GPU utilizar o modelo SIMD em seu processamento, quando uma *thread* solicita um recurso para o SM, podemos entender que todas as *threads* irão realizar a mesma solicitação. A

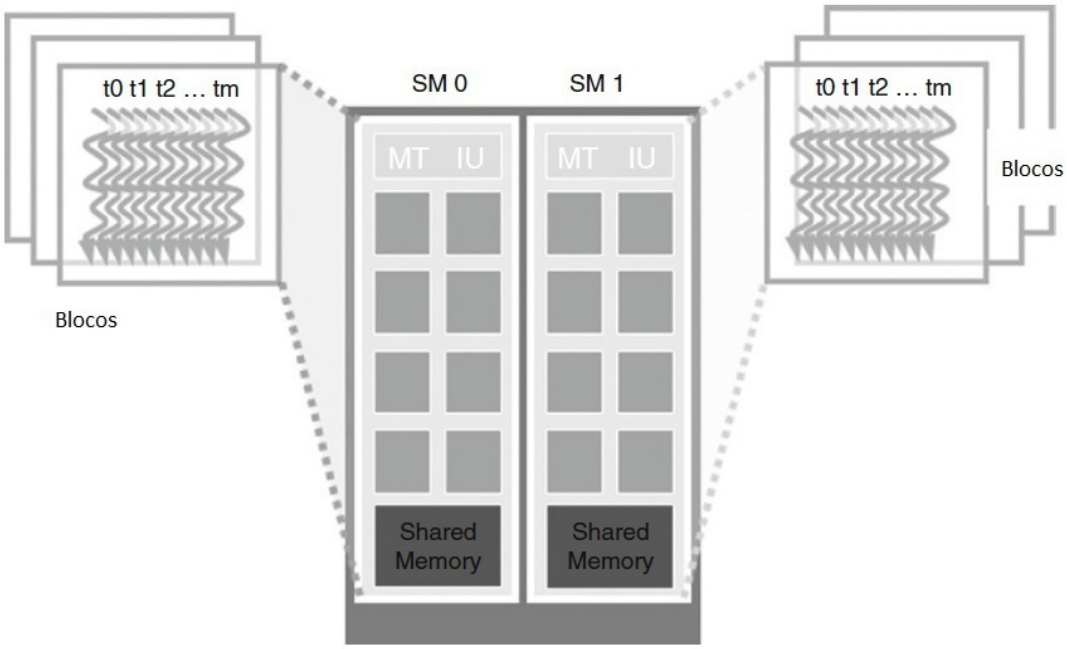


Figura retirada da referência [26].

Figura 3.5: *Streaming Multiprocessors* processando blocos.

Figura 3.6 mostra a organização de um *grid*.

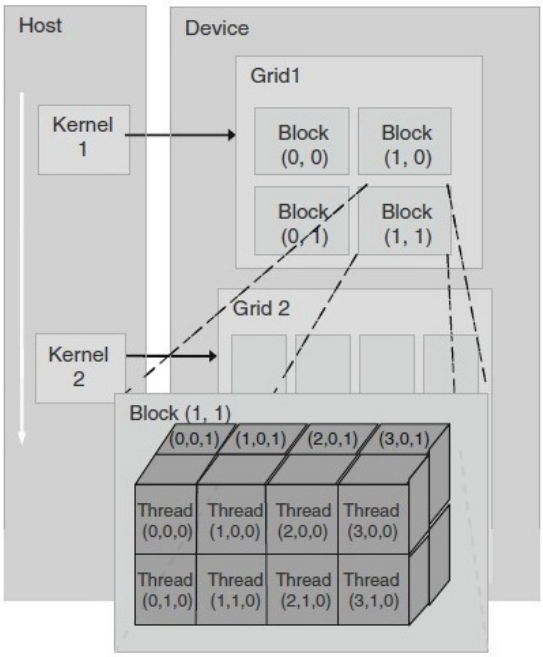


Figura retirada da referência [26].

Figura 3.6: Organização de um *grid*.

Capítulo 4

Subsequência máxima

O problema da subsequência máxima é definido como se segue. Dada uma sequência de n números $(x_1, x_2, \ldots, x_n)$, deseja-se encontrar uma subsequência contígua que apresente uma soma máxima [2]. Uma subsequência contígua é dada por $(x_i, \ldots, x_j)$ com $1 \leq i \leq j \leq n$. A subsequência máxima será apresentada por $M = (x_i, \ldots, x_j)$, para um dado par de índices i e j. A soma desse trecho será apresentado por T_M. Supomos que pelo menos um dos x_i é positivo. Desse modo, o T_M sempre terá uma pontuação positiva.

Obviamente se todos os números da sequência original são positivos, a subsequência máxima é toda a sequência original. No problema da subsequência máxima é permitido que haja números negativos. Por exemplo, dada a sequência

$$(3, 5, 10, -5, -30, 5, 7, 2, -3, 10, -7, 5),$$

a subsequência máxima é $M = (5, 7, 2, -3, 10)$ e a soma máxima é $T_M = 21$.

O problema da subsequência máxima aparece em várias aplicações da Biologia Computacional [7], tais como: identificação de domínios transmembranas em proteínas, análise de proteínas e sequências de DNA, identificação de genes, comparação de bases de DNA ou aminoácidos, e identificação de regiões hidrofóbicas.

A seguir, daremos uma breve explicação da última aplicação acima.

4.1 Identificação de regiões hidrofóbicas

Nessa seção, mostramos uma aplicação do problema de subsequência máxima. Os detalhes podem ser vistos no Capítulo 11 intitulado *Predictive methods using protein sequences*, de autoria de S. Banerjee-Basu e A. D. Baxevanis, do livro organizado por

Baxevanis e Ouellette [7].

O capítulo em questão foca na utilização de técnicas computacionais para descobertas baseadas na sequência de aminoácidos. Diferente de DNA, composta por quatro bases, o alfabeto de 20 aminoácidos em proteínas permite uma maior diversidade de estrutura e função [7]. Baseada nas propriedades químicas de cada um dos 20 aminoácidos, ferramentas foram desenvolvidas para a predicção e identificação de proteínas com bases nessas propriedades. Uma tal ferramenta é ExPASy do Swiss Institute of Bioinformatics [5].

Inerente a cada um dos 20 aminoácidos é a hidrofobicidade: a tendência do ácido em enterrar-se no núcleo de uma proteína, distante de moléculas vizinhas de água. Essa tendência, junto com outras considerações, influência como uma proteína finalmente dobra na sua conformação 3-D.

TGREASE é uma ferramenta do sistema de programas FASTA [42] (disponível na Universidade de Virginia) para determinar as regiões hidrofóbicas de uma sequência de aminoácidos. A cada aminoácido é atribuído um valor numérico (positivo ou negativo) que reflete a hidrofobicidade relativa com base em um número de características físicas. Aminoácidos com valores altos são mais hidrofóbicos, ao passo que os aminoácidos com valores mais negativos são mais hidrofílicos. O problema de subsequência máxima é então resolvido para essa sequência de valores numéricos associados (repetidas vezes, a fim de obter todas as subsequências maximais), com as subsequências maximais obtidas correspondendo às regiões hidrofóbicas.

4.2 Algoritmo sequencial que obtém a subsequência máxima

Um algoritmo sequencial eficiente para obter a subsequência máxima foi descoberto por J. Kadane [6, 8]. Esse algoritmo é simples, elegante e tem complexidade $O(n)$. O algoritmo de Kadane se baseia na seguinte ideia. Se a subsequência máxima M de $(x_i, x_{i+1}, \ldots, x_k)$ e a sua pontuação T_M já foram determinadas, então, será possível estender facilmente este resultado para determinar a subsequência máxima da sequência $(x_i, x_{i+1}, \ldots, x_{k+1})$. Isto é mostrado no algoritmo 1.

Neste algoritmo consideramos dois casos:

Primeiro caso: x_k é o último número da sequência M. Se $x_{k+1} > 0$, então x_{k+1} será concatenado a M e o valor de x_{k+1} será somado a T_M. Caso contrário M e T_M permanecem inalterados.

Segundo caso: x_k não está na subsequência M. Nesse caso será preciso definir o sufixo máximo e a sua pontuação. O sufixo máximo da sequência $(x_1, x_2, \ldots, x_k)$ é o sufixo de máxima soma e será apresentado por $S = (x_s, \ldots, x_k)$. A sua pontuação será apresentada por T_S. As linhas de 6 a 14 do Algoritmo 1 mostram o que deve ser feito no segundo caso.

Algorithm 1 Subsequência Máxima Incremental.

Entrada: A sequência original $(x_1, x_2, ..., x_k)$ e o próximo elemento x_{k+1}, a subsequência máxima $M = (x_i, ..., x_j)$, a soma T_M, o sufixo $S = (x_s, ..., x_k)$, e a soma do sufixo T_s. Inicialmente M e S estão na 1.a posição do vetor, com T_M e T_S valendo 0.
Saída: A subsequência máxima atualizada da sequência $(x_1, x_2, ..., x_k, x_{k+1})$.

1: **if** x_k é o último número de M **then**
2: **if** $x_{k+1} > 0$ **then**
3: concatena x_{k+1} a M e ajusta $T_M = T_M + x_{k+1}$
4: **end if**
5: **else**
6: **if** $T_S + x_{k+1} > T_M$ **then**
7: concatena x_{k+1} a S e ajusta $T_S = T_S + x_{k+1}$, $M = S$ e $T_M = T_S$
8: **else**
9: **if** $T_S + x_{k+1} > 0$ **then**
10: concatena x_{k+1} a S e ajusta $T_S = T_S + x_{k+1}$
11: **else**
12: ajusta S para vazio e ajusta $T_S = 0$
13: **end if**
14: **end if**
15: **end if**

4.3 Algoritmo paralelo de Alves, Cáceres e Song utilizando o modelo CGM

No ano de 2004, Alves, Cáceres e Song publicaram um artigo [3] onde foi resolvido o problema da subsequência máxima utilizando o modelo CGM (*Coarse Grained Multicomputer*). Mais detalhes sobre o modelo CGM encontram-se na Seção 4.3.1.

O primeiro passo do algoritmo de Alves, Cáceres e Song [3] consiste em dividir a sequência de n números $(x_1, x_2, \ldots, x_n)$ entre os processadores. Assim, cada processador fica responsável por processar um intervalo da sequência original. Desse modo, dada uma sequência de n números $(x_1, x_2, \ldots, x_n)$, divide-se essa sequência entre os processadores, fazendo com que cada processador fique com um intervalo da sequência de $\frac{n}{p}$ números. É suposto que n é divisível por p, onde p denota a quantidade de processadores. O intervalo $(x_1, x_2, \ldots, x_{n/p})$ é armazenado no processador um, o intervalo $(x_{n/p+1}, \ldots, x_{2n/p})$ é armazenado no processador dois, e assim por diante. A Figura 4.1 ilustra como essa

divisão é feita, para $\frac{n}{p} = 3$.

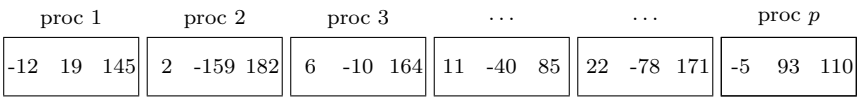


Figura 4.1: Cada processador recebe um intervalo de $\frac{n}{p}$ números.

O intervalo de $\frac{n}{p}$ números será representado por $I = (y_1, y_2, \ldots, y_{n/p})$ [3]. Para um intervalo qualquer J, será usada a notação T_J para representar a soma dos números do intervalo. Seja $M = (y_a, \ldots, y_b)$ a subsequência máxima de I. Cada intervalo I pode ser reduzido em apenas no máximo cinco números, representados por M, P, S, N_1, N_2 onde:

1. $M = (y_a, \ldots, y_b)$ é a subsequência máxima de I com $T_M \geq 0$
2. $P = (y_1, \ldots, y_r)$ é o prefixo máximo de $(y_1, y_2, \ldots, y_{a-1})$ com $T_P \geq 0$
3. $S = (y_s, \ldots, y_{n/p})$ é o sufixo máximo de $(y_{b+1}, \ldots, y_{n/p})$ com $T_S \geq 0$
4. N_1 é o intervalo entre P e M com $T_{N_1} \leq 0$
5. N_2 é o intervalo entre M e S com $T_{N_2} \leq 0$

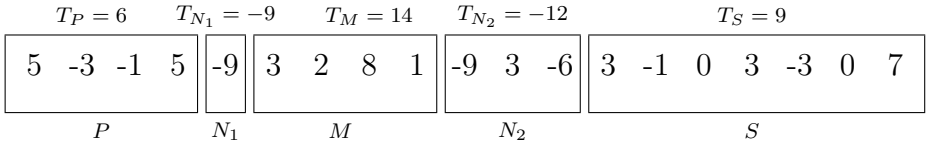


Figura 4.2: Exemplo dos cinco valores.

O seguinte lema pode ser demonstrado.

Lema 1 *Se M não é vazio, então um dos seguintes casos é válido [3].*

1. *P está à esquerda de M, com $r < a$, e com N_1 no meio.*
2. *P é vazio. Temos N_1 à esquerda de M.*
3. *P é vazio e N_1 é vazio.*

Em relação ao sufixo máximo S, temos um lema semelhante. A prova é similar ao lema anterior.

Lema 2 *Se M não é vazio, então um dos seguintes casos é válido [3].*

1. *S está à direita de M, com $s > b$, e com N_2 no meio.*

2. *S é vazio. Temos N_2 à direita de M.*

3. *S é vazio. N_2 é vazio.*

Quando P (resp. N_1, M, N_2, S) é vazio, T_P (resp. T_{N_1}, T_M, T_{N_2}, T_S) será definido como sendo igual a 0.

Se todos os números y_i são negativos supomos M, P, S vazios com $T_M = T_P = T_S = 0$, e podemos definir $T_{N_1} = 0$ e T_{N_2} = soma de todos os números negativos.

Assim,

$$T_P + T_{N_1} + T_M + T_{N_2} + T_S = \sum_{i=1}^{n/p} y_i.$$

Os números preparados por cada processador, T_P, T_{N_1}, T_M, T_{N_2} e T_S, são usados no algoritmo paralelo.

Tendo computado os cinco números acima, cada processador os envia ao processador 1. O processador 1 recebe assim $5p$ números e resolve sequencialmente o problema de subsequência máxima dos $5p$ números e assim obtém a soma da subsequência máxima [2]. Esse processamento é feito no tempo de $O(p)$. Se além da soma, deseja-se também a subsequência máxima, então basta considerar para cada um dos cinco números também os índices do seu início e término. Por exemplo, além de considerar T_M, devemos armazenar os índices a e b.

