Tese apresentada à Pró-Reitoria de Pós-Graduação e Pesquisa do Instituto Tecnológico de Aeronáutica, como parte dos requisitos para obtenção do título de Mestre em Ciências no Programa de Pós–Graduação em Engenharia Eletrônica e Computação, Área de Informática

Vitor Venceslau Curtis

ALGORITMOS PARA O PROBLEMA *SUBSET-SUM* EM GPU

Tese aprovada em versão final pelos abaixo assinados:

Prof. Dr. Carlos Alberto Alonso Sanches

Orientador

Prof. Dr. Celso Massaki Hirata

Pró-Reitor de Pós-Graduação e Pesquisa

Campo Montenegro

São José dos Campos, SP - Brasil

2013

Dados Internacionais de Catalogação-na-Publicação (CIP)

Divisão de Informação e Documentação

Curtis, Vitor Venceslau

Algoritmos para o problema *Subset-Sum* em GPU / Vitor Venceslau Curtis.
São José dos Campos, 2013.
101f.

Tese de Mestrado – Curso de Engenharia Eletrônica e Computação. Área de Informática – Instituto Tecnológico de Aeronáutica, 2013. Orientador: Prof. Dr. Carlos Alberto Alonso Sanches. .

1. GPU. 2. Subset-sum. 3. Knapsack. I. Centro Técnico Aeroespacial. Instituto Tecnológico de Aeronáutica. Divisão de Ciência da Computação. II. Algoritmos para o problema *Subset-Sum* em GPU

REFERÊNCIA BIBLIOGRÁFICA

CURTIS, Vitor Venceslau. **Algoritmos para o problema *Subset-Sum* em GPU**. 2013. 101f. Tese de Mestrado – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DO AUTOR: Vitor Venceslau Curtis

TITULO DO TRABALHO: Algoritmos para o problema *Subset-Sum* em GPU.

TIPO DO TRABALHO/ANO: Tese / 2013

Vitor Venceslau Curtis
Av. Paranaguá, 1061
CEP 03.806-010 – São Paulo–SP

ALGORITMOS PARA O PROBLEMA *SUBSET-SUM* EM GPU

Vitor Venceslau Curtis

Composição da Banca Examinadora:

Prof. Dr.	Nei Yoshihiro Soma	Presidente	-	ITA
Prof. Dr.	Carlos Alberto Alonso Sanches	Orientador	-	ITA
Prof. Dr.	Horácio Hideki Yanasse	Membro Externo	-	UNIFESP SJC
Prof. Dr.	Jairo Panetta	Membro	-	ITA

ITA

Dedico este trabalho a aqueles que me deram muito mais que a vida.

Agradecimentos

Gostaria de agradecer a todos os que me ajudaram neste trabalho e de me desculpar com aqueles que eu tenha injustamente deixado de fora desta breve lista.

Primeiramente, ao meu orientador Dr. Carlos Alberto Alonso Sanches, que praticamente definiu minha pesquisa, restringiu seus limites e analisou todo o trabalho com imensa paciência, o que deve ter-lhe custado muitas horas de lazer. Sem ele, ainda estaria procurando um tema.

Aos professores Dr. Stephan Stephany e Dr. Jairo Panetta, por ensinarem meus primeiros conceitos de computação paralela, alta performance e arquitetura de computadores, além de se colocarem sempre à disposição para sanar dúvidas e oferecer acesso a restritos parques de computação.

A todos os funcionários e professores com quem tive contato durante as aulas no ITA, por me ensinarem muito, incluindo questões morais desconhecidas para mim até então, fazendo-me ter orgulho de estar no Brasil.

Agradeço a um grande e eterno mestre, Aguinaldo Prandini Ricieri, por mostrar um novo mundo escondido atrás de meus olhos e por ser o verdadeiro motivo de eu estar no ITA, o que nunca imaginei em minha vida.

Não poderia deixar de agradecer também à minha família, que me ofereceu todo o incentivo e motivação de sempre.

Ao meu irmão, pelas ilustrações e por deixar de fazer seus trabalhos, para que eu pudesse utilizar sua máquina durante vários dias.

À minha noiva, que mais uma vez me ajudou de várias formas e também acabou se tornando uma especialista no assunto, além de me auxiliar com sugestões, que muitas

vezes me levaram a novas descobertas.

Finalmente, aos meus maravilhosos pais, que são pessoas extraordinárias e que fazem de tudo para que meus sonhos se realizem. Muito obrigado. Espero poder retribuir tudo o que me ensinaram, sendo sempre justo e amável com todos à minha volta, pois sei que é o maior presente que posso oferecer-lhes.

"Moore's Law is dead."
— Gordon Moore

Resumo

Este trabalho utiliza o problema *subset-sum* (SSP) como estudo de caso, com o objetivo de analisar a complexidade de paralelização em Unidades de Processamento Gráficas (GPU). O SSP foi escolhido por pertencer à classe dos problemas NP-Completo, possuir grande necessidade de memória e não ter cálculo de ponto flutuante, além de ser amplamente estudado na área acadêmica devido a sua importância prática e teórica. Estas características representam um desafio para paralelização em GPUs, pelo fato de serem especialistas em cálculos de ponto flutuante e por possuir pouca quantidade de memória em relação ao grande número de núcleos. Basicamente, são apresentados 3 novos algoritmos, implementados em linguagem CUDA C, com baixo consumo de memória: somente $O(n+m-w_{\min})$, onde $m = \min\{c, c-\sum_{i=1}^{n} w_i\}$, c é a capacidade da mochila, n é a quantidade de itens, w_i é o i-ésimo peso e $w_{\min}$ é o menor peso, ao invés de $O(nc)$ do paradigma de Bellman, referentes aos algoritmos do estado da arte implementados na mesma arquitetura. Esta característica permite um ganho significativo na quantidade de instâncias solucionáveis, além do melhor tempo computacional. Para uma variedade de *benchmarks*, obteve-se bons tempos de execução em comparação com os melhores resultados práticos conhecidos até agora. Isto foi possível graças a um novo método para a solução do SSP, permitindo sua computação em tempo $O(n)$ e mesmo espaço, caso c processadores sejam utilizados.

Abstract

This work uses the subset-sum (SSP) as a case study in order to analyze the complexity of parallelization on Graphic Processing Units (GPU). The SSP was chosen for belonging to the class of NP-Complete problems, have large memory requirements and not have floating-point calculation, besides being widely studied in academia because of its practical and theoretical importance. These characteristics represent a challenge for parallelization on GPUs, because they are experts in floating-point calculations and by having a small amount of memory in relation to the large number of cores. Basically, we present three new algorithms, implemented in CUDA C, with low memory usage: only $O(n+m-w_{\min})$*, where* $m = \min\{c, c - \sum_{i=1}^{n} w_i\}$*,* c *is the knapsack capacity,* n *is the number of items,* w_i *is the* i*-th weight and* $w_{\min}$ *is the lightest weight, instead of* $O(nc)$ *from Bellman's paradigm, related to the state of art algorithms implemented in the same architecture. This characteristic allows a significant gain in the number of solvable instances, besides a better computational time. For a variety of benchmarks, we have obtained good execution times compared to the best practical results so far known. This was possible due to a new method for the SSP solution, which allows the computation time* $O(n)$ *and same space, when* c *processors are used.*

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Nos últimos anos, a indústria de processadores reduziu o tamanho dos transístores ao ponto de chegar próximo ao seus limites físicos e de dissipação de calor, tornando não mais possível o aumento da frequência de processamento e a criação de *cores* com núcleos mais robustos, que até então tornavam a computação mais rápida sem a necessidade de alterações no *software* (HILL; MARTY, 2008)(DONGARRA, 2012). Como alternativa a este problema, a indústria optou por aumentar a quantidade de *cores* nos processadores, iniciando uma era de multi-processamento (SUTTER, 2005).

Neste cenário, para alguns problemas altamente paralelos, a computação com Unidades de Processamento Gráfico (GPUs) vem ganhando atenção por fornecer maior desempenho e menor consumo de energia que soluções baseadas apenas em CPUs *multicore*, sendo utilizadas em grandes centros de computação de alta performance, como é o caso do atual supercomputador mais rápido do mundo, Titan - Cray XK7, equipado com GPUs NVIDIA K20x (MEUER *et al.*, 2012)(DONGARRA, 2012).

Muitos trabalhos recentes exploram a paralelização em GPUs de algoritmos clássicos com o objetivo de obter o máximo desempenho possível deste novo paradigma (*e.g.* (BOYER; BAZ; ELKIHEL, 2012), (MERRILL; GARLAND; GRIMSHAW, 2012), (HONG *et al.*, 2011), (LIN *et al.*, 2010), (KAWANAMI; FUJIMOTO, 2012) e (SØRENSEN, 2012)), uma vez

que compõem a base de diversos *softwares*. Porém, nem todos os algoritmos podem ser adaptados a GPU, pois este tipo de implementação envolve alguns desafios não presentes na computação sequencial (*e.g.* condição de corrida, sincronização, menor disponibilidade de memória, padrão de acesso à memória e desvios condicionais), tornando uma solução nesta arquitetura frequentemente inviável.

O escopo deste trabalho limita-se à implementação em GPU de um caso específico do problema da mochila chamado *Subset-Sum Problem* (SSP), com o objetivo de obter desempenho superior a trabalhos correlacionados.

A motivação principal da escolha do SSP foi devido ao fato de ser um dos problemas NP-Completo mais estudados na Ciência da Computação (GAREY; JOHNSON, 1979), ter alta complexidade espacial (BOYER; BAZ; ELKIHEL, 2012) e não ser um problema que envolva muitos cálculos, situação em que as GPUs conseguem apresentar alto desempenho por possuírem uma grande quantidade de unidades lógicas dedicadas. Também vale a pena destacar que sua resolução pode ser aplicada a outros problemas clássicos: caixeiro viajante, satisfabilidade, fatoração e programação inteira, dentre outros, além possuir importantes resultados na área de complexidade teórica e ser a base de alguns modernos sistemas de criptografia (KATE; GOLDBERG, 2011).

Mais detalhes sobre o SSP e suas aplicações podem ser encontrados nas seguintes referências: (KATE; GOLDBERG, 2011), (SUN, 2003), (GEMMELL; JOHNSTON, 2001), (IRVINE; CLEARY; RINSMA-MELCHERT, 1995) e (BOHMAN, 1996).