Então o resultado final é dado [3]:

Teorema 1 *O algoritmo paralelo computa a soma da subsequência máxima de uma sequência dada $(x_1, x_2, \ldots, x_n)$ em um número constante de rodadas de comunicação, envolvendo a transmissão de $O(p)$ números e tempo de computação local de $O(n/p)$.*

4.3.1 Modelo CGM

O modelo *Coarse Grained Multicomputer* (CGM) foi proposto por Dehne et al. [12]. Este modelo é composto por p processadores cada um com memória local, conectados por um meio qualquer de intercomunicação [27]. No modelo CGM os processadores processam o mesmo algoritmo. Os algoritmos processados no modelo CGM realizam sequências de

super-passos [13], onde cada super-passo é dividido em uma fase de computação local e uma fase de comunicação global. A Figura 4.3 ilustra o modelo CGM.

Normalmente, durante uma rodada de computação local é utilizado o melhor algoritmo sequencial para o processamento dos dados disponibilizados localmente em cada processador. O modelo CGM considera dois parâmetros [17] para a análise dos algoritmos:

1. O tamanho do problema n;
2. O número de processadores disponíveis p.

Em uma rodada de comunicação cada processador envia $O(\frac{n}{p})$ dados e recebe $O(\frac{n}{p})$ dados. O custo de um algoritmo CGM é composto pela soma dos tempos obtidos nas rodadas de computação local e de comunicação.

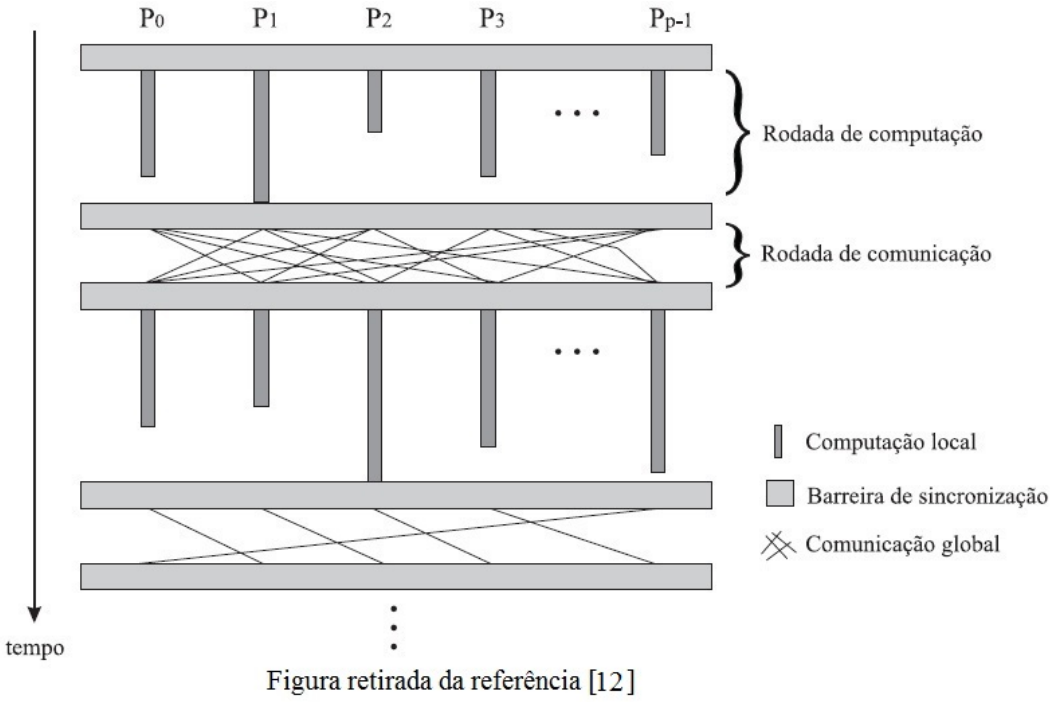


Figura 4.3: Modelo CGM.

4.4 Implementação do algoritmo paralelo de Alves, Cáceres e Song em GPU

A contribuição desse trabalho consiste em implementar e adaptar o algoritmo paralelo de Alves, Cáceres e Song [3] em placas gráficas GPUs.

Para realizar a implementação do algoritmo paralelo de Alves, Cáceres e Song [3] em uma GPU, foi necessário alocar um vetor de dados na CPU para armazenar os n números da sequência dada. Após o seu preenchimento, foi utilizada a função *cudaMemcpy()* [30] para copiá-lo para a GPU. A função *cudaMemcpy()* copia os dados para a memória global da GPU [30]. A memória global é o primeiro nível de memória de uma GPU. As primeiras implementações dessa pesquisa foram realizadas da seguinte maneira: cada *thread* acessava os seus dados diretamente da memória global. A sequência original foi dividida em intervalos menores, deixando assim, cada *thread* responsável por um intervalo de $n/qtdThreads$, onde:

- n denota a quantidade total de elementos do vetor de dados, e
- $qtdThreads$ denota a quantidade total de *threads* criadas na GPU.

Depois que cada *thread* obteve o seu intervalo de dados, cada *thread* processa o seu intervalo e no final gera os seus cincos valores, conforme descritos na Seção 4.3. Com os cincos valores fornecidos por cada *thread*, é formado um novo vetor, que é copiado de volta para a CPU. A CPU executa o algoritmo de Kadane [8, 6] sobre este vetor, obtendo assim a subsequência máxima.

Nessa primeira implementação, as *threads* acessavam seus dados diretamente da memória global. Agindo dessa forma, não foi possível obter um bom desempenho. Como mencionado na Seção 2.3.1, o acesso à memória global é muito custoso e lento [31]. Após realizar diversos testes, o máximo que foi possível obter foi um *speedup* de 1. *Speedup* é uma medida muito utilizada na computação de alto desempenho. Com o *speedup* é possível mensurar o quanto que uma aplicação paralela foi mais rápida que a sequencial [46]. Em modelos de computação paralela como o CGM, *speedup* é definido por:

$$S_p = \frac{T_1}{T_p}$$

onde:

- p denota o número de processadores;
- T_1 denota o tempo de execução do programa sequencial;
- T_p denota o tempo de execução do programa paralelo com p processadores.

Na nossa implementação em GPU, consideramos T_1 como sendo o tempo de execução usando somente a CPU e T_p como sendo o tempo de execução usando CPU em conjunto com a GPU.

Para obter um bom *speedup* foi preciso utilizar a memória compartilhada. Com o uso da memória compartilhada, foi possível diminuir a latência do acesso à memória. Esse era um dos fatores que estavam degradando o desempenho dessa aplicação. Pelo fato de a memória compartilhada ser uma memória *on-chip*, ela é caracterizada por ter um acesso rápido [31]. O único porém é que o seu tamanho é muito reduzido. Desse modo, foi necessário elaborar uma estratégia de distribuição de dados.

Em CUDA existe uma técnica chamada *coalescing* [30], com a qual é possível diminuir a latência de acesso à memória. Em placas gráficas de *compute capability* superior a 1.2 o *coalescing* é realizado com um conjunto de 32 *threads*. Essa técnica diz que se forem utilizadas 32 *threads* para lerem 32 elementos da memória global de uma vez só, é possível diminuir a latência em até 10 vezes [30].

A estratégia elaborada é composta por três etapas. A primeira etapa consiste em declarar um vetor de dados na memória compartilhada e, logo após, utilizar a técnica do *coalescing* para preenchê-lo com os dados. Com isso, foi possível realizar a transferência de dados da memória global para a memória compartilhada de maneira eficiente.

A GPU utilizada nessa pesquisa possui uma memória compartilhada de 16 Kbytes. Nessa implementação foram utilizados números do tipo inteiro que consomem 4 bytes. Com isso, foi possível alocar na memória compartilhada um vetor de 4.096 elementos. A memória compartilhada é acessível por bloco. Cada bloco de *thread* lançado na GPU consegue alocar um vetor de 4.096 elementos. Desse modo, se lançamos um *kernel* com k blocos, teremos k vetores de 4.096 elementos.

A segunda etapa da estratégia elaborada consiste em determinar o número de *threads* e blocos lançados. Foi lançado um *kernel* com k blocos e com 128 *threads* em cada bloco. A sequência original dos n números foi dividida em intervalos menores, deixando assim, cada bloco responsável por um intervalo $I = \frac{n}{qtdBlocos}$ onde:

- n denota a quantidade total de elementos do vetor de dados;
- $qtdBlocos$ denota a quantidade total de blocos lançados na GPU.

A figura abaixo mostra como essa divisão foi realizada.

Depois que cada bloco obteve o seu intervalo I, dividimos o intervalo I em J intervalos de 4.096 elementos, onde $J = \frac{I}{4.096}$. Agora cada bloco fica responsável por processar J intervalos de 4.096 elementos. Para facilitar o entendimento, um intervalo de 4.096 elementos será representado pela letra Z. A Figura 4.5 ilustra como essa divisão foi realizada.

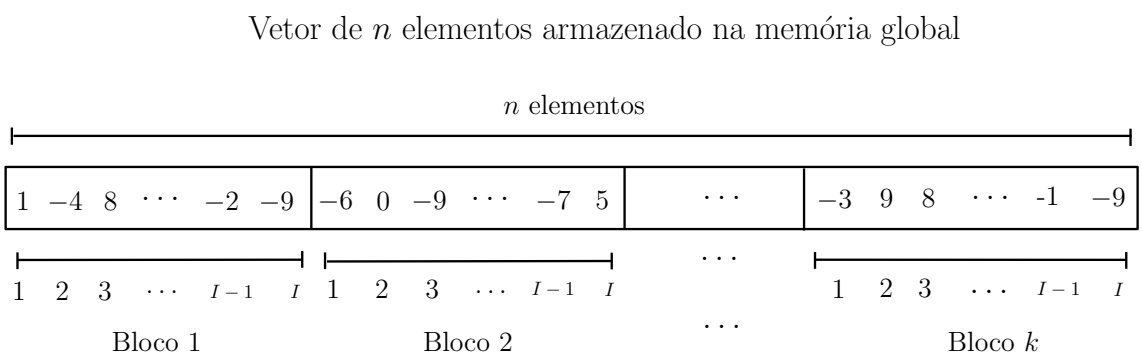


Figura 4.4: Intervalos I.

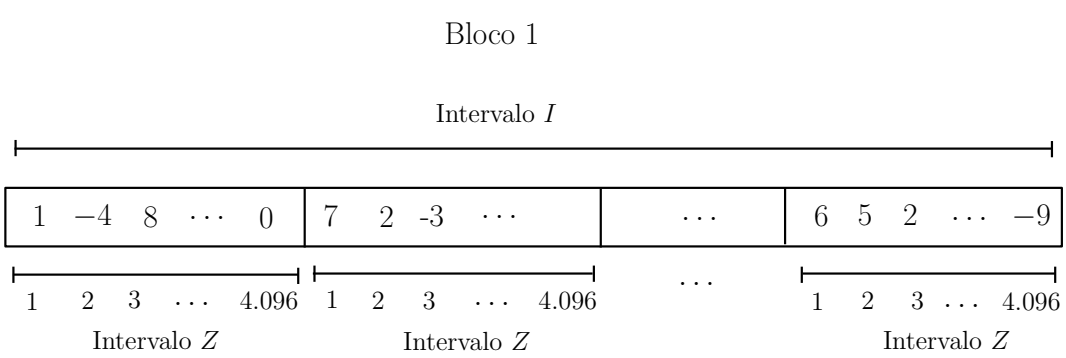


Figura 4.5: Intervalos Z.

Cada bloco transfere o seu intervalo Z da memória global, para a memória compartilhada (vetor compartilhado). Depois que os dados foram transferidos para a memória compartilhada, eles são processados. Cada bloco repete esses dois procedimentos (transferências de dados e processamento) J vezes, até que todos intervalos Z sejam processados e, consequentemente até que todos blocos processem o seu intervalo I.

A parte final da estratégia consiste em fazer com que as *threads* utilizem o vetor compartilhado e não mais o vetor armazenado na memória global. Uma das dificuldades que foi enfrentada nessa pesquisa foi o tamanho reduzido do vetor compartilhado. Esse vetor compartilhado possui capacidade de somente 4.096 elementos. Desse modo, foi preciso determinar de forma eficiente a quantidade de *threads* em cada bloco, para que cada *thread* tenha uma quantidade suficiente de dados para processar. Foram lançadas 128 *threads* em cada bloco, com isso, cada *thread* ficou responsável por processar uma quantidade de 32 elementos do vetor compartilhado. Dessa forma, todas as 128 *threads* trabalham em paralelo utilizando o vetor compartilhado.

As *threads* trabalham da seguinte maneira: primeiro as 128 *threads* transferem os dados da memória global para a memória compartilhada (vetor compartilhado). A transferência é realizada em 4 grupos de 32 *threads*. Dessa forma, é possível utilizar a técnica do *coalescing* [30]. Depois da transferência, cada *thread* processa os seus 32 elementos. Após o processamento, todas elas transferem novamente os dados para o vetor compartilhado

e novamente cada *thread* processa os seus 32 elementos. Cada vez que a transferência e processamento de dados são executados, as 128 *threads* processam um intervalo Z e consequentemente processam um vetor compartilhado (128 *threads* $\times$ 32 elementos = 4.096 elementos processados). Esse procedimento só irá finalizar, quando todos blocos tiverem processado o seu intervalo I e consequentemente tiverem processados J intervalos Z. É possível abstrair melhor esse procedimento observando a Figura 4.6.

No final de todo processo, cada *thread* gera os cinco valores. Com os cincos valores de cada *thread* é formado um novo vetor, que é copiado de volta para a CPU. No final o processamento sequencial é realizado na CPU e a subsequência máxima é obtida.

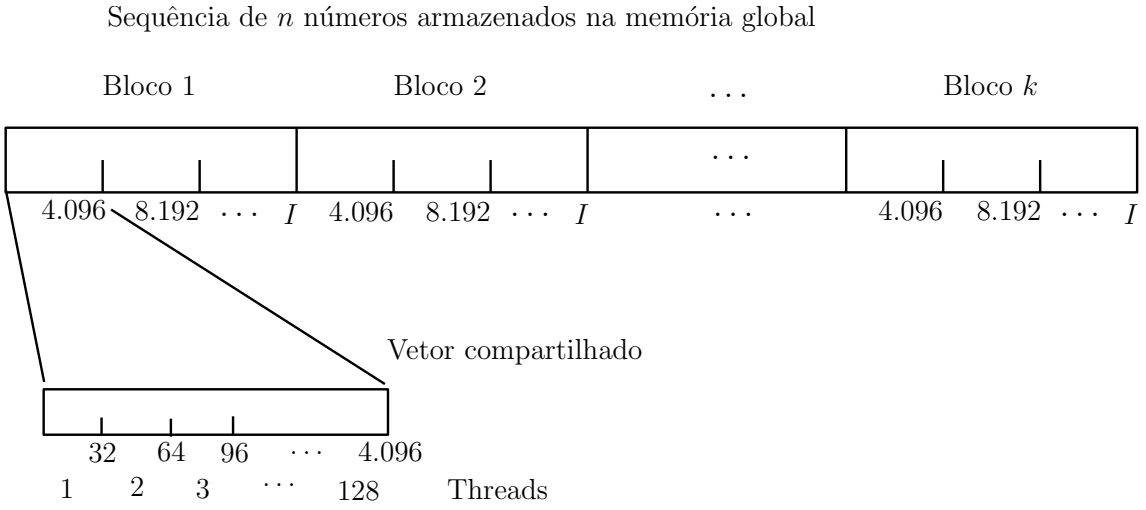


Figura 4.6: Distribuição dos dados para o processamento.

No capítulo seguinte, introduzimos o problema da submatriz máxima. No Capítulo 6, são apresentados resultados experimentais para ambos os problemas.

Capítulo 5

Submatriz máxima

O problema de submatriz máxima é uma variação do problema da subsequência máxima. O problema de submatriz máxima aparece em reconhecimento de padrão bidimensional, e corresponde a um estimador de semelhança a um padrão em uma imagem digital [43].

Existem vários artigos que discutem e propõem algoritmos para a resolução desse problema. No ano de 1995, Perumalla e Deo publicaram um artigo [43], no qual propuseram um algoritmo paralelo para a resolução desse problema utilizando o modelo PRAM [46]. Dada uma matriz $n \times n$, esse algoritmo obtém uma submatriz de soma máxima em tempo $O(\log n)$.

No ano de 2004, Alves, Cáceres e Song publicaram um trabalho [3], no qual propuseram uma adaptação do algoritmo de Perumalla e Deo para o modelo CGM. Esse algoritmo requer $O(1)$ rodadas de comunicação.

A seguir, será apresentada a definição do problema de submatriz máxima. Seja dada uma matriz K, com elementos $k_{i,j}$, com $0 \leq i \leq n-1$ e $0 \leq j \leq m-1$, onde n e m denotam, respectivamente, a quantidade de linhas e colunas da matriz K. Deseja-se encontrar uma submatriz de K tal que a soma de seus elementos seja máxima [3]. A pontuação dessa submatriz será representada por T_M. Para este problema, os elementos da matriz K podem conter valores positivos ou negativos. Será suposto que pelo menos um dos elementos da matriz K tenha valor positivo. Com isso, haverá uma pontuação positiva para T_M [3].

Caso todos os elementos da matriz K tenham valores positivos, a submatriz de soma máxima será a própria matriz K. Por simplicidade, sem perda de generalidade, supõe-se $m = n$, isto é, consideramos K como sendo uma matriz $n \times n$.

Para simplificar a apresentação a seguir, usa-se a seguinte notação. Dada uma matriz K de n linhas e n colunas, a notação $K_{[i_1,i_2][j_1,j_2]}$ denota uma submatriz de K com elementos k_{ij} onde, $i_1 \leq i \leq i_2$ e $j_1 \leq j \leq j_2$. Como exemplo, considere a seguinte matriz K:

$$K = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} -10 & -10 & 20 & -20 & -30 \\ 5 & -20 & 10 & 40 & 10 \\ 30 & -40 & 20 & -10 & -15 \\ -20 & 4 & -5 & 50 & 10 \\ 10 & -20 & 10 & -40 & 10 \end{bmatrix} \end{array}$$

A submatriz de soma máxima é:

$$K_{[1,3][2,4]} = \begin{bmatrix} 10 & 40 & 10 \\ 20 & -10 & -15 \\ -5 & 50 & 10 \end{bmatrix}$$

e o seu T_M é 110.

A seguir, será apresentado o algoritmo paralelo de Perumulla e Deo [43] e Alves, Cáceres e Song [3].

5.1 Algoritmo paralelo

O algoritmo paralelo recebe uma matriz K como entrada. A seguinte matriz K será utilizada como exemplo, afim de mostrar todos os passos do algoritmo.

$$K = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} -10 & -10 & 20 & -20 & -30 \\ 5 & -20 & 10 & 40 & 10 \\ 30 & -40 & 20 & -10 & -15 \\ -20 & 4 & -5 & 50 & 10 \\ 10 & -20 & 10 & -40 & 10 \end{bmatrix} \end{array}$$

Considerem-se todas as submatrizes de K da forma $K_{[1,n][g,h]}$, onde $0 \leq g \leq h \leq n-1$.

Por exemplo, para $g = 0$ e $h = 2$, tem-se:

$$K_{[0,4][0,2]} = \begin{bmatrix} -10 & -10 & 20 \\ 5 & -20 & 10 \\ 30 & -40 & 20 \\ -20 & 4 & -5 \\ 10 & -20 & 10 \end{bmatrix}$$

Para cada submatriz $K_{[0,n-1][g,h]}$, calcula-se o vetor coluna $C^{g,h}$ de n elementos, onde

$$c_i^{g,h} = \sum_{j=g}^{h} k_{i,j}$$

No exemplo, para $K_{[0,4][0,2]}$, tem-se o seguinte vetor coluna:

$$C^{0,2} = \begin{bmatrix} 0 \\ -5 \\ 10 \\ -21 \\ 0 \end{bmatrix}$$

Em seguida, para cada vetor coluna $C^{g,h}$ assim obtido, computa-se a sua subsequência máxima.

No exemplo do vetor coluna $C^{0,2}$, a subsequência máxima da sequência $(0, -5, 10, -21, 0)$ é (10) com soma 10.

Para obter a soma T_M da submatriz máxima, o algoritmo paralelo considera a maior soma de todas as subsequências máximas assim obtidas.

No exemplo, para $g = 2$ e $h = 4$, tem-se

$$K_{[0,4][2,4]} = \begin{bmatrix} 20 & -20 & -30 \\ 10 & 40 & 10 \\ 20 & -10 & -15 \\ -5 & 50 & 10 \\ 10 & -40 & 10 \end{bmatrix}$$

O vetor coluna correspondente é $C^{2,4}$:

$$C^{2,4} = \begin{bmatrix} -30 \\ 60 \\ -5 \\ 55 \\ -20 \end{bmatrix}$$

Para os números deste vetor, a subsequência máxima é $(60, -5, 55)$ com soma igual a 110. Essa soma é a maior soma ao considerar todos os pares de g e h, para $0 \leq g \leq h \leq 4$.

Assim a submatriz máxima da matriz K é $K_{[0,4][2,4]}$, com $T_M = 110$.

5.1.1 Uso da soma de prefixos para acelerar a soma de um intervalo

Como visto acima, um passo do algoritmo paralelo consiste em obter o vetor coluna $C^{g,h}$, onde cada elemento é obtido assim:

$$c_i^{g,h} = \sum_{j=g}^{h} k_{i,j}$$

Cada elemento de $C^{g,h}$ é a soma de um intervalo contíguo de uma linha da matriz K, entre as posições g e h.

Essa soma pode ser obtida em tempo constante, com um pré-processamento de cada linha da matriz K. Esse pré-processamento consiste no cálculo da soma de prefixos, assim definida:

Dada uma sequência de números $U = (u_0, u_1, u_2, \ldots, u_{n-1})$, a soma de prefixos da sequência é uma sequência $V = (v_0, v_1, v_2, \ldots, v_{n-1})$ onde,

$$v_i = \sum_{k=1}^{i} u_k.$$

Uma forma eficiente de calcular a soma de prefixos é usar a seguinte recorrência:

$$v_0 = 0$$

$$v_i = v_{i-1} + u_i$$

$$\text{para } \; 1 \leq i \leq n-1.$$

Uma vez feito o pré-processamento do cálculo da soma de prefixos, a soma de um intervalo contíguo qualquer da sequência U pode ser obtida em tempo constante.

Assim, para obter a soma $u_g + \ldots + u_h$, basta fazer a subtração $v_h - v_{g-1}$, pois:

$$v_h - v_{g-1} = (u_1 + \ldots + u_{g-1} + u_g + \ldots + u_h) - (u_1 + \ldots + v_{g-1}) = u_g + \ldots + u_h.$$

5.2 Implementação paralela utilizando GPU

Como mencionado na seção anterior, algoritmos paralelos para o problema de submatriz máxima foram apresentados para o modelo PRAM [43] e para o modelo CGM [3]. Esse trabalho de pesquisa consiste em propor uma implementação desse algoritmo paralelo usando GPUs.

Para implementar o algoritmo aqui abordado em GPUs, consideramos e dividimos o algoritmo em três etapas. Considere a matriz K, de n linhas e n colunas. A primeira etapa consiste em realizar a soma de prefixos em cada linha da matriz K. A segunda etapa consiste em determinar as submatrizes $K_{[0,n-1][g,h]}$, os vetores colunas $C^{g,h}$, e calcular a subsequência máxima de cada vetor coluna $C^{g,h}$. A terceira etapa consiste em encontrar o maior valor de todas as subsequências máximas obtidas, obtendo-se assim T_M.

Seja dada uma matriz de entrada K de n linhas e n colunas. Essa matriz é linearizada, por linhas, dando origem a um vetor de n^2 elementos. A implementação é iniciada com a alocação de um vetor de tamanho n^2 para armazenar esses elementos. Foi utilizada a função *cudaMemcpy()* para copiar esse vetor para a GPU. A função *cudaMemcpy()* copia os dados para a memória global da GPU.

5.2.1 Primeira etapa: soma de prefixos

A primeiro etapa da implementação obtém a soma de prefixos em cada linha da matriz K. É lançado um *kernel* para realizar a soma de prefixos. Nesse *kernel*, a quantidade de *threads* é igual ao número de linhas da matriz K. Cada *thread* fica responsável por realizar a soma de prefixos em uma determinada linha de K. Caso a quantidade de linhas da matriz for superior ao número de *threads*, cada *thread* ficará responsável por $n/qtdThread$ linhas da matriz K, onde:

1. n denota a quantidade de linhas da matriz K, e

2. *qtdThread* denota a quantidade total de *threads* lançadas na GPU.

A figura abaixo ilustra a realização da soma de prefixos.

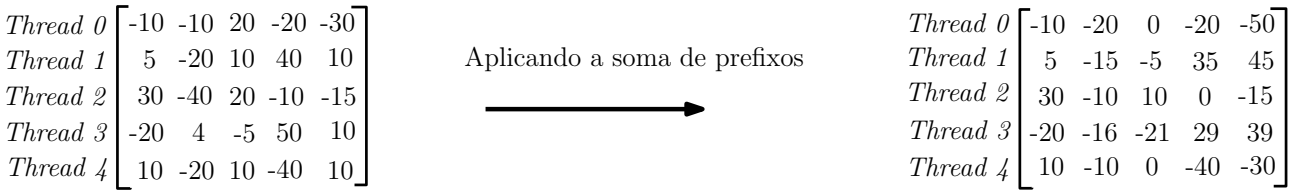


Figura 5.1: Soma de prefixos em paralelo.

5.2.2 Segunda etapa: determinar $K_{[0,n-1][g,h]}$, $C^{g,h}$ e as subsequências máximas de $C^{g,h}$

Depois que a soma de prefixos foi realizada em cada linha da matriz K, realizada na primeira etapa, a segunda etapa é determinar as submatrizes $K_{[0,n-1][g,h]}$, obter os vetores coluna $C^{g,h}$ e as suas subsequências máximas. Para isso é lançado um *kernel* com 16 blocos, com 1.024 *threads* em cada bloco, totalizando 16.384 *threads*.

Para determinar as submatrizes $K_{[0,n-1][g,h]}$, é preciso determinar pares de índices (g,h). Como visto na seção anterior, cada par de índices (g,h) delimita uma submatriz $K_{[0,n-1][g,h]}$. Nessa implementação cada *thread* fica responsável por um par de índices (g,h) e, consequentemente, por uma submatriz $K_{[0,n-1][g,h]}$. Quando a quantidade de pares (g,h) for superior a quantidade de *threads*, divide-se a quantidade de pares de índices (g,h) entre as *threads*. Desse modo, cada *thread* fica responsável por mais de um par de índices (g,h). O cálculo dos pares de índices (g,h) é realizado em tempo de execução. Cada *thread* tem um índice de identificação chamado *id*, onde $id = 0, 1, 2, \ldots, 16.383$. Seja n a quantidade de colunas da matriz K. Conforme dedução a ser apresentada na Seção 5.3, o cálculo de g e h do par (g,h) associado à *thread* $id = \ell$ é feito pelas fórmulas:

$$g = \lfloor \frac{(2n+1) - \sqrt{(4n^2 + 4n + 1 - 8\ell)}}{2} \rfloor.$$

$$h = \ell - \frac{(2n-1)g - g^2}{2}.$$

Depois que cada *thread* calculou os seus respectivos índices g e h, os próximos passos que cada *thread* irá realizar são: calcular o vetor coluna $C^{g,h}$ e a subsequência máxima de $C^{g,h}$. Como visto na seção anterior, cada elemento do vetor coluna $C^{g,h}$ é calculado por:

$$c_i^{g,h} = \sum_{j=g}^{h} k_{i,j}$$

Isto é, cada elemento $c_i^{g,h}$ é a soma de um intervalo contíguo de uma linha da matriz K, entre as posições g e h. Para obter a soma dos elementos de g até h, cada *thread* realiza a subtração $v_h - v_{g-1}$, pois:

$$v_h - v_{g-1} = (u_1 + \ldots + u_{g-1} + u_g + \ldots + u_h) - (u_1 + \ldots + v_{g-1}) = u_g + \ldots + u_h.$$

Note-se que a soma de prefixos já foi calculada na primeira etapa (Seção 5.2.1). Como exemplo, a figura abaixo mostra as *threads* de $id = 7$ e $id = 11$ com as suas respectivas submatrizes e os seus vetores colunas.

Para a *thread* de *id = 7*, temos $g = 1$ e $h = 3$. Para a *thread* de *id = 11*, temos $g = 2$ e $h = 4$.