1.1 Contribuições deste trabalho

As principais contribuições apresentadas neste trabalho podem ser resumidas nos itens:

- Três algoritmos para resolução do SSP em GPU com complexidade espacial de apenas $O(n + m - w_{\min})$, onde $m = \min\{c, c - \sum_{i=1}^{n} w_i\}$, n é a quantidade de itens do problema, c é a capacidade da mochila, w_i é o peso do item i e $w_{\min}$ é o menor peso, similar ao melhor resultado teórico conhecido (SANCHES; SOMA; YANASSE, 2008).

- Algumas otimizações que fazem o melhor algoritmo proposto, nos testes realizados, ter menor tempo de execução e suportar instâncias maiores em relação ao recente algoritmo de Boyer, Baz & Elkihel (2012).

- Realização de testes em uma maior variedade de *benchmarks*, em relação aos trabalhos de Boyer, Baz & Elkihel (2012) e Bokhari (2012).

- Elaboração de uma nova forma de recursão que generaliza os cálculos do algoritmo de Yanasse & Soma ((SOMA; YANASSE, 1987)), possibilitando sua computação paralela com tempo $O(n)$ e espaço $O(n + m - w_{\min})$, através de c processadores.

- Um novo *upperbound* de tempo da resolução paralela do SSP em uma PRAM SIMD: com $1 \leq p \leq \max\{n, c\}$ processadores, este novo algoritmo gasta tempo $O(nc/p)$, com espaço $O(n + m - w_{\min})$.

- Correção do método de sincronização entre blocos de Xiao & Feng (2010) e proposta de um novo método com melhor desempenho.

1.2 Organização

Os capítulos deste trabalho estão organizados como segue:

- **Capítulo 2**: apresenta a formalização do SSP e o estado da arte neste campo de pesquisa, descreve sumariamente a arquitetura CUDA e discorre sobre as formas de sincronismo entre *threads*.

- **Capítulo 3**: apresenta alguns dos principais algoritmos citados, a recursão utilizada nas implementações e alguns obstáculos na paralelização do SSP em GPU.

- **Capítulo 4**: apresenta as otimizações algorítmicas utilizadas nas implementações.

- **Capítulo 5**: apresenta as estratégias de paralelização utilizadas e suas respectivas implementações na arquitetura CUDA.

- **Capítulo 6**: apresenta e analisa os resultados de *benchmarks* dos algoritmos propostos, incluindo o novo método de sincronismo entre blocos.

- **Capítulo 7**: apresenta a conclusão deste trabalho e os principais resultados obtidos.

2 Preliminares

Neste capítulo é realizada uma breve apresentação da tecnologia CUDA, uma nova forma de sincronismo entre blocos é proposta, o SSP é descrito formalmente e o estado da arte referente a sua solução sequencial e paralela é apresentado.

2.1 A tecnologia CUDA

Em novembro de 2006, a empresa NVIDIA lançou a tecnologia *Compute Unified Device Architecture* (CUDA), presente em todas as suas recentes GPUs. Elas são compostas por uma série de *Multiprocessadores de Streaming* (SM) com arquitetura *Multiple Instruction, Multiple Data* (MIMD), cada um com um conjunto de processadores especiais (chamados *CUDA cores* ou simplesmente *cores*), cujas instruções são controladas por um ou mais gerenciadores de *threads*, resultando em uma particular arquitetura *Single Instruction, Multiple Data* (SIMD), chamada de *Single Instruction, Multiple Thread* (SIMT) (NVIDIA, 2012b).

A **Figura 2.1** apresenta o modelo Fermi, com 16 SMs de 32 *cores* (NVIDIA, 2009). Nesta versão, a GPU possui até seis bancos de memória de 64-bit em uma interface de 384-bit, suportando no total até 6 GB de memória GDDR5 DRAM, nomeada *memória global*, uma interface *host* para se conectar com a CPU através do barramento PCI-Express, *cache*

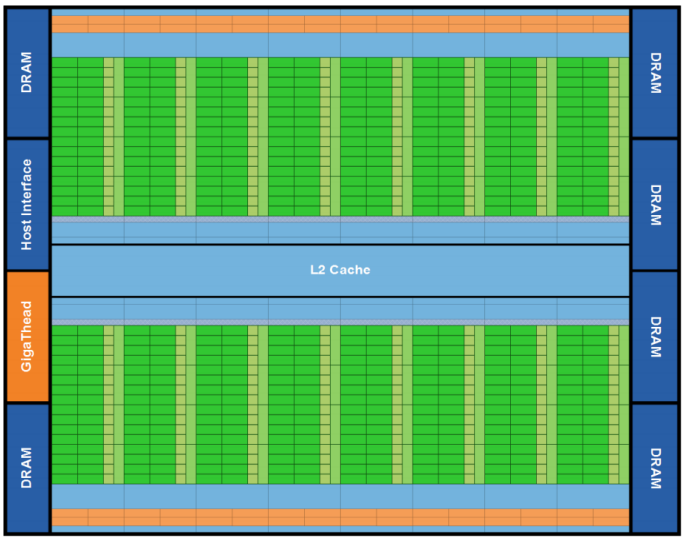

FIGURA 2.1 – Arquitetura CUDA com 16 SMs de 32 *cores*.

L2 (*off-chip*) compartilhado entre todos os SMs e o *GigaThread Scheduler*, que distribui os blocos de *threads* para as SMs.

A arquitetura de uma SM Fermi é apresentada na **Figura 2.2**, onde é possível observar que cada uma possui 16 unidades *load-store* (LD/ST) para operações de memória, 4 unidades de funções especiais (SFU), como funções trigonométricas, 64 KB de *cache* configurável entre memória compartilhada (*Shared Memory*) e *cache* L1 (*write-back* para L2), 8 KB de *cache* de memória constante (*Uniform Cache*), *cache* para o conjunto de instruções (*Instruction Cache*), 32K de registradores de 32-bit, além de lógicas para controle das *threads* e *cores* com capacidade de realizarem operações de ponto flutuante e inteiros.

2.1.1 O modelo de programação CUDA

Geralmente, o desenvolvimento de programas com a tecnologia CUDA pode ser feito através de bibliotecas, extensões às linguagens tradicionais (C, C++ e Fortran) ou diretivas OpenACC nas linguagens Fortran e C. Neste trabalho, foi escolhida a abordagem

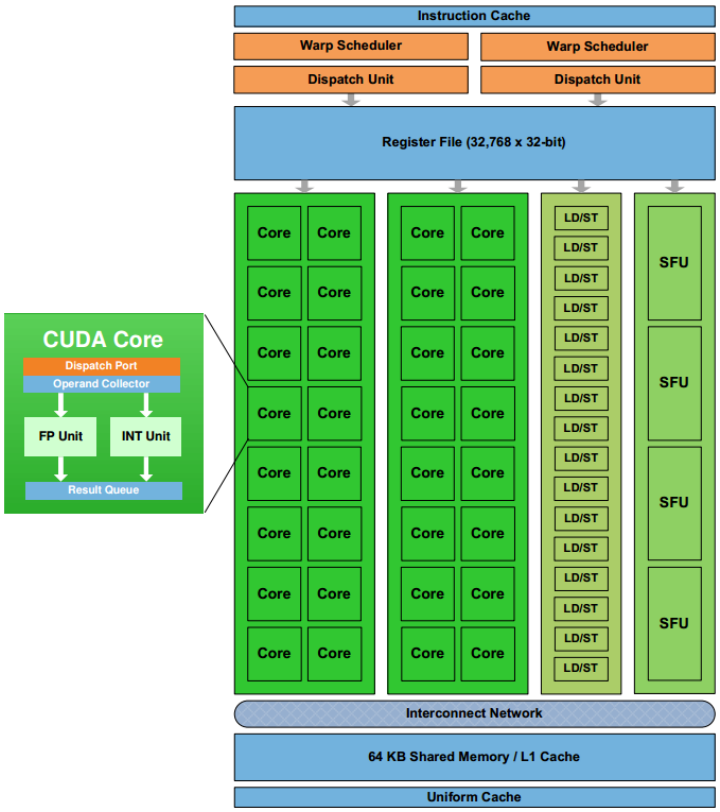

FIGURA 2.2 – Fermi SM.

através da expansão da linguagem C, por ser o método que oferece melhor desempenho (NVIDIA, 2012b).

Esta linguagem é fortemente baseada em C ANSI, possuindo apenas algumas extensões. Após o pré-processamento realizado pelo compilador NVCC, esses programas tornam-se compatíveis com qualquer compilador C.

Basicamente, o compilador NVCC simplesmente produz versões binárias ou código PTX das funções (que recebem o nome de *kernel*) do código-fonte escritas para serem executadas na GPU e substitui as chamadas a estas funções por chamadas ao *driver* CUDA, responsável por invocar os *kernels*. Durante a inicialização do programa, as rotinas CUDA identificam o modelo da GPU e escolhem as versões binárias correspondentes. Caso não exista uma versão binária adequada, ela poderá ser compilada em tempo de execução através do código PTX gerado, que é um código binário de alto nível. Isto torna possível

sua portabilidade para diversos tipos de GPUs.

Desta forma, somente os *kernels* podem ser processados na GPU, e cada *kernel* pode ser disparado com uma ou mais *threads*. Com o objetivo de organizar a computação paralela das *threads* na GPU, a arquitetura utiliza hierarquias: *warps*, blocos e *grids*.

Warp é um conjunto de 32 *threads* que são sempre executadas juntas pelos *cores* de uma SM, em estilo SIMD. Bloco é um conjunto de *warps* designados para uma mesma SM. Desta forma, quando os *cores* de uma SM estiverem livres, a SM irá buscar nos blocos, uma *warp* pronta para execução, distribuindo suas *threads* entre os *cores*. Como cada SM possui duas unidades de disparo de instruções, mais de um *warp*, com diferentes instruções, pode ser executado simultaneamente, otimizando o uso dos recursos disponíveis.

Os blocos podem ser configurados em organizações de *threads* com até 3 dimensões. Isto facilita a indexação de suas *threads*, evitando a necessidade de cálculos no uso de matrizes, vetores e outras estruturas de dados com 2 ou 3 dimensões.

No último nível hierárquico, está o *grid* de blocos, que é mais uma camada de abstração para organizar os blocos em estruturas de até 3 dimensões. O *grid* representa todos os blocos de *threads* designados para computação em uma única GPU.

A **Figura 2.3** apresenta a hierarquia CUDA, já descrita, e a relação das *threads* com as respectivas memórias disponíveis. Além do acesso à memória, a hierarquia também influencia nas formas de comunicação e sincronismo entre as *threads*.