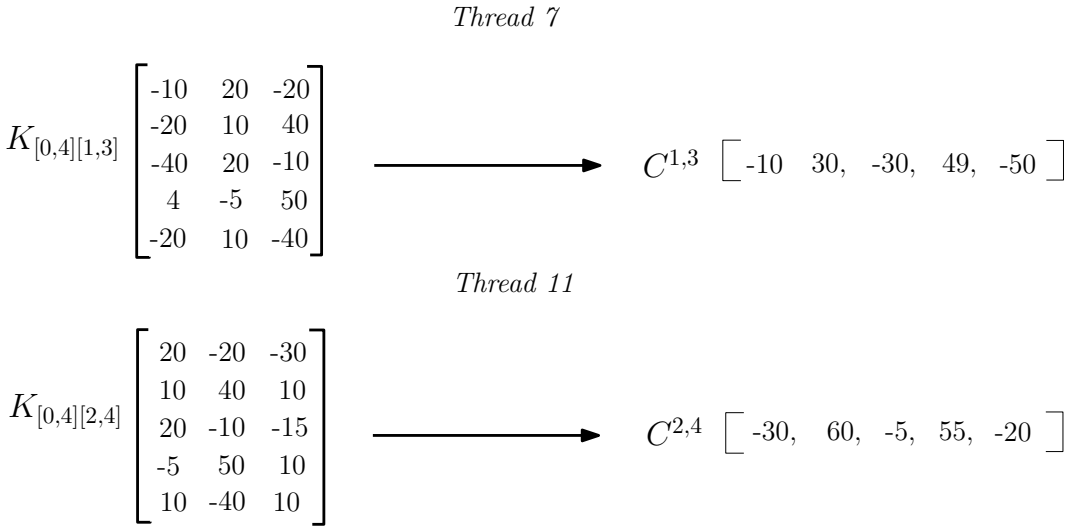


Figura 5.2: Exemplo de submatrizes $K_{[0,n-1][g,h]}$ e vetores colunas $C^{g,h}$.

Após calcular o vetor de colunas $C^{g,h}$, cada *thread* irá obter a sua subsequência máxima. A resposta T_M será a maior soma de todas as somas obtidas correspondentes às subsequências máximas obtidas. No exemplo, para $id = 7$, a subsequência máxima de $(-10, 30, -30, 49, -50)$ é (49) de soma 49. Para *id = 11* a subsequência máxima de $(-30, 60, -5, 55, -20)$ é $(60, -5, 55)$ de soma 110, que é a maior soma de todas e portanto o valor da submatriz de soma máxima de K é 110.

Foi alocado um vetor Vet na memória compartilhada. Esse vetor tem a capacidade de armazenar 12.228 elementos. Vet armazena os elementos $c_i^{g,h}$ dos vetores coluna $C^{g,h}$. Na

computação do vetor coluna $C^{g,h}$, cada *threads* computa 12 elementos $c_i^{g,h}$ que são armazenados em Vet. Em cada bloco de *threads*, há um total de 1.204 *threads* sendo executadas. Dessa forma, cada *thread* armazena 12 elementos $c_i^{g,h}$ em Vet ($1.024 \times 12 = 12.228$). Após o armazenamento, cada *thread* aplica o algoritmo de Kadane em seus 12 elementos $c_i^{g,h}$ armazenados em Vet. Depois de aplicar o algoritmo de Kadane, cada *thread* computa novamente mais 12 elementos $c_i^{g,h}$ e armazena-os em Vet, e assim sucessivamente. Dessa forma, esse procedimento se repete até que todas as *threads* tenham computado e processado (aplicar o algoritmo de Kadane) todos elementos $c_i^{g,h}$ de seus respectivos vetores coluna $C^{g,h}$. A Figura 5.3 ilustra esse procedimento. Após o término do processamento dos vetores coluna $C^{g,h}$, o valor da soma de cada subsequência máxima é obtido e armazenado em um vetor W.

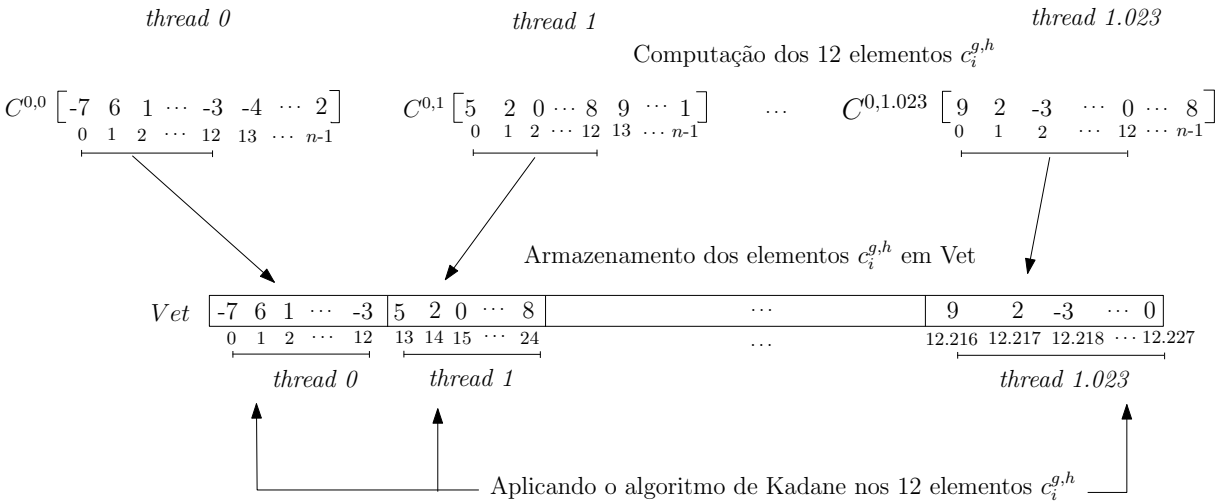


Figura 5.3: Processamento dos vetores colunas $C^{g,h}$

O algoritmo de Kadane [8, 6] é um algoritmo linear. Então, o fato de processarmos os dados em 12 e 12 elementos não implica em nenhum fator negativo. É preciso transferir os elementos em 12 e 12 pois o vetor compartilhado consegue armazenar somente 12.228 elementos ($1.024 \times 12 = 12.228$).

5.2.3 Terceira parte: obter o maior valor de todas as subsequências máximas

Nessa implementação cada *thread* fica responsável por mais de um vetor coluna $C^{g,h}$. Desse modo, no final da execução da segunda etapa cada *thread* obtém mais de um valor soma e armazena a maior delas no vetor W. Conforme mencionado no início da Seção 5.2.2, foi lançado um *kernel* com um total de 16.384 *threads*. O vetor W armazena portanto 16.384 valores.

A terceira etapa consiste na obtenção do maior valor entre os armazenados no vetor W, que foi preparado na segunda etapa. Para isso, W é copiado para a CPU que obtém o máximo que será a resposta T_M desejada.

Resultados experimentais são apresentados no Capítulo 6.

5.3 Pares de índices (g,h)

GPUs processam dados utilizando o modelo de programação paralela SIMD [44]. Esse modelo parte do princípio de que, para obter um bom desempenho no processamento de dados, todas *threads* têm que iniciar e finalizar o seu processamento ao mesmo tempo. Se uma *thread* iniciar o seu processamento depois das demais *threads*, o desempenho da aplicação sofrerá degradação. Tendo em vista a necessidade de todas as *threads* iniciarem e terminarem o processamento juntas, cada *thread* calcula o par de índices g e h cuja execução é de sua responsabilidade, e ao mesmo tempo em que as demais *threads* estão executando o mesmo cálculo.

Nessa implementação cada *thread* ficou responsável por um par de índices (g, h). É útil ter uma fórmula que associa um par (g, h) a cada *id* de uma *thread*. A seguir, deduzimos tal fórmula. Abaixo seguem alguns exemplos de pares de índices (g, h), atribuídos a diferentes *threads*.

	g	h		
par	0	0	-	*thread* 0
par	0	1	-	*thread* 1
par	0	2	-	*thread* 2
		.		
		.		
par	0	$n-1$	-	*thread* $n-1$
par	1	1	-	*thread* n
par	1	2	-	*thread* $n+1$
par	1	3	-	*thread* $n+2$
		.		
		.		
par	1	$n-1$	-	*thread* $2n-3$
par	2	2	-	*thread* $2n-2$
		.		
		.		

Para cada g variando de 0 até $n-1$, temos $(n-g)$ valores possíveis para h. A

quantidade de valores possíveis para g é igual a n. A quantidade de pares possíveis (g, h) é $n(n+1)/2$, onde n denota a quantidade de linhas e de colunas da matriz K. Então, quando g vale 0 temos n valores possíveis para h, quando g vale 1 temos $n-1$ valores possíveis para h, e assim por diante. Podemos visualizar a associação de pares de índices (g, h) a *threads* por meio de uma matriz M de dimensão $n \times n$. Na linha g e coluna h, a entrada $m_{g,h}$ tem o *id* da *thread* que irá processar o par (g, h). Como $g \leq h$, só são importantes as entradas de M da diagonal principal e as acima da diagonal principal. A matriz M pode ser visualizada na Figura 5.4.

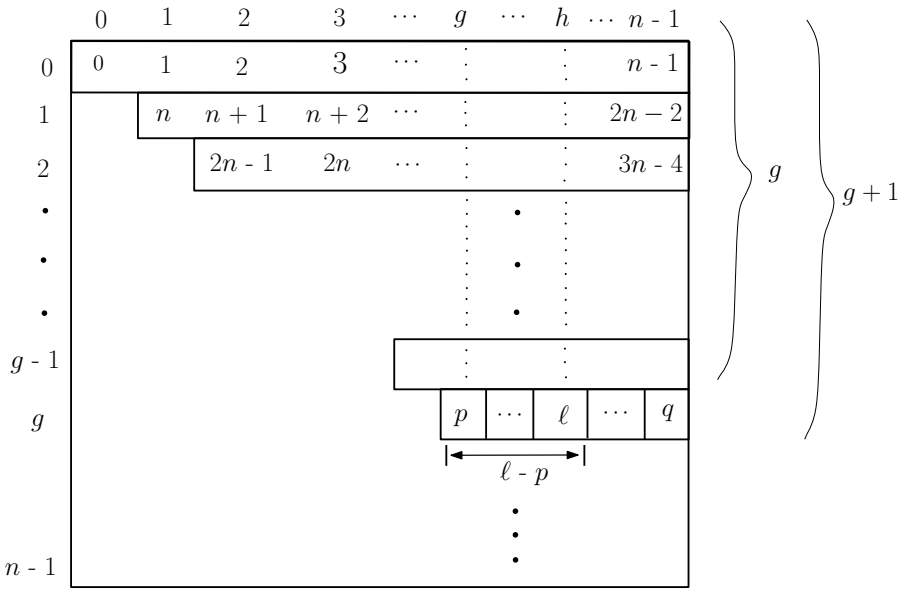


Figura 5.4: Matriz M de *threads*: $m_{g,h}$ é o índice da *thread* que processa o par (g, h)

Nessa pesquisa nos deparamos com o seguinte problema: dado o $id = \ell$ de uma *thread* qualquer, temos que descobrir qual é o par de índices (g, h) que a *thread* ℓ irá processar. O primeiro passo é descobrir o índice da linha de M que contém a entrada de valor ℓ. Seja g o índice dessa linha. Seja p o *id* da primeira *thread* da linha g. Seja q o *id* da última *thread* na linha g (ver Figura 5.4). Observe que:

$$p \leq \ell \leq q \tag{5.1}$$

Calculando o valor de p e q temos

$$p = n + (n-1) + \cdots + n - (g-1).$$

e temos também

$$q = n + (n-1) + \cdots + (n-g) - 1.$$

Portanto, para descobrir o valor de g, devemos resolver as duas inequações em (5.1)

na variável g.

Vamos primeiro simplificar a inequação $p \leq \ell$, ou seja, $n+(n-1)+\cdots+n-(g-1) \leq \ell$.

$$\frac{[n + (n - g + 1)]g}{2} \leq \ell$$

$$\frac{(2n - g + 1)g}{2} \leq \ell$$

$$2ng - g^2 + g \leq 2\ell$$

$$g^2 - (2n + 1)g + 2\ell \geq 0$$

Calcula-se o discriminante Δ:

$$\Delta = (2n + 1)^2 - 8\ell$$

$$\Delta = 4n^2 + 4n + 1 - 8\ell$$

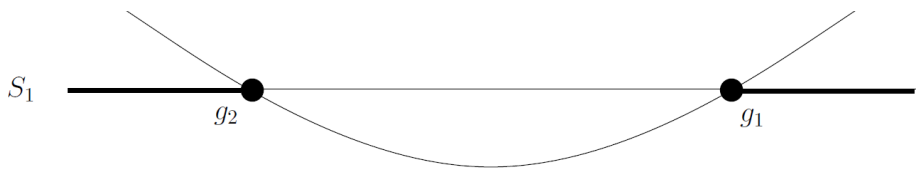
Obtêm-se as duas raízes:

$$g_1 = \frac{(2n + 1) + \sqrt{(4n^2 + 4n + 1 - 8\ell)}}{2}$$

e

$$g_2 = \frac{(2n + 1) - \sqrt{(4n^2 + 4n + 1 - 8\ell)}}{2}.$$

Representando graficamente o conjunto solução S_1 da inequação $p \leq \ell$, temos:



Ou seja, o conjunto solução é $S_1 = (-\infty, g_2] \cup [g_1, +\infty)$.

Vamos simplificar agora a inequação $\ell \leq q$, que é equivalente (porque ℓ, n e g são inteiros) à inequação $\ell < n + (n - 1) + \cdots + (n - g)$.

$$\ell < \frac{[n + (n - g)](g + 1)}{2}$$

$$\ell < \frac{(2n - g)(g + 1)}{2}$$

$$2\ell < 2ng + 2n - g^2 - g$$

$$g^2 + (1 - 2n)g + (2\ell - 2n) < 0$$

Calcula-se o discriminante Δ :

$$\Delta = (1 - 2n)^2 - 4(2\ell - 2n)$$

$$\Delta = 1 + 4n^2 - 4n - 8\ell + 8n$$

$$\Delta = 4n^2 + 4n + 1 - 8\ell$$

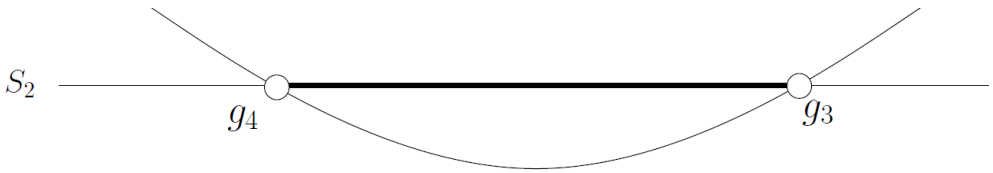
Obtêm-se as duas raízes:

$$g_3 = \frac{(2n - 1) + \sqrt{4n^2 + 4n + 1 - 8\ell}}{2}$$

e

$$g_4 = \frac{(2n - 1) - \sqrt{4n^2 + 4n + 1 - 8\ell}}{2}.$$

Representando graficamente a solução S_2 da inequação $\ell < n + (n - 1) + \cdots + (n - g)$ temos:



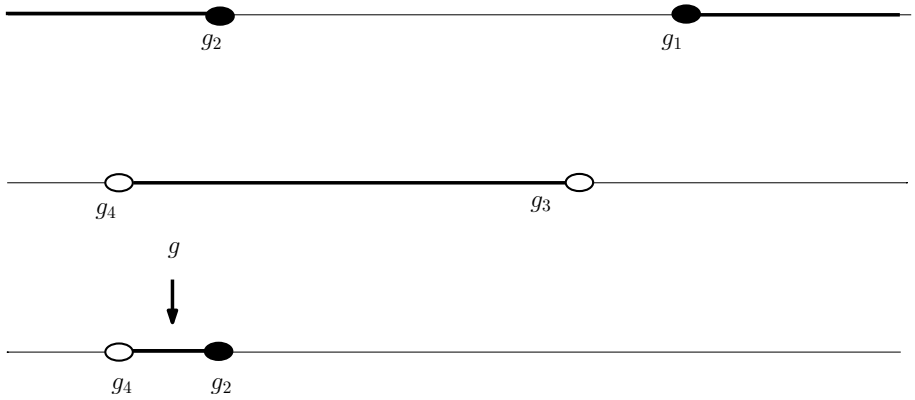
O conjunto solução é $S_2 = (g_4, g_3)$.

Temos então que

$$g \in S_1 \cap S_2 = \big((-\infty, g_2] \cup [g_1, +\infty)\big) \cap (g_4, g_3)$$

Note que $g_3 = g_1 - 1$ e $g_4 = g_2 - 1$.

Fazendo a intersecção de S_1 e S_2 temos



Portanto $S_1 \cap S_2 = (g_4, g_2]$. Como $g \in (g_4, g_2]$, $g_4 = g_2 - 1$, e como g é inteiro, seque que $g = \lfloor g_2 \rfloor$.

Para $id = \ell$, temos assim:

$$g = \lfloor \frac{(2n+1) - \sqrt{(4n^2 + 4n + 1 - 8\ell)}}{2} \rfloor.$$

Depois que obtivemos g podemos obter facilmente h. Se $\ell = p$, $h = g$. Mais em geral, para um ℓ qualquer (ver Figura 5.4), temos:

$$h = g + \ell - p.$$

Sendo

$$p = \frac{(2n - g + 1)g}{2}$$

temos:

$$h = \ell - \frac{(2n - g + 1)g}{2} + g$$

$$h = \ell - \frac{2ng - g^2 + g - 2g}{2}$$

$$h = \ell - \frac{(2n-1)g - g^2}{2}$$

Assim, obtemos as fórmulas para associar um par (g, h) a cada *id* de uma *thread*.

A seguir, mostramos resultados experimentais para este problema.

Capítulo 6

Resultados obtidos

6.1 Resultados obtidos para o problema da subsequência máxima

Os experimentos para o problema da subsequência máxima foram realizados utilizando uma máquina composta por 2 placas gráficas Geforce GTX 295 [19] da NVIDIA. Cada placa gráfica contém 2 GPUs de *compute capability* 1.3, cada GPU contém 30 multiprocessadores, cada multiprocessador é composto por 8 *cores* totalizando 240 *cores* por GPU. Cada multiprocessador contém uma memória compartilhada de 16 Kbytes. Essa máquina também possui um processador i7 da Intel com velocidade de 2,67 GHz e memória RAM de 5 GB. Os resultados experimentais dessa pesquisa foram obtidos a partir da utilização de 1, 2, 3 e 4 GPUs. O maior *speedup* foi obtido com a utilização de 4 GPUs e com um total de 960 núcleos (*cores*).

O processamento da entrada de n números foi feito por 90 blocos e 128 *threads* por bloco. A escolha da quantidade de blocos lançados foi feita pelo fato de cada GPU possuir 30 SM, e assim, com 90 blocos cada SM processa 3 blocos. Após alguns testes observamos que com 90 blocos o processamento foi mais rápido do que com qualquer outra quantidade estipulada.

As tabelas a seguir (Tabelas 6.1, 6.2, 6.3, 6.4) apresentam tempos de execução do algoritmo sequencial para o problema da subsequência máxima usando apenas a CPU e o tempo de execução usando CPU e 1, 2, 3 e 4 GPUs. Para cada instância de entrada com n elementos, executamos o experimento 10 vezes e calculamos uma média dos tempos obtidos que será considerada como o tempo de execução. Essas tabelas também mostram o desvio padrão dos tempos de execução do programa sequencial e dos paralelos e os *speedups* obtidos.

n	Tempo sequencial	Desvio padrão	Tempo paralelo	Desvio padrão	Speedup
100.000	0,003	0,0004	0,050	0,0003	0,055
200.000	0,004	0,0004	0,072	0,005	0,066
500.000	0,010	0,0009	0,144	0,002	0,066
1.000.000	10,760	0,420	0,232	0,004	46,231
2.000.000	20,350	0,514	0,460	0,011	44,202
5.000.000	30,330	0,779	1,163	0,134	26,064
10.000.000	60,450	0,485	2,103	0,364	28,736
20.000.000	129,120	0,900	3,560	0,400	36,263
50.000.000	310,690	0,881	10,303	0,583	28,857
100.000.000	620,690	0,926	20,287	0,551	30,594

Tabela 6.1: Resultados obtidos: tempos em milissegundos - execução paralela com 1 GPU.

n	Tempo sequencial	Desvio padrão	Tempo paralelo	Desvio padrão	Speedup
100.000	0,003	0,0004	0,147	0,0006	0,018
200.000	0,004	0,0004	0,060	0,0002	0,079
500.000	0,010	0,0009	0,112	0,0002	0,086
1.000.000	10,760	0,420	0,158	0,001	68,100
2.000.000	20,350	0,514	0,231	0,010	88,055
5.000.000	30,330	0,779	0,535	0,011	56,659
10.000.000	60,450	0,485	1,148	0,266	52,658
20.000.000	129,120	0,900	2,029	0,314	63,631
50.000.000	310,690	0,881	5,330	0,306	58,200
100.000.000	620,690	0,926	10,369	0,537	59,837

Tabela 6.2: Resultados obtidos: tempos em milissegundos - execução paralela com 2 GPUs.

n	Tempo sequencial	Desvio padrão	Tempo paralelo	Desvio padrão	Speedup
100.000	0,003	0,0004	0,082	0,0004	0,033
200.000	0,004	0,0004	0,058	0,0003	0,081
500.000	0,010	0,0009	0,066	0,0005	0,145
1.000.000	10,760	0,420	0,090	0,002	112,094
2.000.000	20,350	0,514	0,150	0,010	134,865
5.000.000	30,330	0,779	0,386	0,012	78,983
10.000.000	60,450	0,485	0,760	0,017	79,530
20.000.000	129,120	0,900	1,612	0,254	80,099
50.000.000	310,690	0,881	3,286	0,470	94,413
100.000.000	620,690	0,926	6,479	0,508	95,759

Tabela 6.3: Resultados obtidos: tempos em milissegundos - execução paralela com 3 GPUs.

n	Tempo sequencial	Desvio padrão	Tempo paralelo	Desvio padrão	Speedup
100.000	0,003	0,0004	0,045	0,0008	0,062
200.000	0,004	0,0004	0,075	0,0002	0,063
500.000	0,010	0,0009	0,066	0,0002	0,145
1.000.000	10,760	0,420	0,092	0,0003	116,361
2.000.000	20,350	0,514	0,139	0,010	146,363
5.000.000	30,330	0,779	0,287	0,016	105,661
10.000.000	60,450	0,485	0,598	0,033	110,787
20.000.000	129,120	0,900	1,063	0,041	121,435
50.000.000	310,690	0,881	2,517	0,402	123,246
100.000.000	620,690	0,926	4,781	0,486	129,759

Tabela 6.4: Resultados obtidos: tempos em milissegundos - execução paralela com 4 GPUs.

Um aspecto a ser considerado é sobre os *speedups* abaixo de um para instâncias pequenas, como por exemplo, o *speedup* de uma instância n de 200.000 que foi de 0,066 processada com 1 GPU. O motivo para um *speedup* baixo foi o seguinte. Como a quantidade de dados é pequena, só foi possível lançar uma quantidade pequena de blocos e consequentemente não foi possível aproveitar todo o poder computacional da GPU. Em aplicações GPUs o *speedup* normalmente cresce até o ponto em que todos os núcleos da placa ficam ocupados e depois estabiliza em um patamar.

A seguir, serão apresentados os resultados de execução sequencial e paralela com o uso de 1, 2, 3 e 4 GPUs através de gráficos. A Figura 6.1 compara o tempo do programa sequencial com o paralelo utilizando 1, 2, 3 e 4 GPUs.

Para visualizar melhor os tempos de execução dos programas paralelos, a Figura 6.2 mostra os tempos de execução paralela com 1, 2, 3, e 4 GPUs.

A Figura 6.3 mostra os *speedups* obtidos. Observa-se que o melhor *speedup* é alcançado para $n = 2.000.000$.

Conforme já mencionado antes, o desafio aqui é que o algoritmo paralelo de Alves, Cáceres e Song [1], por apresentar comandos do tipo *if-then-else*, não se enquadra no modelo SIMD (*Single Instruction Multiple Data*) [44], que é o modelo da quase totalidade dos algoritmos implementados em GPUs. No modelo SIMD, comandos condicionais podem causar degradação no desempenho, pois parte dos processadores podem executar o ramo *then*, enquanto que outra parte podem executar o ramo *else*. Mesmo assim, os experimentos realizados mostram um ganho promissor, tendo alcançado um *speedup* de 146 no caso de $n = 2.000.000$ para 4 GPUs.

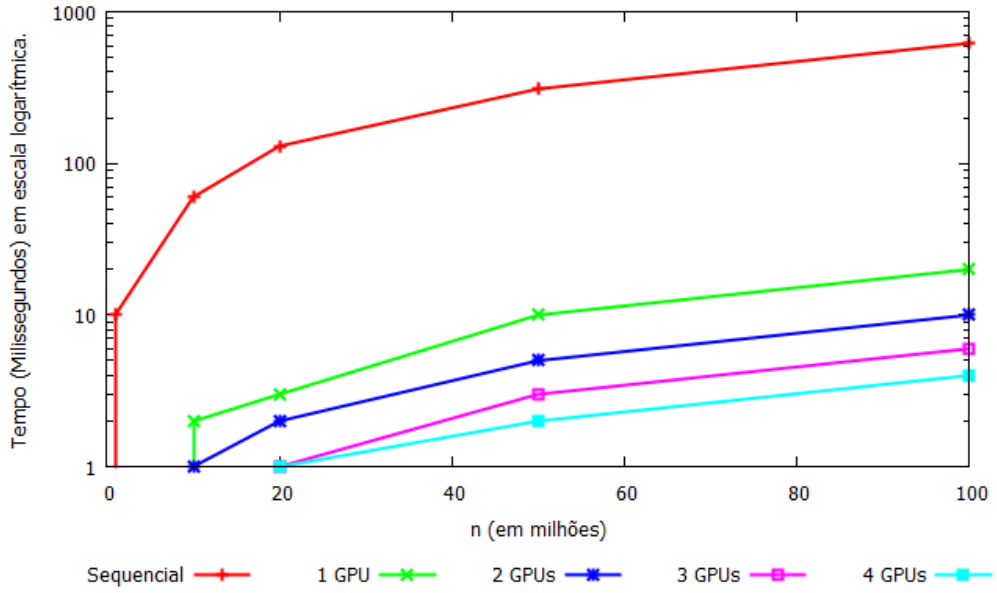


Figura 6.1: Tempo de execução sequencial e tempo de execução paralela com 1, 2, 3 e 4 GPUs.

6.2 Resultados obtidos para o problema da submatriz máxima

A implementação paralela do problema de submatriz máxima foi realizada em uma máquina composta por 2 placas gráficas Geforce GTX 680 [20] da NVIDIA. Cada placa gráfica contém uma GPU de *compute capability* 3.0, com 8 multiprocessadores (SMX) em cada uma delas e memória compartilhada de 48 Kbytes. Cada multiprocessador contém 192 *cores*, totalizando $8 \times 192 = 1.536$ *cores*. As duas placas juntas contêm portanto um total de 16 multiprocessadores totalizando 3.072 *cores*. Essa máquina também possui um processador i7 da Intel com velocidade de 3,20 GHz e memória RAM de 32 GB.

Os resultados experimentais que essa pesquisa apresenta foram obtidos a partir da utilização de 1 e 2 GPUs. O *speedup* maior foi obtido através da utilização de 2 GPUs.

Tabelas 6.5 e 6.6 apresentam os tempos do programa sequencial e do programa paralelo em segundos, para 1 e 2 GPUs, respectivamente. Para cada instância de matriz de $n \times n$ elementos, executamos 10 vezes e calculamos uma média que foi considerada como o tempo de execução. As tabelas também mostram o desvio padrão dos tempos de execução do programa sequencial e do paralelo e também mostra o *speedup* obtido.

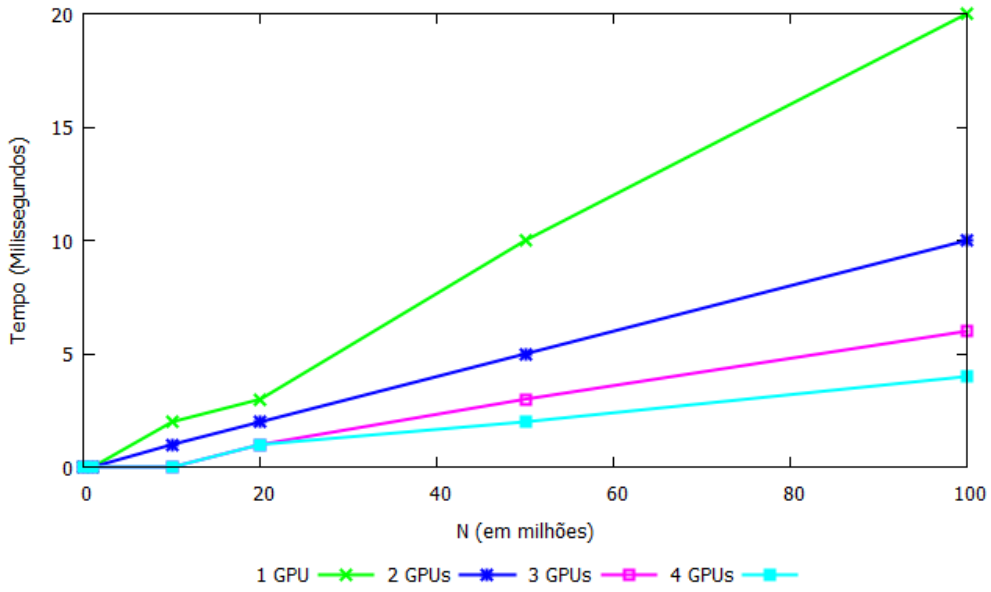


Figura 6.2: Tempo de execução paralela com 1, 2, 3 e 4 GPUs.