Cada *thread* possui uma memória local privada, residente na memória global, com acesso através dos *caches* L1 e L2, usada para *register spills*, chamadas de funções recursivas e variáveis automáticas de *arrays*. Cada bloco possui uma memória compartilhada (*shared memory*), usada para compartilhamento de dados entre suas *threads*, que é per-

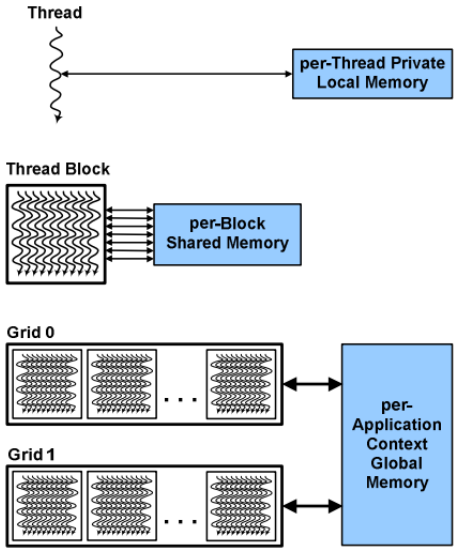

FIGURA 2.3 – Hierarquia CUDA e espaços de memória.

dida ao término da execução do bloco. As *threads* em diferentes blocos, ou seja, todas as *threads* da *grid*, somente compartilham resultados na memória global, que também pode ser acessada pela CPU, e seus dados permanecem entre chamadas de *kernel* (NVIDIA, 2009).

Sincronizações somente são possíveis entre *threads* de um mesmo bloco, ou entre os blocos, ao final de uma chamada de *kernel*, quando o *grid* termina a execução. Porém, soluções por *software* podem ser realizadas, e são discutidas na **Subseção 2.1.6**, por serem utilizadas neste trabalho.

Existem ainda diversos outros recursos disponíveis e restrições importantes que devem ser levadas em conta ao se utilizar este tipo de arquitetura, como por exemplo: padrão de acesso às memórias disponíveis, *hardware* dedicado para funções gráficas específicas, *caches* de textura, tipos especiais de estruturas de dados com atuação diferenciada no *hardware*, etc. Os recursos especiais utilizados serão apresentados oportunamente.

2.1.2 As chamadas de funções *kernel*

As chamadas de funções *kernel* são escritas no código fonte C com a adição de parâmetros dentro da sintaxe $\lll \ggg$, que descrevem a configuração de blocos e *threads*. Por exemplo, caso se deseja invocar o *kernel* de nome *func1* com o parâmetro *p1* e a configuração de 10 blocos e 32 *threads* por bloco, a chamada poderia ser feita com inteiros da seguinte forma: *"func1* $\lll$ *10,32* $\ggg$*(p1);"*.

Quando se deseja chamar um *kernel* com *threads* ou blocos organizados em estruturas de 2 ou 3 dimensões, é necessário o uso de variáveis do tipo *dim3*, que é uma estrutura com 3 inteiros: x, y e z.

Algoritmo 1 Chamada de um kernel

```
1: dim3 dimGrid(3, 2, 1);
2: dim3 dimBloco(10, 7, 5);
3: func1 <<< dimGrid, dimBloco >>> ();
```

Suponha que se deseja chamar o *kernel* de nome *func1* com uma *grid* de 3 blocos na dimensão x, 2 blocos na dimensão y e 1 bloco na dimensão z, onde cada bloco tenha 10 *threads* na dimensão x, 7 *threads* na dimensão y e 5 *threads* na dimensão z. A chamada a este *kernel* é apresentada no **Algoritmo 1**.

Vale ressaltar que, embora pode-se invocar um *kernel* com uma grande quantidade de *threads* e blocos, cada GPU possui recursos limitados para computá-los, ou seja, as SMs são capazes de tratar somente uma parcela destes blocos e *threads* a cada momento. Este limite é chamado de *threads ativas*, *blocos ativos* ou *warps ativos*. Após o término da computação de um bloco, outro é requisitado pela SM para computação.

2.1.3 Identificação das *threads* no *kernel*

Em CUDA, a identificação única das *threads* na *grid* pode ser calculada a partir da configuração utilizada na chamada ao *kernel*, com a ajuda de registradores especiais definidos em tempo de execução, que armazenam os identificadores únicos e as configurações.

As estruturas *gridDim* e *blockDim*, do tipo *dim3*, representam alguns destes registradores e armazenam, respectivamente, a quantidade de blocos na *grid* e *threads* por bloco, em cada dimensão. As estruturas $blockIdx$ e $threadIdx$, do mesmo tipo, armazenam, respectivamente, os índices do bloco na *grid* e da *thread* no bloco, também em cada dimensão.

Assim, caso um *kernel* seja invocado com as dimensões y e z iguais a 1, a quantidade total de *threads* pode ser calculada através da expressão $gridDim.x * blockDim.x$ e o identificador único da *thread* na *grid* pode ser calculado pela expressão $blockIdx.x * blockDim.x + threadIdx.x$. Estes valores são utilizados nos *kernels* deste trabalho, e são representados, respectivamente, pelas variáveis *inc* e *tid*.

2.1.4 Memória mapeada

Em GPUs com versão de capacidade acima de 1.0, existe a possibilidade de mapear um bloco de memória bloqueado na memória física do computador, ou seja, blocos que não permitam sua troca para a memória virtual, como se pertencessem à memória da GPU, aumentando assim a memória disponível para a GPU. Entretanto, esta memória é limitada pela banda do barramento e não é armazenada nos *caches* da GPU, implicando em uma alta latência que possibilita seu uso apenas em padrões de acesso pela GPU onde cada posição da memória é lido ou escrito apenas uma única vez (NVIDIA, 2012b),

(NVIDIA, 2012c).

O SDK CUDA permite o bloqueio na memória física de um bloco linear, alocado por funções como *malloc*, através da função *cudaHostRegister*, e o desbloqueio pode ser realizado pela função *cudaHostUnregister*. Após bloquear a memória, o ponteiro para o endereço mapeado na GPU pode ser configurado através da função *cudaHostGetDevicePointer*. Este endereço deve ser passado como parâmetro durante a chamada ao *kernel* para que a GPU possa enxergar a memória mapeada.

2.1.5 Textura

As texturas são dados, somente de leitura, armazenados na memória global, porém, com *caches* exclusivos e acesso realizado através de *hardware* dedicado. Este recurso é utilizado na segunda implementação (**Seção 5.3**), como uma forma de liberar os *caches* e requisições da memória global.

Várias funcionalidades são oferecidas por este *hardware*, podendo modificar tanto a forma de acesso como o próprio dado (por exemplo: interpolação linear dos dados e acesso normalizado através de indexadores com valores de ponto flutuante).

Algumas etapas são necessárias para a criação de uma textura, e as funções do SDK CUDA utilizadas são específicas para cada tipo de dado. Neste trabalho, é utilizada apenas uma textura referente a um vetor de inteiros sem sinal. Esta textura pode ser declarada em CUDA C como: "*texture <unsigned int, cudaTextureType1D, cudaReadModeElementType> tex;*".

A palavra-chave *texture* define o tipo da variável *tex* como textura, seguido de três parâmetros, que identificam o tipo da textura. O primeiro define os elementos da textura

como inteiros não sinalizados. O segundo define a textura como uma estrutura unidimensional (um vetor). O terceiro faz o acesso aos dados serem como em um vetor, ou seja, acessando cada elemento através de um índice inteiro.

Após a criação da variável textura, é necessário associá-la aos seus respectivos dados alocados na memória global, tornando possível seu acesso pelo *kernel*.

Esta associação pode ser feita com o auxílio da função *cudaBindTexture*: "*cudaBindTexture(0, tex, dw, sizeof(unsigned int)*n);*". O primeiro parâmetro é apenas um valor opcional de *offset*. A textura *tex* é passada no segundo parâmetro e o ponteiro *dw* para os dados na memória global é passado como terceiro parâmetro. Por último, a quantidade de *bytes* dos dados associados deve ser especificada.

No *kernel*, é possível recuperar um dado da textura com a função *text1Dfetch*, uma vez que a textura *tex* declarada é unidimensional. Por exemplo, a requisição do elemento de índice 3 pode ser realizada como: "*tex1Dfetch(tex, 3);*".

2.1.6 Barreira de sincronismo entre blocos

Em (XIAO; FENG, 2010), são apresentadas três técnicas para a realização de sincronismo entre blocos na GPU. A primeira utiliza operações atômicas em uma variável global para indicar que todos os blocos chegaram na barreira de sincronismo. A segunda técnica utiliza um conjunto de variáveis organizadas em uma estrutura de árvore para reduzir a competitividade pela exclusividade das variáveis requisitadas nas operações atômicas. A terceira técnica utiliza dois vetores, onde cada um de seus elementos representa um bloco, com a finalidade de evitar a necessidade de operações atômicas.

A última técnica é a que apresenta melhor resultado, alcançando na média menos

Algoritmo 2 Sincronismo entre blocos

```
1: // goalVal: identificador único de sincronismo
2: // Array_in: vetor de chegada com nBlockNum elementos
3: // Array_out: vetor de sincronismo com nBlockNum elementos

4: function GPU_SYNC(goalVal, Array_in, Array_out)
5:     // Identificação única da thread no bloco
6:     tid_in_block ← threadIdx.x
7:     // Quantidade total de blocos da grid
8:     nBlockNum ← gridDim.x * gridDim.y
9:     // Identificação do bloco ao qual a thread pertence
10:    bid ← blockIdx.x * gridDim.y + blockIdx.y

11:    // Correção 1
12:    // Correção 2

13:    // Somente as threads 0 de cada bloco informam a chegada
14:    if tid_in_block = 0 then
15:        Array_in[bid] ← goalVal
16:    end if

17:    // Somente o bloco 1 libera as threads 0
18:    if bid = 1 then
19:        if tid_in_block < nBlockNum then
20:            while Array_in[tid_in_block] ≠ goalVal do
21:                // Nada
22:            end while
23:        end if
24:        __syncthreads()
25:        if tid_in_block < nBlockNum then
26:            Array_out[tid_in_block] ← goalVal
27:        end if
28:    end if

29:    // As threads 0 aguardam liberação
30:    if tid_in_block = 0 then
31:        while Array_out[bid]! = goalVal do
32:            // Nada
33:        end while
34:    end if

35:    // As threads de um bloco aguardam a thread 0
36:    __syncthreads()
37: end function
```

do que um terço do tempo gasto pelo sincronismo implícito realizado entre chamadas de *kernel* (XIAO; FENG, 2010). O pseudocódigo apresentado pelos autores é reproduzido no **Algoritmo 2**, onde os parâmetros da função são, respectivamente, um identificador único para cada sincronismo e dois vetores com o tamanho do número de blocos.