Para ilustrar melhor os resultados, dois gráficos são apresentados. A Figura 6.4 compara o tempo do programa sequencial com o tempo dos programas paralelos com 1 e 2 GPUs. A Figura 6.5 mostra os *speedups* obtidos com 1 e 2 GPUs.

O melhor *speedup* alcançado foi 584, para matriz $n \times n$ com $n = 15.360$, com 2 GPUs.

Todos os experimentos acima foram realizados com o uso da memória compartilhada. Diferente da implementação do algoritmo da subsequência máxima, na implementação do algoritmo da submatriz máxima foi possível obter um *speedup* significativo utilizando a memória global. Entretanto, nós empenhamos para realizar uma implementação utilizando a memória compartilhada a fim de obter resultados ainda melhores. Com isso, nos deparamos com as mesmas dificuldades que foram enfrentadas na implementação do algoritmo da subsequência máxima, ou seja, o tamanho reduzido da memória compartilhada. Para suprir essas dificuldades, foi utilizada a mesma estratégia de distribuição de dados criada na implementação da subsequência máxima. Porém, aqui usamos a arquitetura *Kepler* [33], que possui uma memória compartilhada de 48 Kbytes [36]. Desse modo, foi possível alocar na memória compartilhada um vetor de inteiros, com tamanho de 12.288. Foi lançado um *kernel* com 1.024 *threads* e, desse modo, cada *thread* fica responsável por 12 elementos do vetor ($12 \times 1.024 = 12.228$ elementos). Na Figura 6.6, pode ser observada a diferença entre os tempos de execução da implementação com a memória compartilhada

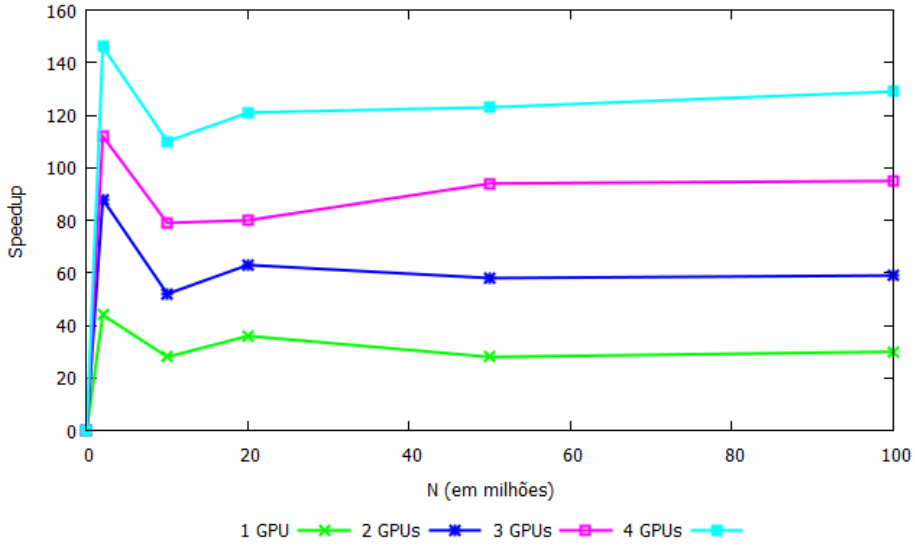


Figura 6.3: Gráfico dos speedups obtidos.

e da implementação com a memória global.

Cabe ainda uma observação. Como já foi mencionado anteriormente, a memória compartilha é mais rápida do que a memória global [23]. Com isso, ganhamos mais desempenho ao trazer os dados para serem processados na memória compartilhada, ao invés de processá-los na memória global. Porém, a capacidade máxima de armazenamento do vetor alocado na memória compartilhada é de apenas 12.288, sendo que a quantidade de dados a serem processados é superior a esta quantidade. Dessa forma, os dados são transferidos para a memória compartilhada de 12.288 em 12.288 elementos.

n	Tempo sequencial (segundos)	Desvio padrão	Tempo paralelo (segundos)	Desvio padrão	Speedup
512	0,538	0,683	0,006	0,006	80,226
1.024	5,781	1,145	0,047	0,424	121,990
2.048	74,595	1,627	0,346	1,224	215,287
4.096	664,498	2,220	2,820	1,547	235,965
6.144	2.479,325	2,562	10,790	1,955	229,797
8.192	5.898,160	2,784	22,776	2,461	259,036
10.240	11.729,870	3,037	46,150	2,855	254,190
12.288	20.951,880	3,267	85,410	2,910	245,306
14.336	34.295,050	3,539	120,644	2,981	284,261
15.360	47.625,508	3,769	162,447	3,112	292,366

Tabela 6.5: Resultados obtidos utilizando 1 GPU: tempos em segundos.

n	Tempo sequencial (segundos)	Desvio padrão	Tempo paralelo (segundos)	Desvio padrão	Speedup
512	0,538	0,683	0,003	0,004	150,155
1.024	5,781	1,145	0,025	0,153	230,084
2.048	74,595	1,627	0,175	0,431	424,950
4.096	664,498	2,220	1,409	1,018	471,524
6.144	2.479,325	2,562	5,987	1,401	414,0528
8.192	5.898,160	2,784	11,022	1,717	535,101
10.240	11.729,870	3,037	24,044	2,406	487,847
12.288	20.951,880	3,267	42,780	2,699	489,7517
14.336	34.295,050	3,539	60,377	2,849	568,035
15.360	47.625,508	3,769	81,500	3,177	584,355

Tabela 6.6: Resultados obtidos utilizando 2 GPUs: tempos em segundos.

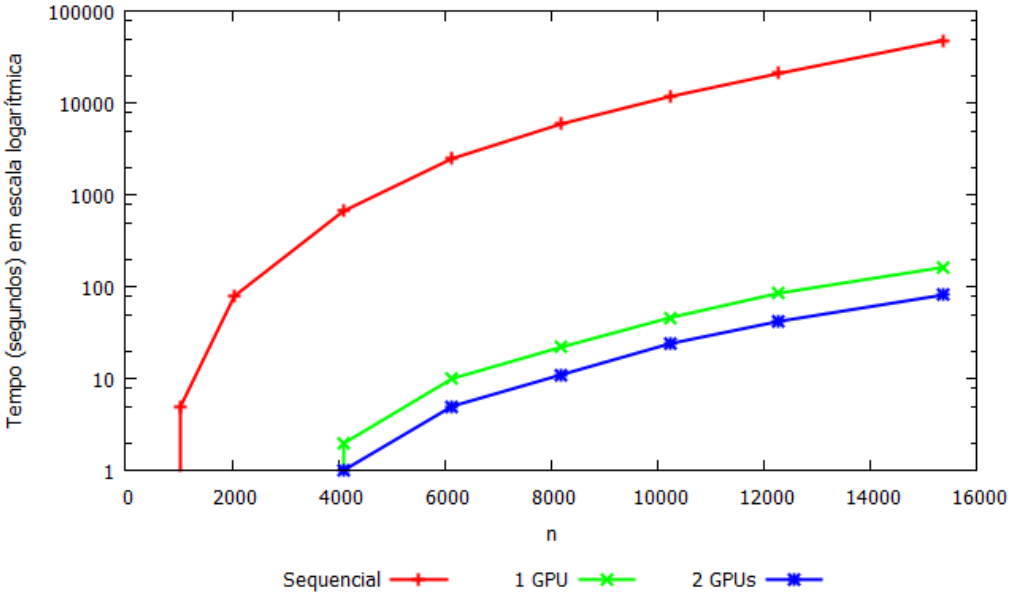


Figura 6.4: Tempo em segundos do programa sequencial e dos programas paralelos com 1 e 2 GPUs

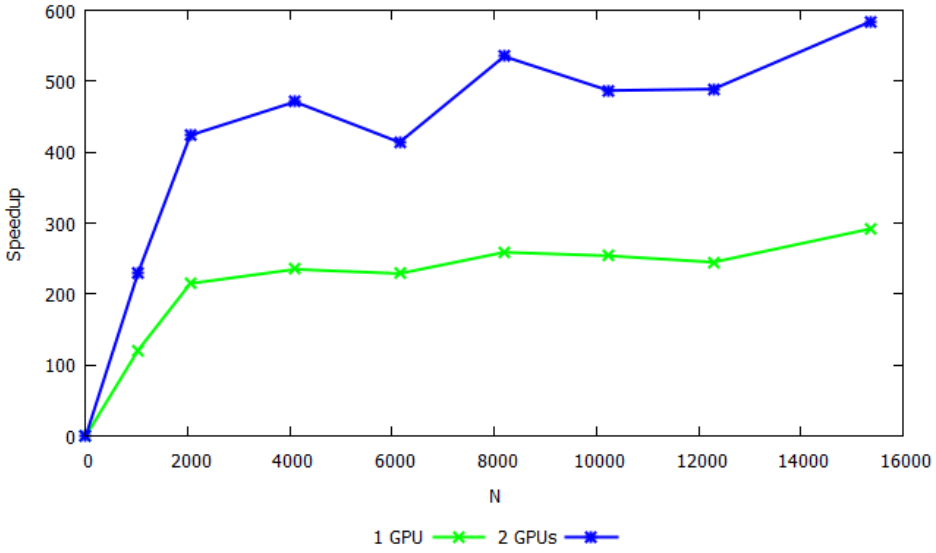


Figura 6.5: Gráfico dos speedups obtidos com 1 e 2 GPUs

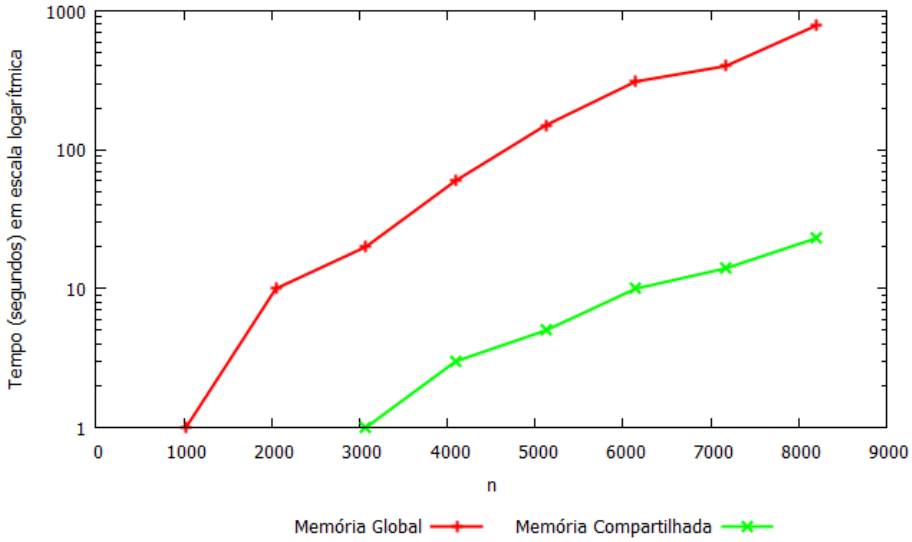


Figura 6.6: Tempos da implementação com a memória global e da implementação com a memória compartilhada

Capítulo 7

Conclusão

Atualmente, GPUs com centenas ou milhares de processadores ou *cores* apresentam um poder computacional muito grande, em comparação com a de uma CPU [25]. Pelo seu benefício-custo, há um grande interesse no seu uso para acelerar as computações de problemas que demandam grande poder computacional.

Neste trabalho, são apresentadas implementações paralelas em GPU de dois problemas, com aplicações em Biologia Computacional e Reconhecimento de Padrões. A primeira implementação paralela lida com o problema da subsequência máxima. Nosso trabalho se baseia no algoritmo paralelo de Alves, Cáceres e Song [1]. A segunda lida com o problema da submatriz máxima, usando como base os algoritmos paralelos de Perumalla e Deo [43] e de Alves, Cáceres e Song [3].

Os desafios encontrados no presente trabalho foram vários. Primeiro, tivemos que elaborar uma estratégia de distribuição de dados eficiente entre os diversos níveis de memória existentes numa arquitetura GPU. Detalhes foram apresentados nos capítulos anteriores, com uma comparação mostrada na Figura 6.6 que ilustra bem a sua importância.

O outro desafio, mais acentuado na implementação do algoritmo da subsequência máxima, é explicado a seguir. Conforme já mencionado antes, o desafio aqui é que o algoritmo paralelo de Alves, Cáceres e Song [1], por apresentar comandos do tipo *if-then-else*, não se enquadra no modelo SIMD (*Single Instruction Multiple Data*) [44], que é o modelo da quase totalidade dos algoritmos implementados em GPUs. Em uma GPU, comandos condicionais podem causar degradação no desempenho, pois parte dos processadores podem executar o ramo *then*, enquanto que outra parte pode executar o ramo *else*. Mesmo assim, os experimentos realizados para resolver o problema da subsequência máxima mostram um ganho promissor, com um *speedup* de 146 no caso de $n = 2.000.000$ para 4 GPUs.

Isso pode ser explicado pela quantidade grande de *streaming multiprocessors* (SM) da

máquina utilizada (GTX 295), cada uma com uma pequena quantidade de processadores. Ela apresenta 4 GPUs com um total de 120 SMs, cada SM contendo apenas 8 processadores. Ao executar um comando *if-then-else*, os processadores de um SM podem degradar o desempenho do SM, pois alguns podem executar o ramo *then* e outros o ramo *else*. Como cada SM é independente do outro, cada qual possuindo o seu próprio controlador de fluxo de instruções, o conjunto de 120 SMs acabam produzindo um bom desempenho globalmente.