Neste algoritmo, as *threads* dentro de um mesmo bloco aguardam sua *thread* 0 realizar o sincronismo entre todos os blocos, portanto, somente as *threads* com *tid_in_block=0* participam da sincronização entre os blocos.

Nas linhas 14-16, a *thread* 0 de cada bloco escreve o valor de sincronismo $goalVal$ na respectiva posição (referente ao seu bloco) do vetor de entrada $Array_{in}$, indicando que a barreira de sincronismo foi alcançada. No final (linhas 30-34), as *threads* 0 aguardam este mesmo valor $goalVal$ no vetor de saída $Array_{out}$. O sincronismo da linha 36 faz com que as demais *threads* de um bloco aguardem a liberação da correspondente *thread* 0.

A liberação de todas as *threads* 0 é feita por apenas um bloco ($bid = 1$, na linha 18) para garantir a sincronização. Durante esta liberação, $nBlockNum$ *threads*, ou seja, tantas *threads* quanto o número de blocos, observam as posições que indicam a chegada das *threads* 0 no ponto de sincronismo. Após este evento, cada *thread* correspondente alcança uma barreira de sincronismo entre as *threads* do bloco (linha 24). Assim, uma restrição deste método é que um bloco contenha uma quantidade de *threads* igual ou superior ao números de blocos.

Quando todas as *threads* do bloco 1 alcançarem este sincronismo, então todas as *threads* 0 também alcançaram a barreira de sincronismo entre blocos. Neste momento, as *threads* do bloco 1 são liberadas para sinalizar o vetor $Array_{out}$, que por sua vez, libera as *threads* 0 presas no laço das linhas 31-33, finalizando a sincronização.

Entretanto, foi observada uma falha nesta estratégia. Suponha que a *thread* 40 do bloco 0 não tenha chegado à barreira de sincronismo entre blocos, mas isto tenha ocorrido com a *thread* 0 do mesmo bloco. Neste caso, esta *thread* 0 indicará sua chegada no vetor $Array_{in}$ e aguardará a liberação no vetor $Array_{out}$. As demais *threads* 0 dos outros blocos podem também chegar na barreira de sincronismo, indicando este evento no vetor $Array_{in}$, o que liberará as *threads* do bloco 1, que realizam o sincronismo entre blocos, fazendo com que os valores do vetor $Array_{out}$ sejam atualizados e terminando o sincronismo entre blocos. No entanto, a *thread* 40 do bloco 0 pode ainda não ter chegado à barreira entre blocos, ocasionando uma falsa sincronização.

A seguir, é proposta uma simples correção para este problema. Basta fazer com que a *thread* 0 de cada bloco somente indique sua chegada ao ponto de sincronismo após ter certeza de que todas as *threads* de seu bloco também chegaram à esta barreira. Para isto, foi adicionada uma chamada à $\_\_syncthreads()$ na linha 12 do **Algoritmo 2**.

Como a função $\_\_syncthreads()$ garante coerência de memória apenas para as *threads* de um mesmo bloco, caso seja exigida a coerência de memória entre todas as *threads* da *grid*, será ainda necessário incluir uma chamada à função $\_\_threadfence()$ na linha 11, que possui exatamente este propósito.

Estas correções podem invalidar os resultados de desempenho obtidos por Xiao & Feng (2010), devido às duas novas instruções que impedem as *threads* 0 de anunciarem rapidamente sua chegada à barreira de sincronismo entre blocos, pois não é garantido que as outras *threads* do mesmo bloco também tenham alcançado a barreira.

2.1.6.1 Barreira de dependência cíclica entre blocos

Na estratégia anterior, um bloco deve aguardar a finalização de todos os demais, ou seja, a duração da sincronização é limitada sempre pelo bloco mais lento, o que pode implicar em um tempo de espera considerável para algoritmos com desbalanceamento de tarefas. Nesta subseção, é proposto um novo método de sincronização entre blocos com o objetivo de reduzir o tempo ocioso das *threads*.

Esta nova estratégia pode ser utilizada quando existir dependência cíclica entre blocos anteriores, ou seja, quando uma tarefa da etapa t do bloco b depende da computação de tarefas da etapa t dos $b-1$ blocos anteriores, e o primeiro bloco depende do término de todas as tarefas da etapa $t-1$ dos demais blocos, antes de iniciar a nova etapa t. Desta forma, ao invés de todos aguardarem o bloco mais lento, cada bloco deverá apenas esperar os blocos anteriores.

Isto pode ser feito através da abstração de um relógio, vetor *Clock*, correlacionando uma hora a cada etapa da computação, e atribuindo uma posição do vetor *Clock* para cada bloco.

A cada sincronização, um bloco somente poderá atualizar seu relógio caso o relógio do bloco anterior estiver em um horário posterior. A ideia é não permitir que blocos posteriores estejam com horário à frente de blocos anteriores. O primeiro bloco é o responsável por iniciar a atualização do relógio; porém, devido à dependência, não poderá realizá-lo enquanto todos os demais blocos não tiverem o mesmo valor em seus relógios.

A **Figura 2.4** apresenta um exemplo do vetor *Clock* e os respectivos horários de cada bloco. Neste exemplo, considere que o bloco 2 termine sua computação no tempo t_0. Como o bloco anterior possui o mesmo horário, o bloco 2 não poderá atualizar seu relógio.

	0	1	2	3
t_0	1	1	1	1
t_1	2	1	1	1
t_2	2	2	1	1
t_3	2	2	2	1
t_4	2	2	2	2
t_5	3	2	2	2

FIGURA 2.4 – Exemplo de sincronismo cíclico entre blocos.

No tempo t_1, o bloco 0 termina sua computação e, como o último bloco está no mesmo horário, atualiza seu relógio para 2. No tempo t_2, o bloco 1 termina sua computação e também ajusta seu relógio. No tempo t_3, o bloco 2 enxerga a atualização do bloco 1, atualiza seu relógio e finalmente prossegue com sua computação.

Quando um bloco terminar toda sua computação, deverá desligar o relógio escrevendo 0 em sua respectiva posição do vetor *Clock*, quebrando a dependência cíclica e fazendo com que todos os outros blocos também terminem sua dependência, em uma reação em cadeia. O pseudocódigo é apresentado no **Algoritmo 3**.

Assim como no método anterior, a quantidade de *threads* por bloco deverá ser igual ou maior do que a quantidade de blocos. O vetor *Clock* deverá ainda ser declarado como do tipo *volatile*, para indicar ao compilador que seus valores podem ser alterados por qualquer *thread*. Caso contrário, a *thread* 0 de um bloco poderá realizar a leitura de *Clock* e armazená-la em *cache* ou registradores, nunca enxergando as alterações dos outros blocos, ocasionando *deadlock*.

Este algoritmo recebe como parâmetro apenas o vetor *Clock*, e há a necessidade do vetor estar previamente inicializado com valores 1 em todas suas posições no início da

Algoritmo 3 Sincronização cíclica entre blocos - Parte 1

```
1: // Clock: Vetor previamente inicializado com elementos iguais a 1

2: function __cycleSyncBlocks(Clock)
3:     // Identificação única da thread no bloco
4:     tid_in_block ← threadIdx.x
5:     // Quantidade total de blocos da grid
6:     nBlockNum ← gridDim.x * gridDim.y
7:     // Identificação do bloco ao qual a thread pertence
8:     bid ← blockIdx.x * gridDim.y + blockIdx.y

9:     // Sincronismo das threads de um bloco, com coerência de memória
10:    __threadfence()
11:    __syncthreads()

12:    // Somente as threads 0 de cada bloco são utilizadas
13:    if tid_in_block = 0 then
14:        // Recupera o horário atual do bloco
15:        now ← Clock[bid]

16:        // Se for o primeiro, observar o último bloco
17:        if bid = 0 then
18:            // Aguarda o último bloco alcançar o horário
19:            // ou desligar o relógio
20:            while Clock[nBlockNum − 1] ≠ 0 ∧ Clock[nBlockNum − 1] ≠ now do
21:                // Nada
22:            end while
23:            // Se o relógio não foi desligado
24:            if Clock[nBlockNum − 1] ≠ 0 then
25:                // Atualizar o horário
26:                now ← now + 1
27:                // Caso ocorra overflow
28:                if now = 0 then
29:                    // Reiniciar o horário
30:                    now ← 1
31:                end if
32:            else
33:                // Caso contrário, também desligar o relógio em cadeia
34:                now ← 0
35:            end if
```

computação.

Somente as *threads* 0 realizam o sincronismo entre blocos, de modo análogo ao método anterior, e iniciam recuperando o horário atual do relógio na variável não sinalizada *now*

(linha 15). Em seguida, a *thread* 0 do primeiro bloco analisa o último bloco (linhas 13-35), enquanto que cada *thread* 0 dos demais blocos analisa seu respectivo bloco anterior (linhas 36-52 do **Algoritmo 4**).

Algoritmo 4 Sincronização cíclica entre blocos - Parte 2

```
36:         else
37:             // Caso contrário, observar o bloco anterior

38:             // Atualizar o horário
39:             now ← now + 1
40:             if now = 0 then
41:                 now ← 1
42:             end if
43:             // Aguarda o bloco anterior chegar no horário atual
44:             // ou desligar seu relógio
45:             while Clock[bid − 1] ≠ 0 ∧ Clock[nBlockNum − 1] ≠ now do
46:                 // Nada
47:             end while
48:             // Desligar o relógio em cadeia
49:             if Clock[bid − 1] = 0 then
50:                 now ← 0
51:             end if
52:         end if

53:         // Threads 0 atualizam os relógios
54:         Clock[bid] ← now
55:     end if

56:     // Demais threads aguardam o sincronismo entre blocos,
57:     // realizado pelas threads 0
58:     __syncthreads()
59: end function
```

O procedimento segue como apresentado no exemplo anterior, com exceção da adição de condicionais que verificam o desligamento do relógio. A *thread* 0 do primeiro bloco verifica na linha 24 se o relógio do último bloco está desligado, desligando também o seu em caso positivo (linha 34). Caso contrário, o horário do primeiro bloco é atualizado (linha 26), e em seguida é verificado se o incremento gerou *overflow* (linha 28). Este teste é importante, uma vez que o horário igual a 0 representa o desligamento do relógio. Após definido o novo horário, o relógio é atualizado para este valor na linha 54.

O **Algoritmo 4** apresenta o pseudocódigo referente ao procedimento das *threads* 0 para os demais blocos. A única diferença é que, neste caso, as *threads* 0 primeiro atualizam o horário atual e depois aguardam o bloco anterior alcançar o novo horário, antes de atualizarem seus respectivos relógios.

Ao final, após o sincronismo entre blocos, as *threads* 0 de cada bloco alcançam a barreira da linha 58, onde as demais *threads* aguardam o sincronismo.

2.2 O problema *Subset-Sum*

O SSP foi inicialmente apresentado por George Pólya (PÓLYA, 1956) na década de 50 e Paul Erdös (SUN, 2003) na década de 60, e pode ser descrito informalmente como o problema de encontrar qual subconjunto de uma lista de inteiros que possui uma soma específica. Segundo (GAREY; JOHNSON, 1979), este problema pertence à classe dos problemas NP-Completos e possui diversas variações: somente responder se o subconjunto existe; encontrar o maior subconjunto; encontrar o subconjunto de uma lista de inteiros com números negativos e positivos que somem 0; etc.

Este trabalho aborda uma versão do SSP que pode ser descrita como um caso especifico do *Problema da Mochila 0/1* (KP01). Considerando um conjunto de n itens $A = (a_1, a_2, \ldots, a_n)$, cada um com peso $w_i \in \mathbb{N}^+$ e lucro $p_i \in \mathbb{N}^+$, $1 \leq i \leq n$, KP01(A,c) é o problema de encontrar o subconjunto de A mais lucrativo sem que seus pesos excedam a capacidade da mochila $c \in \mathbb{N}^+$. KP01(A,c) pode ser formalmente definido como o seguinte problema de programação inteira:

$$\begin{cases} \max \sum_{i=1}^{n} p_i x_i \\ \sum_{i=1}^{n} w_i x_i \leq c \\ x_i \in \{0,1\}, 1 \leq i \leq n \end{cases}$$

A solução deste problema pode ser representada pelo vetor binário $X = \{x_1, \ldots, x_n\}$. Com o intuito de evitar soluções triviais, pode-se assumir, sem perda de generalidade, que $w_{\min} < w_{\max} < c < \sum_{i=1}^{n} w_i$, onde $w_{\min}$ e $w_{\max}$ são, respectivamente, o menor e o maior peso dos itens em A.

Deste modo, SSP pode ser definido como um caso particular do KP01(A,c), apenas considerando $p_i = w_i$, $1 \leq i \leq n$. Devido ao fato dos itens do SSP terem peso e lucro iguais, é comum referir-se ao peso w_i como se fosse o próprio item a_i.

Os algoritmos que solucionam o KP01 de forma exata podem ser classificados basicamente em dois paradigmas: força bruta e programação dinâmica.

De acordo com (O'NEIL; KERLIN, 2010), uma instância do KP01 pode ser classificada entre esparsa e densa, onde quanto mais densa maior é a chance de existir uma combinação entre $w_{\min}$ e $\sum_{i=0}^{n} w_i$, e quanto mais esparsa menor é a chance de existir esta combinação.

Usualmente, o método de programação dinâmica encontra as soluções para instâncias densas do KP01 em menor tempo de execução, enquanto que a força bruta detém menores tempos em instâncias esparsas. Isto se deve ao fato da programação dinâmica computar todas as possíveis combinações para todos os itens de forma incremental e, nos casos esparsos, haverá mais capacidades sem combinação possível. Por outro lado, na técnica de força bruta, apenas as combinações possíveis são analisadas.

A solução óbvia, através de força bruta, computa todas as combinações dos n itens,

ocasionando uma complexidade $O(2^n)$.

A primeira resolução do KP01 em tempo e espaço $O(2^{n/2})$ usando força bruta foi o algoritmo das duas listas de Horowitz & Sahni (HOROWITZ; SAHNI, 1974). A ideia é dividir os n itens em duas partes iguais, gerando duas listas ordenadas com $O(2^{n/2})$ possíveis soluções cada. A solução final será obtida através de um percurso em ambas as listas.

Utilizando o método de programação dinâmica de Bellman (BELLMAN, 1957), o KP01 pode ser solucionado em tempo e espaço pseudopolinomial $O(nc)$.

2.3 O Estado da Arte na resolução do SSP

Com relação ao SSP, o melhor algoritmo sequencial, em termo de complexidade teórica, gasta tempo $O(2^{\sqrt{x}})$, onde x é o tamanho total em *bits* do conjunto de pesos. Originalmente desenvolvido para resolver o caso específico *partition problem*, este algoritmo foi descoberto por Stearns & Hunt (STEARNS; HUNT, 1990): basicamente, divide o problema original em subproblemas esparsos e densos. Após a divisão, a programação dinâmica é utilizada no subproblema denso e o subproblema esparso é resolvido por força bruta, alcançando uma complexidade sub-exponencial em tempo.

Recentemente, O'Neil & Kerlin (O'NEIL; KERLIN, 2010) publicaram um algoritmo sequencial mais simples e que também resolve o SSP em tempo sub-exponencial utilizando uma lista dinâmica esparsa e realizando verificações de redução nesta lista para manter somente combinações candidatas à solução. Este algoritmo é similar a outros de programação dinâmica para o SSP que usam lista dinâmica; entretanto, com o uso da lista dinâmica e algumas otimizações, foi possível chegar à complexidade sub-exponencial no

tempo.

Dentre os melhores resultados práticos até hoje, pode-se citar o algoritmo Decomp de Pisinger (KELLERER; PFERSCHY; PISINGER, 2004), que é uma versão modificada do algoritmo de Horowitz & Sahni. Neste trabalho, os pesos são também divididos em duas listas disjuntas, porém uma lista l_a representa itens a_i fora da mochila e a outra lista l_b contém itens b_j dentro da mochila. A cada passo, um item b_j é retirado da lista l_b e outro item a_i é adicionado na lista l_a.

Uma operação de casamento é realizada para verificar as capacidades alcançadas combinando as duas listas, e o processo segue até que todos os n itens sejam incluídos e retirados. Este método apresenta a mesma complexidade de tempo e espaço que o algoritmo de Bellman $O(nc)$; entretanto, pode ser melhorado (KELLERER; PFERSCHY; PISINGER, 2004).

Com base em uma técnica parecida, Pisinger foi o primeiro a apresentar um algoritmo para o SSP com complexidade temporal menor do que $O(nc)$ (PISINGER, 1995). Seu algoritmo Balsub é uma técnica de programação dinâmica restrita a apenas algumas capacidades da mochila, solucionando o SSP em tempo e espaço $O(nr)$, onde $r \leq w_{\max}$, ou seja, o algoritmo soluciona o SSP em tempo e espaço $O(nw_{\max})$. Embora o autor não apresente nenhum método para retornar o vetor solução X, em (KELLERER; PFERSCHY; PISINGER, 2004) é mencionado que o vetor pode ser encontrado com a ajuda dos vetores computados e uma análise reversa das escolhas que os preencheram.

As primeiras resoluções do SSP com espaço $O(n + c)$ foram os trabalhos (SOMA; YANASSE, 1987) e (SOMA; TOTH, 2002), sendo o último uma variação do primeiro, com complexidade temporal de, respectivamente, $O(n(c-2w_{\min})+c)$ e $O((n-\log_2 c^2)(c-2w_{\min})+c)$.

Dentre as soluções não sequenciais, os melhores resultados teóricos para o SSP estão em (SANCHES; SOMA; YANASSE, 2007; SANCHES; SOMA; YANASSE, 2008). Em (SANCHES; SOMA; YANASSE, 2007), é apresentada uma paralelização ótima do algoritmo de duas listas: no modelo *Parallel Random-Access Machine* (PRAM) SIMD, o SSP foi resolvido em tempo $O(2^{n/2}/p)$ com $p = 2^q$ processadores, onde $0 \leq q \leq \frac{n}{2} - 2\log_2 n$. Até hoje, é o melhor resultado conhecido no paradigma de força bruta.

Considerando o paradigma de programação dinâmica, Sanches, Soma & Yanasse (SANCHES; SOMA; YANASSE, 2008) desenvolveram três novos algoritmos paralelos escaláveis na PRAM SIMD, baseados no algoritmo de Yanasse & Soma (SOMA; YANASSE, 1987), estabelecendo novos *upper bounds* de tempo e espaço para o SSP. Por exemplo, um destes algoritmos gasta tempo $O(\frac{n}{p}(c - 2w_{\min}) + \frac{c}{p} + n)$ e espaço $O(n + c)$, com $\log_2 n \leq p \leq n$ processadores.

Resultados práticos da solução do SSP em arquitetura paralela são apresentados pelo recente trabalho (BOKHARI, 2012), que faz um estudo em três arquiteturas contemporâneas: Cray Extreme *multithread* com 128 processadores, IBM de memória compartilha com 16 processadores e GPU NVIDIA com 240 *cores*, gastando tempo e espaço $O(nc)$.

Outro recente trabalho com resultados práticos é (BOYER; BAZ; ELKIHEL, 2012), para a solução do KP01 em GPU, também gastando tempo e espaço $O(nc)$, mas reduzindo o tráfego de dados entre GPU e CPU graças a uma estratégia de compactação.

Vale ainda ressaltar que os algoritmos de O'Neil & Kerlin (O'NEIL; KERLIN, 2010) e Boyer, Baz & Elkihel (BOYER; BAZ; ELKIHEL, 2012) apenas encontram soluções de instâncias do SSP onde a combinação máxima é idêntica ao valor desejado c, ou seja, não são capazes de encontrar soluções ótimas menores do que a capacidade da mochila.

3 Alguns algoritmos conhecidos

Neste capítulo, serão descritos em mais detalhes alguns algoritmos conhecidos para a resolução do SSP. Na última seção, será apresentada uma nova função recursiva, que será utilizada nas implementações paralelas deste trabalho.

3.1 Algoritmo clássico de Bellman

No caso particular do KP01, a programação dinâmica de Bellman (BELLMAN, 1957) computa a solução ótima $f_n(c)$ de KP01$(\{a_1, \ldots, a_n\}, c)$ a partir de subsoluções $f_i(k)$, onde $0 \leq i \leq n$ e $0 \leq k \leq c$, ou seja, são considerados somente os i primeiros itens de A e a capacidade da mochila k. Definindo $f_0(k) = f_i(0) = 0$, a solução de Bellman pode ser expressa pela seguinte recursão:

$$f_i(k) = \begin{cases} f_{i-1}(k) & 0 \leq k < w_i \\ \max\{f_{i-1}(k - w_i) + p_i, f_{i-1}(k)\} & w_i \leq k \leq c \end{cases}$$

A função f pode ser representada em uma matriz, conhecida como a tabela de Bellman, que é computada em n passos, cada um correspondendo ao cálculo de c capacidades da mochila. Tomando o item a_i, são analisadas todas as possíveis combinações com os primeiros i itens e somente no passo seguinte que serão analisados os preenchimentos com

os $i+1$ primeiros itens.