Já na implementação em GPU do algoritmo paralelo da submatriz máxima, observa-se que a maior parte das computações envolvidas seguem o modelo SIMD: cálculo das somas de prefixos, determinação das submatrizes $K_{[0,n-1][g,h]}$, e dos vetores coluna $C^{g,h}$. Apenas a parte final, onde é feito o cálculo das subsequências máximas de $C^{g,h}$, não se enquadra no modelo SIMD. Isso explica o resultado superior (com *speedup* de 584, para $n = 15.360$, com 2 GPUs) na implementação paralela do algoritmo da submatriz máxima.

As contribuições do trabalho são:

- Adaptar os algoritmos da subsequência máxima e da submatriz máxima para implementação em GPU, explorando as característica da arquitetura GPU.
- Implementar os algoritmos explorando o uso da memória global e da memória compartilhada.
- Obter fórmulas fechadas que associam a cada *thread* um par de índices (g, h), que delimitam as colunas de uma submatriz de sua responsabilidade. Com isso, podem ser disparadas computações paralelas e independentes de todas as *threads*.

Como trabalhos futuros, podemos citar os seguintes.

- Realizar mais experimentos para analisar o efeito causado por comandos condicionais no desempenho dos algoritmos. Uma forma é executar o algoritmo com entradas de dois tipos: Uma que causa maior degradação em desempenho (parte dos *cores* executam o ramo *then* ou outra parte o ramo *else*) a outra que não sofre degradação (todos os *cores* seguem o mesmo ramo).
- Implementar em GPU o algoritmo que obtém todas as subsequências maximais, conforme proposto por Alves, Cáceres e Song [4]. Esse algoritmo é mais complexo e pode requerer um esforço não trivial para a sua implementação.
- Estender o algoritmo paralelo da submatriz máxima para obter todas as submatrizes maximais de uma matriz dada. Este problema encontra-se em aberto.

Apêndice A

Código - Subsequência máxima em GPU

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

#define N 5000000
#define BLOCK_SIZE  90
#define nThreadsPerBlock 128

// Vetor que armazena os cinco valores de cada thread.
#define NFinal (BLOCK_SIZE * nThreadsPerBlock) * 5

// kernel que transfere os dados para a memória compartilhada.
__device__ int* memoria(int *vetDados, int ElemPorBlocos, int qtdProces){

  __shared__ int vetComp[4076];

  //  auxGrupoDe32 diz quantos grupos de 32 dados foram processados
  int auxGrupoDe32 = (qtdProces * 32);

  // comecoBloco diz a onde cada bloco irá começar a trabalhar
  int comecoBloco = blockIdx.x * ElemPorBlocos;

  // qtdElemThread diz quantos elementos da thread ira processar
  int qtdElemThread = ElemPorBlocos / blockDim.x;

  // idCompartilhada diz a onde cada thread irá armazenar os dados na
     memória compartilhada
  int idCompartilhada = threadIdx.x;

  // idGlobal diz a onde cada thread ira buscar os dados na memória global
```

```
29   int idGlobal = comecoBloco + ((threadIdx.x / 32) * qtdElemThread) + (
         threadIdx.x - ((threadIdx.x / 32) * 32)) + auxGrupoDe32;

31   int i;
     for(i = 0; i < 4076; i += blockDim.x){
33   // transferência dos dados para a memória compartilhada.
       vetComp[idCompartilhada] = vetDados[idGlobal];
35     idCompartilhada += blockDim.x;
       idGlobal += (qtdElemThread * 4);
37   }

39   return vetComp;
}
41
// kernel que executa todo o processamento
43 __global__ void subSeqMax(int *vet, int *vetFinal, int ElemPorThread, int n
     ){

45   // ponteiro para apontar para a memória compartilhada
     __shared__ int *p;
47
     //      M          t_m        S       suf
49   int ini_M, fim_M, t_M, ini_S, fim_S, suf;
     t_M = suf = 0;
51
     // comecoThread  diz a onde cada thread irá armazenar os dados na memória
         compartilhada
53   int comecoThread = (threadIdx.x * 32);

55    // cada bloco executa esse laço para processar o seu intervalo I

57   int j;
     for(j = 0; j < (n / 4076); j++){
59
       p = memoria(vet,n,j);
61
       __syncthreads();
63
         if(threadIdx.x < 128){
65
           int i;
67         for(i = comecoThread -1; i < comecoThread + 32; i++){
             if(i == fim_M){
69               fim_S++;
                 suf += p[i+1];
71
                 if(suf < 0){
```

```
73                  suf = 0;
                    fim_S = -1;
75              }

77          ini_S = fim_S == 0 ? 0 : ini_S; // Inicio S

79              if(p[i+1] > 0){
                    fim_M++;
81                  t_M += p[i+1];
                    ini_M = fim_M == 0 ? 0 : ini_M; // Inicio M
83              }
            }
85          else{
                if(suf + p[i+1] > t_M){
87                  fim_S++;
                    if(ini_M == -1){
89                   fim_S = ini_S = i +1;
                    }
91
                    suf += p[i+1];
93                  ini_M = ini_S;
                    fim_M = fim_S;
95                  t_M = suf;
                }
97              else{
                        if(suf + p[i+1] > 0){
99                          fim_S++;
                            if(suf == 0){
101                             ini_S = fim_S = i+1;
                            }
103
                            suf += p[i+1];
105                     }
                        else{
107                         ini_S = fim_S = i + 2;
                            suf = 0;
109                     }
                 }
111         }
        }
113   }
  }
115
  if(threadIdx.x < 128){
117   int idThread = blockIdx.x * blockDim.x + threadIdx.x;

119   // vetFinal é o vetor que irá armazenar os 5 valores de cada thread
```

```
    vetFinal[(idThread * 5)] =  vetFinal[(idThread * 5)+1] = vetFinal[(
        idThread * 5)+2] = vetFinal[(idThread * 5)+3] =
121 vetFinal[(idThread * 5)+4] = 0;

123 // colocando o t_M em vetFinal
    vetFinal[(idThread * 5)+2] = t_M;
125
    // calculando o Prefixo
127 int pref_Max, soma_Pref;
    soma_Pref  = 0;
129 pref_Max = 0;

131 int i;
    if(ini_M > ((ElemPorThread*idThread))){
133     for(i = 0; i < ini_M; i++){
            soma_Pref += vet[i];
135
            if(soma_Pref > pref_Max){
137             pref_Max = soma_Pref;
            }
139     }

141     if(pref_Max <= 0){
            // pref_Max <= 0 então só temos números negativos antes de M
143     vetFinal[(idThread * 5)] = 0;
        vetFinal[(idThread * 5)+1] = soma_Pref;
145     }
        else{
147     // colocando o t_P em vetFinal
        vetFinal[(idThread * 5)] = pref_Max;
149     // colocando o t_N1 em vetFinal
        vetFinal[(idThread * 5)+1] = soma_Pref - pref_Max;
151     }
    }
153
    // calculo do sufixo
155 int suf_Max, soma_Suf;
    soma_Suf = suf_Max = 0;
157
    if(fim_M < ((ElemPorThread*idThread)+ElemPorThread)){
159     for(i = ((ElemPorThread*idThread)+ElemPorThread); i > fim_M; i--){
            soma_Suf += vet[i];
161
            if(soma_Suf > suf_Max){
163             suf_Max = soma_Suf;
            }
165     }
```

```
167         if(suf_Max <= 0){
            // colocando o t_S em vetFInal.
169             vetFinal[(idThread * 5)+3] = 0;
            // colocando o t_N2 em vetFinal
171         vetFinal[(idThread * 5)+4] = suf_Max;

173         }
            else{
175           // colocando o t_S em vetFinal
                  vetFinal[(idThread * 5)+3] = suf_Max;
177            // colocando o t_N2 em vetFinal
              vetFinal[(idThread * 5)+4] = suf_Max - soma_Suf;
179         }
      }
181   }
}
183
// método que executa o algoritmo de Bates e Constable sequencialmente
185 void subSeqMaxFinal(int *vet, int n){

187     //      M         t_m       S       suf
      int ini_M, fim_M, t_M, ini_S, fim_S, suf;
189     ini_M = fim_M = ini_S = fim_S = -1;

191     t_M = suf = 0;

193   int i;
      for(i = -1; i < n-1; i++){
195         if(i == fim_M){
              fim_S++;
197           suf += vet[i+1];

199           if(suf < 0){
                suf = 0;
201             fim_S = -1;
              }
203
              ini_S = fim_S == 0 ? 0 : ini_S; // Inicio S
205
                if(vet[i+1] > 0){
207               fim_M++;
                  t_M += vet[i+1];
209               ini_M = fim_M == 0 ? 0 : ini_M; // Inicio M
                }
211         }
            else{
```

```
213             if(suf + vet[i+1] > t_M){
                    fim_S++;
215                 if(ini_M == -1){
                     fim_S = ini_S = i +1;
217                 }

219                 suf += vet[i+1];
                    ini_M = ini_S;
221                 fim_M = fim_S;
                    t_M = suf;
223
                }
225             else{
                        if(suf + vet[i+1] > 0){
227                         fim_S++;
                            if(suf == 0){
229                             ini_S = fim_S = i+1;
                            }
231
                            suf += vet[i+1];
233
                        }
235                     else{
                            ini_S = fim_S = i + 2;
237                         suf = 0;
                        }
239             }
        }
241 }

243 printf(" \n\n A sub Sequencia deu %d  \n\n", t_M);
}
245
int main(){
247
  int *vet_d; int *vetFinal_d;
249
  // vetor de dados
251 int *vet_h = (int *) malloc(sizeof(int) * N);
  // vetor final que armazenará os cinco valores de cada thread
253 int *vetFinal_h = (int *) malloc(sizeof(int) * NFinal);

255 // Preenchimento dos dados
  int i;
257 for(i = 0; i < N; i++){
      vet_h[i] = (rand() / ((double)RAND_MAX + 1) * 2000) - 1000;
259 }
```

```
261   for(i = 0; i < NFinal; i++){
          vetFinal_h[i] = 0;
263   }

265   cudaMalloc((void**)&vet_d, N * sizeof(int)); //Vetor de dados
      cudaMalloc((void**)&vetFinal_d, NFinal * sizeof(int));// Vetor Final
267
      cudaMemcpy(vet_d, vet_h, N * sizeof(int), cudaMemcpyHostToDevice);
269
      // dividindo os dados entre os blocos
271   int ElemPorBlocos = (N / BLOCK_SIZE);
      // dividindo os dados entre as threads
273   int ElemPorThread = (ElemPorBlocos / nThreadsPerBlock);

275   // kernel que executa o processamento na GPU
      subSeqMax<<<BLOCK_SIZE, nThreadsPerBlock>>>(vet_d, vetFinal_d,
          ElemPorThread,N / BLOCK_SIZE);
277
      // copiando o vetor vetFinal que contém os 5 valores de cada thread
279   cudaMemcpy(vetFinal_h, vetFinal_d, NFinal * sizeof(int),
          cudaMemcpyDeviceToHost); //Resposta Final

281     subSeqMaxFinal(vetFinal_h, NFinal);