Algoritmo 5 Algoritmo sequencial de Bellman

```
1:  // Inicia a coluna 0 com 0
2:  for i ← 1; i ≤ n; i ← i + 1 do
3:      B[i][0] ← 0
4:  end for
5:  // Inicia linha 0 com 0
6:  for k ← 0; k ≤ c; k ← k + 1 do
7:      B[0][k] ← 0
8:  end for

9:  // Percorre todos os itens
10: for i ← 1; i ≤ n; i ← i + 1 do
11:     // Percorre todas as capacidades
12:     for k ← 1; k ≤ c; k ← k + 1 do
13:         if k < w_i then
14:             // Copia a solução anterior
15:             B[i][k] ← B[i − 1][k]
16:         else
17:             // Calcula a solução de maior lucro
18:             B[i][k] ← max{B[i − 1][k], B[i − 1][k − w_i] + w_i}
19:         end if
20:     end for
21: end for
```

O **Algoritmo 5** apresenta uma implementação desta solução para o SSP, onde B é a matriz de Bellman. A primeira parte do algoritmo (linhas 2-8) inicializa a matriz de Bellman e o restante calcula a recursão.

A recuperação do vetor X para qualquer capacidade k se faz com o auxílio desta tabela de Bellman. Como na coluna k e linha n está contido por definição o máximo lucro obtido adicionando os primeiros n elementos, basta analisar o primeiro passo j, a partir de n, em que não foi possível alcançar esta solução máxima. Como, somente com o item subsequente a_{j+1}, é possível alcançar a capacidade máxima, ele está na solução ótima. Retirando seu peso w_{j+1} da capacidade k, o processo continua com a nova capacidade restante $k - w_{j+1}$, até que o lucro máximo tenha sido alcançado.

3.2 Algoritmo de Bellman com matriz binária

Em (BOYER; BAZ; ELKIHEL, 2012) e (BOKHARI, 2012) é apresentada uma nova versão da programação dinâmica de Bellman para o KP01, com tempo e espaço $O(nc)$, em que a tabela de Bellman é preenchida apenas com informações binárias ao invés da soma dos lucros dos itens, além de uma otimização com o objetivo de evitar computações que não cheguem à capacidade da mochila c.

A otimização de Toth (TOTH, 1980) verifica se durante a computação da tabela de Bellman, para cada etapa i e capacidade k, onde $1 \leq i \leq n$ e $1 \leq k \leq c$, é possível ou não chegar na capacidade c com os próximos itens disponíveis, isto é, se $k + \sum_{j=i+1}^{n} w_j \geq c$.

Assim, para cada etapa i, pode-se isolar a variável k com o objetivo de encontrar a capacidade mínima que deve ser analisada para se chegar à solução: $k \geq c - \sum_{j=i+1}^{n} w_j$. Como o peso w_i é considerado na etapa i em capacidades $k \geq w_i$, então pode-se definir a capacidade mínima para cada item i como: $k_{i\,\min} = \max\{w_i, c - \sum_{j=i+1}^{n} w_j\}$.

Algoritmo 6 Algoritmo de Bellman sequencial com matriz binária

```
1:  // Percorre todos os itens
2:  for i ← 1; i ≤ n; i ← i + 1 do
3:      // Otimização de Toth
4:      k_{i min} = max{w_i, c − Σ_{j=i+1}^{n} w_j}
5:      // Percorre todas as capacidades
6:      for k ← k_{i min}; k ≤ c; k ← k + 1 do
7:          // Caso o lucro seja maior
8:          if T[k] < T[k − w_i] + p_i then
9:              // Atualiza o vetor T e seta a matriz de decisão
10:             T[k] ← T[k − w_i] + p_i
11:             B[i][k] ← 1
12:         end if
13:     end for
14: end for
```

O algoritmo de Bellman com matriz binária utiliza um vetor auxiliar T de tamanho c para armazenar as somas dos lucros e uma tabela binária B, chamada de tabela de decisão.

A cada passo i e capacidade k, a posição binária $B[i][k]$ é setada caso seja realizada a escrita no vetor T auxiliar, como pode ser visto no **Algoritmo 6**.

A recuperação do vetor X se faz pela mesma forma que a de Bellman, porém procurando o item j correspondente na linha da tabela de decisão com valor 0 e observando o lucro obtido com o vetor auxiliar.

Contudo, vale ressaltar que ao utilizar a otimização de Toth, o algoritmo se restringe a solucionar apenas combinações ótimas $w_i x_i = c$, pois como $k_{i\,\min} = \max\{w_i, c - \sum_{j=i+1}^{n} w_j\}$, caso a solução ótima seja $c - \delta$, a capacidade mínima deveria ser: $k'_{i\,\min} = \max\{w_i, c - \delta - \sum_{j=i+1}^{n} w_j\}$, podendo ser menor do que $c - \sum_{j=i+1}^{n} w_j$ pela variação δ.

3.3 Algoritmo de Yanasse & Soma

Yanasse e Soma (*cf.* (KELLERER; PFERSCHY; PISINGER, 2004), pp. 85) desenvolveram uma variação do algoritmo de Bellman específica para o SSP, que posteriormente foi paralelizada em (SANCHES; SOMA; YANASSE, 2008). Esta versão utiliza um único vetor g com $c+1$ posições que são calculadas de tal forma que, se existir um preenchimento para a mochila com capacidade k e o item a_i é o maior índice i presente nesta solução, então $g[k] = i$, onde $0 \leq k \leq c$ e $1 \leq i \leq n$. Caso a capacidade k admita múltiplas soluções, então somente o menor índice entre estas soluções será armazenado em g.

O **Algoritmo 7** que descreve esta ideia, possui basicamente três fases:

- Inicialização de g (linhas 2-8): todas as posições $c + 1$ são definidas para $n + 1$ (indicando nenhuma solução); em seguida, as soluções compostas por um único peso w_i, $1 \leq i \leq n$, são inseridas em g em ordem decrescente.

Algoritmo 7 Algoritmo sequencial de Yanasse & Soma

```
1:  // Inicializando o vetor g com n + 1 (nenhuma solução)
2:  for k ← 0; k ≤ c; k ← k + 1 do
3:      g[k] ← n + 1
4:  end for
5:  // Inicializando as soluções com um único item
6:  for i ← n; i ≥ 1; i = i − 1 do
7:      g[w_i] ← i
8:  end for

9:  // Inicia melhor solução s com w_max
10: s ← w_max
11: // Percorre todas as capacidades
12: for k ← w_min; k ≤ c − w_min; k ← k + 1 do
13:     // Computa as novas combinações
14:     for i ← g[k] + 1; i ≤ n; i ← i + 1 do
15:         // Novas combinações
16:         w' ← k + w_i
17:         if w' ≤ c then
18:             // Mantém menor índice
19:             g[w'] ← min{g[w'], i}
20:             // Atualiza solução
21:             s ← max{s, w'}
22:         end if
23:     end for
24: end for

25: // Inicializaçao do vetor X
26: for i ← 1; i ≤ n; i = i + 1 do
27:     x_i ← 0
28: end for
29: // Encontra solução s
30: k ← s
31: while k ≠ 0 do
32:     x_{g[k]} ← 1
33:     k ← k − w_{g[k]}
34: end while
```

- Avaliação dos valores de g remanescentes (linhas 10-24): os índices entre $w_{\min}$ e $c - w_{\min}$ são percorridos em ordem crescente, para o cálculo dos possíveis preenchimentos da mochila.

- Recuperação da solução ótima X (linhas 26-34): os itens do preenchimento ótimo são encontrados.

Do ponto de vista teórico, o **Algoritmo 7** é mais eficiente que o algoritmo de Bellman, uma vez que soluciona o SSP em tempo $O(n(c-2w_{\min})+c)$ e somente com espaço $O(n+c)$.

Este algoritmo apresenta três principais diferenças em relação à programação dinâmica tradicional de Bellman:

- Armazena os índices dos itens ao invés da soma dos pesos. Por este motivo, os elementos do vetor g necessitam de $\lceil \log_2(n) \rceil$ *bits* cada um, enquanto os elementos da tabela de Bellman ocupam $\lceil \log_2(w_{\max}) \rceil$ *bits*

- O vetor g expressa somente combinações exatas ao invés de somas máximas encontradas, ou seja, caso não exista uma combinação com soma t, logo $g[t] = n + 1$. No caso do algoritmo de Bellman, caso não exista uma solução igual a t a tabela armazenará a máxima solução possível menor que t.

- Ao invés de computar a adição do item a_i a cada passo i, o algoritmo de Yanasse & Soma combina, a cada passo k, todos os preenchimentos $k + w_i$, alterando o índice de $g[k + w_i]$ caso este seja maior que i.

O algoritmo de Bellman também permite uma importante variação: computar em um mesmo passo uma única capacidade da mochila, considerando diversos números de itens. Isto equivale a preencher a tabela de Bellman pelas colunas, ao invés das linhas. O algoritmo de Yanasse & Soma, por outro lado, somente pode ser computado com o laço mais externo percorrendo as capacidades, uma vez que o laço interno inicia por $g[k] + 1$.

3.4 Algoritmos paralelos

O algoritmo de Bellman com matriz binária foi escolhido para paralelização em (BOYER; BAZ; ELKIHEL, 2012), e, embora não tenha sido citado pelos autores, o algoritmo apenas soluciona combinações ótimas $w_i x_i = c$, pelo fato de utilizarem a otimização de Toth. Este é o mesmo algoritmo utilizado no trabalho (BOKHARI, 2012), porém, ela soluciona o SSP ao invés do KP01 e não utiliza a otimização de Toth.

Em ambos os casos, a computação de cada linha i da tabela de decisão é realizada através de uma chamada ao *kernel*, apenas com o auxílio da linha anterior $i-1$, armazenando os resultados na memória principal do *host* após o término do *kernel*.

O trabalho (BOYER; BAZ; ELKIHEL, 2012) realiza ainda uma compactação da linha antes de envia-la à CPU, com o objetivo de reduzir o tráfego de dados. Com esta compactação, para um *benchmark* específico, foi obtido no melhor caso uma redução com fator de 0,00031 do tamanho original gasto pelo algoritmo de Bellman com matriz binária.