283     cudaFree(vetFinal_d);
        cudaFree(vet_d);
285
  return 0;
287 }
```

SubSeqMax6.15.c

Apêndice B

Código - Submatriz máxima em GPU

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

#define N_COL 3072
#define N_LIN 3072

#define BLOCK_SIZE_SOMA_PREF 3
#define N_THREAD_PER_BLOCK_SOMA_PREF 1024

#define BLOCK_SIZE_CALC 32
#define N_THREAD_PER_BLOCK_CALC 1024

// Kernel que executa a soma de prefixos
__global__ void somaPrex(int* matriz, int nColuna, int nLin){

    int idThread = blockIdx.x * blockDim.x + threadIdx.x;

    // Dividindo a quantidade de linhas entre as threads
    int qtdColSoma = (nLin / (blockDim.x * gridDim.x));

    // comecDeTrabThr diz onde cada thread irá começa a trabalhar
    int comecDeTrabThr = ((qtdColSoma * nColuna) * idThread);

    // Executando a soma de prefixos
    int i,j;
    for(j=0; j<qtdColSoma;j++){
      for(i=1; i<nColuna; i++){
        matriz[(comecDeTrabThr + i) + (j*nColuna)] += matriz[(j*nColuna) + (
            comecDeTrabThr + i) - 1];
```

```
      }
    }
}

// Kernel que executa o processamento na GPU
__global__ void calc(int* matriz, int nCol, int nLin, int qtdDeParesGH,
    int *Subseqs){

  // Calculando os identificadores de cada thread
  long int idThread = blockIdx.x * blockDim.x + threadIdx.x;
  int tidAux        = blockIdx.x * blockDim.x + threadIdx.x;

  __shared__ int vetComp[12288];

  int i,j,k,l,auxComp,comecoVetThreadComp;

  //      M          t_m        S       suf
  int ini_M, fim_M, t_M, ini_S, fim_S, suf;
  ini_M = fim_M = ini_S = fim_S = -1;

  // comecoVetThreadComp diz a onde cada thread irá escrever no vetor
     compartilhado
  comecoVetThreadComp = (threadIdx.x * 12);

  auxComp = t_M = suf = 0;

  // Laço que processa a quantidades de pares (g,h)
  for(l=0; l< ((qtdDeParesGH/(blockDim.x * gridDim.x))+1); l++){

     // Calculando de g
     float delta = ((4*(nCol*nCol)) + (4*nCol) + 1 - (8 * idThread));
     float auxG = ((((2*nCol) +1) - sqrt(delta))/2);

     // Arredondando g para baixo
     auxG = floor(auxG);

     long int g = (long int) auxG;

    // Calculando h
          long int h= idThread - (((((2*nCol)-1)*g) - (g* g))/2);

    // Processando as colunas da matriz
     if(g>=0 && g<=nCol && h>=0 && h<=nCol && idThread < qtdDeParesGH){
       if(g==0){
         for(j=0; j < nLin/12; j++){// Processando em 12 em 12
           for(k=0;k<12;k++){
```

```
            vetComp[(threadIdx.x * 12)+k] = matriz[(h + (nCol * (k+(j*12))
                ))];
75      }

77      // Aplicando o algoritmo de Bates e Constable nos 12 elementos
        for(i = (comecoVetThreadComp -1); i < (comecoVetThreadComp + 12)
            -1; i++){
79        if(i == fim_M){
            fim_S++;
81          suf += vetComp[i+1];

83          if(suf < 0){
              suf = 0;
85            fim_S = -1;
            }
87
            ini_S = fim_S == 0 ? 0 : ini_S; // Inicio S
89
              if(vetComp[i+1] > 0){
91              fim_M++;
                t_M += vetComp[i+1];
93              ini_M = fim_M == 0 ? 0 : ini_M; // Inicio M
              }
95        }
          else{
97          if(suf + vetComp[i+1] > t_M){
              fim_S++;
99            if(ini_M == -1){
               fim_S = ini_S = i +1;
101           }

103           suf += vetComp[i+1];
              ini_M = ini_S;
105           fim_M = fim_S;
              t_M = suf;
107         }
            else{
109               if(suf + vetComp[i+1] > 0){
                      fim_S++;
111                   if(suf == 0){
                          ini_S = fim_S = i+1;
113                   }

115                   suf += vetComp[i+1];
                  }
117               else{
                      ini_S = fim_S = i + 2;
```

```
119                         suf = 0;
                        }
121                 }
                }
123             }
               }
125
                // Escrevendo o t_M no vetor resultado
127            if(t_M > auxComp){
                    Subseqs[tidAux] = t_M;
129                auxComp = t_M;
                }
131
               idThread += (blockDim.x * gridDim.x);
133           }
              else{
135             for(j=0; j < (nLin/12); j++){ // Processando em 12 em 12
                  for(k=0;k<12;k++){
137                 vetComp[(threadIdx.x * 12)+k]  = matriz[(h + (nLin * (k+(j*12)
                        ))))] - matriz[((g-1) + (nLin * (k+(j*12))))];
                }
139
                // Aplicando o algoritmo de Bates e Constable nos 12 elementos
141             for(i = (comecoVetThreadComp -1); i < (comecoVetThreadComp + 12)
                    -1; i++){
                  if(i == fim_M){
143                 fim_S++;
                    suf += vetComp[i+1];
145
                    if(suf < 0){
147                   suf = 0;
                      fim_S = -1;
149                 }

151                 ini_S = fim_S == 0 ? 0 : ini_S; // Inicio S

153                   if(vetComp[i+1] > 0){
                        fim_M++;
155                     t_M += vetComp[i+1];
                        ini_M = fim_M == 0 ? 0 : ini_M; // Inicio M
157                   }
                }
159             else{
                    if(suf + vetComp[i+1] > t_M){
161                   fim_S++;
                      if(ini_M == -1){
163                     fim_S = ini_S = i +1;
```

```
                    }
165
                    suf += vetComp[i+1];
167                 ini_M = ini_S;
                    fim_M = fim_S;
169                 t_M = suf;
                  }
171               else{
                      if(suf + vetComp[i+1] > 0){
173                       fim_S++;
                          if(suf == 0){
175                           ini_S = fim_S = i+1;
                          }
177
                          suf += vetComp[i+1];
179                   }
                      else{
181                       ini_S = fim_S = i + 2;
                          suf = 0;
183                   }
                  }
185             }
              }
187            }

189           // Escrevendo o t_M no vetor resultado
              if(t_M > auxComp){
191               Subseqs[tidAux] = t_M;
                 auxComp = t_M;
193             }

195           idThread += (blockDim.x * gridDim.x);
        }
197    }
  }
199}

201int main(){

203   // Calculando a quantidade de pares (g,h)
    int i; int qtdDeParesGH = ((N_COL*(N_COL+1)) / 2);
205
    // Alocando a matriz no host
207 int *matriz_h = (int *)malloc(sizeof(int *) * (N_COL*N_LIN));
    int *subSeq_h = (int *)malloc(sizeof(int *) * (BLOCK_SIZE_CALC*
       N_THREAD_PER_BLOCK_CALC));
209 int *matriz_d; int *subSeq_d;
```

```
211     // Preenchendo a matriz no host
        for(i=0; i<(N_COL*N_LIN); i++){
213         matriz_h[i] = (rand() / ((double)RANDMAX + 1) * 2000) - 1000;
        }
215
        for(i=0; i<(BLOCK_SIZE_CALC*N_THREAD_PER_BLOCK_CALC); i++){
217         subSeq_h[i] = 0;
        }
219
        // Reservando espaco na GPU
221     cudaMalloc((void**)&matriz_d, (N_COL*N_LIN) * sizeof(int));
        cudaMalloc((void**)&subSeq_d, (BLOCK_SIZE_CALC*N_THREAD_PER_BLOCK_CALC)
            * sizeof(int));
223
        // Copiando o vetor para a GPU
225     cudaMemcpy(matriz_d, matriz_h, (N_COL*N_LIN) * sizeof(int),
            cudaMemcpyHostToDevice);

227     // Lançando o kernel para realizar a soma de prefixos
        somaPrex<<<BLOCK_SIZE_SOMA_PREF, N_THREAD_PER_BLOCK_SOMA_PREF>>>(
            matriz_d,N_COL,N_LIN);
229
        cudaThreadSynchronize();
231
        // Lançando o kernel que executa o processamento na GPU
233     calc<<<BLOCK_SIZE_CALC, N_THREAD_PER_BLOCK_CALC>>>(matriz_d,N_COL,N_LIN,
            qtdDeParesGH,subSeq_d);

235     // Copiando os t_M para a CPU
        cudaMemcpy(subSeq_h, subSeq_d, (BLOCK_SIZE_CALC*N_THREAD_PER_BLOCK_CALC)
            * sizeof(int), cudaMemcpyDeviceToHost);
237
        // Encontrando a maior subSeq
239     int maiorSubSeq = subSeq_h[0];
        for(i=0; i < (BLOCK_SIZE_CALC*N_THREAD_PER_BLOCK_CALC); i++){
241       maiorSubSeq = subSeq_h[i] > maiorSubSeq ? subSeq_h[i]:maiorSubSeq;
        }
243
        cudaFree(matriz_d);
245     cudaFree(subSeq_d);

247    printf("\n Maior SubSequencia encontrada \n\n %d \n\n",maiorSubSeq);

249 return 0;
}
```

SubMatrizMax5.21.c

Referências Bibliográficas

[1] Alves, C. E. R., Cáceres, E. N., and Song, S. W. Computing Maximum Subsequence in Parallel. *II Brazilian Workshop on Bioinformatics - WOB 2003*, Macaé, RJ, Dec. 3-5, 2003, pp. 80-87.

[2] Alves, C. E. R., Cáceres, E. N. and Song, S. W. A BSP/CGM Algorithm for Finding All Maximal Contiguous Subsequences of a Sequence of Numbers. *Lecture Notes in Computer Science*, Vol. 4128, W. E. Nagel et al. (editors), Springer-Verlag. Dresden, Germany. Aug 29 to Sep 1, 2006, pp. 831-840.

[3] Alves, C. E. R., Cáceres, E. N. and Song, S. W. BSP/CGM Algorithms for Maximum Subsequence and Maximum Subarray. *Lecture Notes in Computer Science*, Vol. 3241, J. Dongarra, P. Kacsuk and D. Kranzlmüller (eds.), Springer-Verlag. Budapest, Hungary, Sep 19-22, 2004, pp. 139-146.

[4] Alves, C. E. R., Cáceres, E. N. and Song, S. W. Finding All Maximal Contiguous Subsequences of a Sequence of Numbers in $O(1)$ Communication Rounds. *IEEE Transactions on Parallel and Distributed Systems*, IEEE Computer Society, Vol. 24, No. 3, 2013, pp. 724-733.

[5] Appel, R. D., Bairoch, A. and Hochstrasser, D. F. A new generation of information retrieval tools for biologists: the example of the ExPASy www server. *Trends Biochem. Sci.* 19:258-260, 1994

[6] Bates, J. L., and Constable, R. L. Proofs as Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, 1985, pp. 113-136.

[7] Baxevanis, Andy and Ouellette, Francis. *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*. Second Edition, Wiley Interscience, 2001.

[8] Bentley, J. L. Programming pearls: algorithm design techniques. *Communications of the ACM*, 27(9):865-873, 1984.

[9] Boer, Dirk Vanden. General-Purpose Computing on GPUs. Technical report, *School of Information Technology Transnationale Universiteit Limburg Diepenbeek*, June 6, 2005

[10] Cirne, Marcos Vinícius Mussel. Introdução à Arquitetura de GPUs. Relatório técnico, *Instituto de Computação, Universidade de Campinas (Unicamp)*, 2007.

[11] Community GPU. Site: http://gpgpu.org
Acessado em 01/02/2012.

[12] Dehne F. Coarse Grained Parallel Algorithms. *Special Issue of Algorithmica*, 24(3/4):173-176, 1999.

[13] Dehne F., Fabri A., Scalable parallel geometric algorithms for coarse grained multicomputers. *In SCG '93: Proceedings of the ninth annual symposium on Computational geometry*, páginas 298-307. ACM, 1993. ISBN 0-89791-582-8.

[14] Dynamic Parallel in CUDA, 2012.
Manual de programação fornecido pela NVIDIA.
Disponível em: http://www.nvidia.com.

[15] Especificação do processador Xeon da Intel. Wikipedia.
http://pt.wikipedia.org/wiki/Xeon
Acessado em março de 2012.

[16] Ferraz, Samuel B. GPUs. Relatório técnico, *Faculdade de Computação, Universidade Federal de Mato Grosso do Sul*, 2009.

[17] Garcia T., Myoupo J., Seme D., Universite Picardie, e Jules Verne. A coarse-grained multicomputer algorithm for the longest common subsequence problem.*In 11-th Euromicro Conference on Parallel Distributed and Network based Processing (PDP'03, pag. 349-356. 2003.*

[18] Gebali, Fayez and El Miligi, Haytham. *Bioinformatics High Performance Parallel Computer Architectures*, Bertil Schmidt (ed.), CRC Press, 2011.

[19] Geforce GTX 295, NVIDIA, 2012.
http://www.nvidia.com.br/object/product_geforce_gtx_295_br.html
Acessado em 19/07/2012.

[20] Geforce GTX 680, NVIDIA, 2012.
http://www.nvidia.com.br/object/geforcegtx680br.html
Acessado em 03/01/2013.

[21] Gohner, Uli. Computing on GPUs. 7^{th} *European LS-DYNA Conference*, pp. 60-64, 2009.

[22] GTC 2012 Keynote (Part 03): The NVIDIA Kepler GPU Architecture, 2012.
Disponível em: http://www.youtube.com/watch?v=TxtZwW2Lf-w
Acessado em: 03/01/2013

[23] Hanrahan, P. and Buck, I. Data Parallel Computation on Graphics Hardware. Technical report, *Stanford University Computer Science Department.*

[24] Ikeda, Patricia Akemi. Um estudo do uso eficiente de programas em placas gráficas. Dissertação de mestrado, *Departamento de Ciência da Computação, Instituto de matemática e estatística, Universidade de São Paulo*, Agosto de 2011.

[25] Kirk, David B. and Hwu, W. *Programming massively parallel processors.* Morgan Kaufmann, 2010.

[26] Lopes, B. C. and Azevedo, R. J. Computação de alto desempenho utilizando CUDA. Relatório técnico, *Instituto de Computação, Universidade Estadual de Campinas (Unicamp)*, 2008.

[27] Morin, Pat. Coarse grained parallel computing on heterogeneous systems. *In SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing*, páginas 628-634. ACM, 1998. ISBN 0-89791-969-6.

[28] Nickolls, J. and Dally, W. J. The GPU Computing Era. *IEEE Micro, vol. 30, no. 2, pp. 56-69, March/April, 2010.*

[29] NVIDIA CUDA API Reference Manual - Version 4.0, 2012,
Manual de programação em CUDA fornecido pela NVIDIA Corporation.
Disponível em: http://www.nvidia.com.

[30] NVIDIA CUDA C Best Practices Guide - Version 4.0, 2012.
Manual de programação em CUDA fornecido pela NVIDIA Corporation.
Disponível em: http://www.nvidia.com.

[31] NVIDIA CUDA C Programming Guide - Version 2.8, 2010.
Manual de programação em CUDA fornecido pela NVIDIA Corporation.
Disponível em: http://www.nvidia.com.

[32] NVIDIA CUDA Homepage.
Disponível em: http://developer.nvidia.com/cuda.
Acessado em 01/03/2013

[33] NVIDIA especificação da arquitetura Kepler.
Homepage: http://www.nvidia.com.br/object/nvidia-kepler-br.html
Acessado em 03/01/2013

[34] NVIDIA Fermi Compatibility Guide For Cuda Applications, Maio 2011.
Manual da arquitetura Fermi fornecido pela NVIDIA Corporation.
Disponível em: http://www.nvidia.com.

[35] NVIDIA GPU Technical Specifications. Especificação das GPUs NVIDIA.
Disponível em: http://www.nvidia.com.

[36] NVIDIA Kepler GK110 Next-Generation CUDA Compute Architecture, 2012.
Manual da arquitetura Kepler fornecido pela NVIDIA Corporation.
Disponível em: http://www.nvidia.com.

[37] NVIDIA Next Generation CUDA Compute Architecture: Fermi, 2009. Manual da arquitetura Fermi fornecido pela NVIDIA Corporation.
Disponível em: http://www.nvidia.com.

[38] NVIDIA Next Generation CUDA Compute Architecture: Kepler GK110, 2012.
Manual da arquitetura Kepler fornecido pela NVIDIA Corporation.
Disponível em: http://www.nvidia.com.

[39] NVIDIA Tuning Cuda Applications For Fermi, Maio 2011, Manual da arquitetura Fermi fornecido pela NVIDIA Corporation.
Disponível em: http://www.nvidia.com.

[40] Owens, John D., Houston, Mike, Luebke, David, Green, Simon, Stohe, John E., and Phillips, James. GPU Computing, *Proceedings of the IEEE*, vol. 96, pages 879-899, May, 2008.

[41] Patterson, David. The Top 10 Innovations in the New NVIDIA Fermi Architecture, and The Top 3 Challenges.*Scientific American,* 150^{th} *Anniversary Edition, vol. 273, (no. 3): pp. 62-67*, September 2009.

[42] Pearson, W. R. and Lipman, D. J. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci.* USA 85:2444-2448, 1988.

[43] Perumalla, K. and Deo, N. Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Processing Letters*, 5(3):367-373, 1995.

[44] Roosta, Seyed H. *Parallel Processing and Parallel Algorithms*. Springer, 2000.

[45] Sanders, J. and Kandrot, E. *CUDA by Example: an Introduction to General-purpose GPU Programming. Addison-Wesley*, 2011. Disponível em: http://developer.nvidia.com/object/cuda-by-example.html

[46] JáJá, Joseph. *Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[47] Wynters, Erik. Parallel Processing on NVIDIA Graphics Processing Units Using CUDA. *Journal of Computing Sciences in Colleges, Volume 26 Issue 3, pp. 58-66, January 2011.*