A compactação da linha foi realizada pelo fato da observação dos autores, através de análises dos resultados do *benchmark* realizado, que as colunas da direita geralmente possuem valor 1, enquanto que as colunas da esquerda possuem valor 0. Devido a isto, os autores resolveram representar apenas os valores intermediários entre l e r, onde l é a primeira coluna com valor igual a 1 e r é a última coluna com valor igual a 0, além de marcar suas posições iniciais.

Em seus trabalhos, Boyer, Baz & Elkihel (2012) alcançaram uma redução de até 26 vezes do tempo de execução em um *benchmark* específico e Bokhari (2012) concluiu que as GPUs apresentam bons resultados para pequenos problemas, quando os dados cabem dentro da memória da GPU.

3.5 Algoritmo mais adequado para GPU

Embora as recentes implementações do SSP em GPU apresentem bons resultados ((BOYER; BAZ; ELKIHEL, 2012) e (BOKHARI, 2012)), elas podem exigir uma grande quantidade de memória, mesmo com o alto grau de compactação obtido pelo algoritmo de Boyer, Baz & Elkihel, pois gastam espaço $O(nc)$, o que limita severamente a quantidade de instâncias que podem ser solucionadas. Por exemplo, se este algoritmo fosse utilizado para resolver uma instância com $n = 10^6$ itens com valores armazenados em variáveis de 32-bit e uma capacidade da mochila $c = 2^{30}$, com a melhor taxa de compactação obtida (∼0,031%), a tabela de decisão necessitaria de $\frac{2^{30}*10^6*0{,}00031}{1024^3*8} = 38{,}75$ GB de memória principal, tornando-se inviável nas GPUs atuais.

Neste sentido, a paralelização de (SANCHES; SOMA; YANASSE, 2008) leva vantagem por utilizar espaço $O(n+c)$, tornando-se uma melhor alternativa de implementação em GPU. No mesmo exemplo acima, o vetor g, com elementos de 4 bytes, gastaria $\frac{2^{30}*4}{1024^3} = 4$ GB.

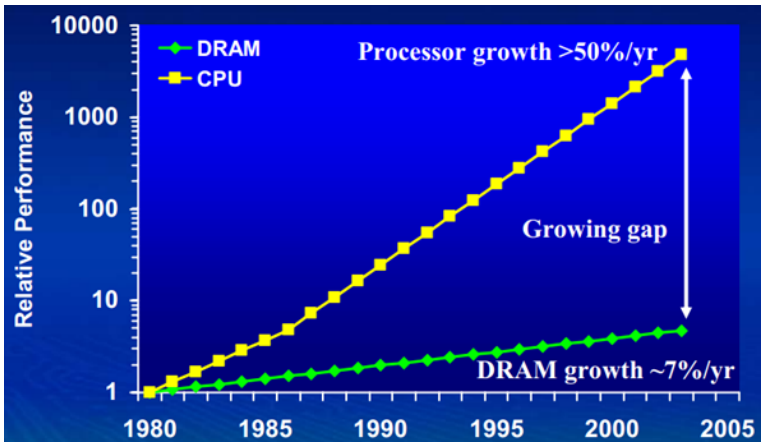

FIGURA 3.1 – Defasagem entre latência de memória e processamento.

A justificativa pela escolha de um algoritmo com menor consumo de espaço se faz principalmente pelas características das GPUs atuais, com pouca memória para ser compartilhada entre milhares de *threads*. Outra razão é a defasagem da latência da memória

em relação à capacidade de computação dos *cores*, que aumenta ao longo dos anos, como pode ser observado na **Figura 3.1** (YUNG; RUSU; SHOEMAKER, 2002).

Deste modo, foi preferido optar por algoritmos que façam menor uso de memória, mesmo que para isto o tempo seja penalizado. Assim, com o avanço tecnológico dos processadores, o algoritmo se tornará cada vez mais eficiente, enquanto que algoritmos com maior consumo de memória poderão ser severamente limitados pela latência.

Dentre os algoritmos sequenciais apresentados, os que se mostram melhores candidatos à paralelização em GPU são o Balsub e de Yanasse & Soma, sendo este último já estudado teoricamente por (SANCHES; SOMA; YANASSE, 2008) em máquinas PRAM.

Neste trabalho, foi estudada a paralelização em GPU de uma recursão baseada no algoritmo de Yanasse & Soma, pelas seguintes razões:

- Este algoritmo possui um comportamento mais próximo da programação dinâmica de Bellman, onde existem mais estudos e otimizações que podem ser utilizados.
- O Balsub apresenta mais etapas de computação dependentes, o que pode ocasionar perda de desempenho devido ao desbalanceamento de tarefas entre as *threads*.
- Conforme será visto no próximo capítulo (**Seção 4.1**), há uma otimização que permite limitar a capacidade c analisada pelo algoritmo de Yanasse & Soma em $c \leq \sum_{i=1}^{n} \frac{w_i}{2}$, o que ocasiona diminuição do espaço gasto, enquanto que o algoritmo Balsub permanecerá com espaço $O(nw_{\max} + n)$, independente do valor de c.

Por outro lado, o algoritmo de Yanasse & Soma possui algumas características que causam perda de desempenho em implementações para GPU, como padrão de acesso à memória não coalescente e menor grau de paralelismo em relação ao algoritmo de Bell-

man. Além disto, sem o aumento do consumo de memória, dificulta a implementação da otimização de Toth.

Isto se torna um grande problema para a paralelização em GPU, principalmente pelo motivo do padrão de acesso à memória, que é um fato citado com destaque na Seção 6.2.1 do manual *CUDA C Best Practices Guide* da própria empresa:

> "... Perhaps the single most important performance consideration in programming for CUDA-capable GPU architectures is the coalescing of global memory accesses. ..."

Por este motivo, ao invés de propor complexas estratégias de paralelismo, foi preferido criar novos algoritmos, baseados nos anteriores, que mantenham as características descritas acima, com a finalidade de tornar sua paralelização em GPU mais fácil e eficiente.

3.5.1 Uma nova recursão

Os algoritmos propostos são baseados no algoritmo de Yanasse & Soma e podem ser definidos formalmente pela recursão $g_i(k)$, que computa os valores do vetor g de Yanasse & Soma em cada etapa i, onde $0 \leq i \leq n$ e $0 \leq k \leq c$. Considerando que o vetor g esteja inicializado com $n+1$ em todas suas posições, uma recursão pode ser definida como segue:

$$g_i(k) = \begin{cases} \min\{t\} & i = 0 \wedge w_t = k, 1 \leq t \leq n \\ i & i > 0 \wedge k \geq w_i \wedge g_{i-1}(k) > i \wedge g_{i-1}(k - w_i) < i \\ g_{i-1}(k) & \textit{caso contrário} \end{cases}$$

Durante a computação da capacidade k da mochila nas etapas $i > 0$, o vetor g será escrito somente se esta capacidade ainda não foi preenchida ($g_{i-1}(k) > i$), e se a capacidade

complementar $k - w_i$ foi preenchida com itens de etapas anteriores ($g_{i-1}(k - w_i) < i$).

A condição $g_{i-1}(k) > i$ de escrita na posição k do vetor g é uma vantagem interessante, já que permite apenas uma única escrita nesta posição quando não é realizada a inicialização, pois caso seja escrita na etapa j, então $g[k]$ receberá o valor j, e nenhum dos próximos passos poderá satisfazer esta condição, o que reduz a quantidade de transações no barramento de memória. Em outras palavras, uma vez preenchida a capacidade k do vetor g, como ela não pode sofrer alterações, nenhuma outra computação sobre esta capacidade será necessária.

Outra vantagem desta recursão é que flexibiliza a paralelização em GPU, permitindo realizar as computações de g pelas colunas ou linhas, ou seja, em n passos de c capacidades ou em c passos de n itens.

Caso se opte pela computação em n passos de c capacidades, em cada passo i, tanto as capacidades analisadas ($g[k], g[k+1], ..., g[n]$), quanto seus respectivos complementos ($g[k-w_i], g[k+1-w_i], ..., g[n-w_i]$), possuirão um padrão de acesso coalescente à memória. Além disto, poupa-se a leitura dos pesos, uma vez que apenas w_i é necessário neste passo.

Por outro lado, caso se deseje computar os c passos de n itens, em cada passo k poupam-se acessos ao vetor g, pois apenas $g[k]$ será necessário, mas seus complementos $g[k-w_i], g[k-w_{i+1}], ..., g[k-w_n]$ não possuirão um padrão de acesso coalescente neste caso. A leitura dos pesos $w_i, w_{i+1}, ..., w_n$ será realizada de forma coalescente. Em contrapartida, nesta computação, uma vez encontrado o valor de $g[k]$, o algoritmo pode passar para a próxima etapa $g[k+1]$, enquanto que o método anterior necessitará acessar $g[k+1]$ para cada iteração $i \leq n$. Uma vez computado o vetor g, a recuperação do vetor X pode ser realizada de modo análogo ao do algoritmo de Yanasse & Soma (linhas 26-34 do **Algoritmo 7**).

4 Algumas otimizações algorítmicas

Neste capítulo são discutidas algumas otimizações testadas nos algoritmos implementados, que apresentaram melhoras no tempo de execução, tanto do ponto de vista prático quanto teórico.

4.1 Otimização de capacidade

Esta é uma simples otimização apresentada em (O'NEIL; KERLIN, 2010) que limita a capacidade da mochila c em $\frac{1}{2}\sum_{i=1}^{n} w_i$, otimizando instâncias quando o valor de c estiver próximo da soma de todos os pesos.

Se existir um subconjunto solução $S \subset A$ com soma de pesos $s = \sum_{\forall a_i \in S} w_i$, existirá um subconjunto remanescente $R = A - S$ com soma de pesos $r = \sum_{i=1}^{n} w_i - s$. Se este subconjunto R for mais fácil de encontrar, *i.e.*, se $r < s$, então seria melhor computá-lo antes e em seguida realizar seu complemento sobre A. Este caso pode ser identificado de uma forma muito simples: se $c > \frac{1}{2}\sum_{i=1}^{n} w_i$, então a computação prosseguirá com a solução complementar $\sum_{i=1}^{n} w_i - c$. No final, o subconjunto solução S será calculado a partir do remanescente.

O **Algoritmo 8** apresenta uma implementação desta estratégia, onde $X[i]$ é o valor

Algoritmo 8 Otimização de limitação de capacidade

```
1: // Verifica a menor capacidade: c ou r
2: mochila ← min{c, Σ_{i=1}^{n} w_i/2}
3: // Computar problema e encontrar vetor solução X
4: // Realizar o complemento, caso r tenha sido escolhido
5: if mochila = r then
6:     for i = 1; i ≤ n; i ← i + 1 do
7:         X[i] =!X[i]
8:     end for
9: end if
```

binário que indica se o i-ésimo item está ou não incluído na solução. Na linha 2 a menor capacidade é escolhida para computação e, após a computação da solução, caso tenha sido escolhida a solução complementar r, é realizado o complemento do vetor solução X (linha 7), com o auxílio do operador binário de negação "!".

Entretanto, esta otimização somente funcionará se houver ao menos uma solução ótima do SSP ou do KP01 com valor c; caso contrário, não haverá solução complementar. Neste trabalho, é proposto uma alteração desta otimização que permite seu uso para encontrar soluções ótimas quando estas forem menores que a capacidade da mochila.

Se a capacidade remanescente r for escolhida para ser computada e não existir uma solução exata, *i.e.*, se $s \neq c$, então a solução ótima $s = c - \delta$ poderá ser encontrada pesquisando a capacidade remanescente ótima $r + \delta$.

Como $r + \delta > r$ pode ser encontrada adicionando um único item às capacidades previamente já computadas, a primeira combinação encontrada a partir de $r + 1$ será esta capacidade ótima remanescente, ou seja, será a menor capacidade maior que r. Consequentemente, seu complemento será a maior capacidade menor que c, sendo, portanto, a solução ótima não exata.

Isto pode ser feito com os pesos ordenados em ordem decrescente, adicionando o menor

Algoritmo 9 Recuperação da solução ótima $r + \delta$

```
1:  // Computa as combinações complementares de r − 1 à r − w_max
2:  k ← r − 1
3:  // Inicializa solução ótima potencial
4:  s ← r + w_max
5:  // Índice do item adicionado, referente a solução s
6:  adicionar ← n + 1

7:  // Verifica todas os n pesos
8:  for i ← n; i ≥ 1 ∧ k + w_max > r; do
9:      // Soluções apenas maiores que r
10:     while k + w_i < r ∧ i ≥ 1 do
11:         i ← i − 1
12:     end while

13:     // Pesquisa soluções candidatas
14:     for ; k + w_i > r ∧ k ≥ 1; k ← k − 1 do
15:         if g[k] < i ∧ k + w_i < s then
16:             // Altualiza melhor solução encontrada
17:             s ← k + w_i
18:             adicionar ← i
19:         end if
20:     end for
21: end for
```

item i às soluções previamente computadas k, desde que $k + w_i > r$. Partindo do menor item $i = n$, são computadas capacidades $r - w_{i+1} > k > r - w_i$, seguindo, assim por diante, com os demais itens. Esta implementação é apresentada no **Algoritmo 9**.

No laço das linhas 8 à 21 até $w_{\max}$ combinações complementares são analisadas, pois o maior item que pode ser adicionado à estas combinações é o item com peso $w_{\max}$. Nas linhas 10-12 são ignorados os pesos que somados a qualquer combinação complementar, resulte em uma solução não potencial, ou seja, menor que r.

As novas soluções encontradas $k + w_i > r$ são verificadas nas linhas 15-19, e caso sejam mais próximas de r, as variáveis s e *adicionar* são atualizadas. Ao final da computação, a variável s será igual à solução ótima $r + \delta$ e *adicionar* conterá o índice do item que deverá ser adicionado na combinação complementar para se encontrar a solução ótima, ou seja,

a combinação complementar será $s - w_{adicionar}$.

Uma forma de reduzir a quantidade de computações é limitar a pesquisa em $\delta \leq w_{\max}$ e $r + \delta < c$. No último caso, se $r + \delta \geq c$, basta trocar a pesquisa de $r + \delta$ por c. Desta forma, a complexidade final desta pesquisa terá tempo $O(w_{\max})$.

4.2 Otimização de limitação

Conforme visto na **Seção 3.2**, é possível aumentar o limite inicial de cada item i para $k_{i\,\min}$ através da otimização de Toth. Utilizando esta mesma ideia, a máxima capacidade possível de se alcançar durante a etapa i também pode ser limitada para $k_{i\,\max} = \min\{c, \sum_{j=1}^{i} w_j\}$.

Embora esta última limitação não restrinja o algoritmo, quanto à sua capacidade de encontrar soluções ótimas, a otimização de Toth o faz. A seguir é apresenta uma forma de contornar este problema.

Quando implementada esta otimização em conjunto com a otimização de capacidade, dois casos podem ocorrer: considerar a capacidade da mochila c ou a remanescente $r + \delta$.

Nos casos em que a capacidade remanescente for escolhida, basta seguir o mesmo procedimento citado na seção anterior, uma vez que a solução ótima $r + \delta$ implicará em um limite inferior maior ou igual ao $k_{i\,\min}$ da solução r.

Se a capacidade da mochila for escolhida, então pode-se utilizar duas abordagens durante o cálculo da capacidade inicial $k_{i\,\min}$. Ao invés de utilizar c na otimização de Toth, este valor será trocado pela maior capacidade encontrada nas etapas anteriores ou durante a verificação das soluções triviais (cf. **Seção 4.4**).

4.3 Otimização de ordenação

Ordenar os itens também pode acelerar a computação, especialmente quando utilizada em conjunto com a otimização de limitação, além de outros benefícios citados na seção seguinte. A ordenação decrescente dos pesos aumenta a soma $\sum_{j=i+1}^{n} w_j$ em cada etapa i, fazendo com que $k_{i\,\min}$ seja maior, porém também aumenta a capacidade máxima $k_{i\,\max}$.

Por isto, é importante analisar o conjunto de itens e decidir quando esta otimização poderá ser realmente útil. Nos casos em que, com alguns itens computados, a soma dos restantes seja próxima de c, esta otimização pode ser vantajosa porque, a cada novo passo, $k_{i\,\min}$ estará mais próximo do valor c.

O limite mínimo $k_{i\,\min}$ também pode ser melhorado no caso da ordenação decrescente, ou seja, quando $w_{i-1} \geq w_i$. Neste caso, a menor combinação computada nos primeiros i passos será a combinação w_i e a segunda menor combinação do peso w_i, no passo i, será $w_i + w_{i-1}$, uma vez que a combinação w_{i-1} é a menor dos primeiros $i-1$ passos. Assim, utilizando a inicialização apresentada no **Algoritmo 7** (linhas 2-8), é possível definir $k_{i\,\min} \geq w_i + w_{i-1}$, ou seja, $k_{i\,\min} = \max\{w_i + w_{i-1}, c - \sum_{j=i+1}^{n} w_j\}$.

4.4 Verificação de soluções triviais

Durante os testes realizados, foi observado que muitas instâncias triviais podem ser rapidamente solucionadas por simples heurísticas. Uma muito útil para as instâncias testadas (cf. **Capítulo 6**), também presente no algoritmo de Boyer, Baz & Elkihel, é a de coletar os pesos sequencialmente até que sua soma atinja o valor desejado c, como apresentado na **Algoritmo 10**. Se, em algum momento, esta soma exceder a capacidade

da mochila (*i.e.*, $\sum_{i=1}^{t} w_i > c$), então o último item t será ignorado (linha 6), continuando o processo a partir do próximo item.

Algoritmo 10 Verificação de solução trivial

```
1: // Inicia solução
2: s ← 0
3: // Verifica todas os n pesos
4: for i ← 1; i ≤ n; i ← i + 1 do
5:     // Adiciona se não estourar a mochila
6:     if s + w_i ≤ c then
7:         s ← s + w_i
8:     end if
9: end for
```

Esta heurística funciona melhor em conjunto com a otimização de capacidade (**Seção 4.1**) e com pesos ordenados em ordem decrescente (**Seção 4.3**). Nesta situação, uma boa parte da capacidade da mochila será preenchida rapidamente pelos itens mais pesados, deixando o ajuste fino para itens de peso menor.

5 Implementações paralelas

Neste capítulo, são analisadas as principais estratégias de paralelização em GPU para a recursão proposta (**Subseção 3.5.1**), e três implementações em CUDA são apresentadas. Embora os códigos fonte não sejam disponibilizados como anexo, devido ao tamanho, eles podem ser encontrados no endereço `https://bitbucket.org/vitorcurtis/subset-sum-on-gpu`.

5.1 Introdução

Esta seção apresenta uma implementação sequencial simplificada da recursão proposta e um estudo das principais formas gerais de paralelização, que serão discutidas nas seções subsequentes. Porém, antes de entrar nos detalhes de paralelização, as últimas subseções apresentam as principais características referentes à GPU escolhida, que devem ser levadas em conta durante a análise e implementação das soluções paralelas.

5.1.1 Análise da recursão proposta

O **Algoritmo 11** apresenta uma implementação sequencial da recursão proposta, sem nenhuma otimização, utilizando dois laços aninhados: um percorrendo os itens e outro as

capacidades, assim como os algoritmos baseados no paradigma de Bellman.

Algoritmo 11 Algoritmo sequencial da recursão proposta

```
1:  // Inicialização do vetor g

2:  // Percorrendo toda capacidade k com o item i
3:  for i ← 1; i ≤ n; i ← i + 1 do
4:      for k ← 1; k ≤ c; k ← k + 1 do

5:          // Condição de escrita
6:          if k ≥ w_i ∧ g[k] > i ∧ g[k − w_i] < i then
7:              g[k] ← i
8:          end if
9:      end for
10: end for

11: // Recuperação do vetor X
```

Como dito na **Subseção 3.5.1**, os laços aninhados (linhas 3 e 4) são permutáveis. Devido a isto, uma forma simples de paralelizar este algoritmo seria escolher percorrer os itens ou as capacidades no laço interno, e paralelizar apenas este laço. Com o objetivo de obter alto grau de paralelismo, a melhor escolha seria a de paralelizar o laço que varia sobre as capacidades (passaria a ser o laço interno), já que para os casos não triviais é comum $n \ll c$. Assim, ao término da computação de um item, uma sincronização deve ser realizada antes de iniciar o próximo item.

Nesta abordagem, em uma PRAM SIMD com $1 \leq p \leq \max\{n, c\}$ processadores, a complexidade temporal para a computação do vetor g seria de $O(\frac{nc}{p})$, mantendo o espaço $O(n + m + w_{\min})$, o que resulta em uma melhora do *upperbound* de tempo, pois bastaria paralelizar o laço interno.

Embora o vetor g contenha a solução para todas as possíveis capacidades menores ou igual a c, vale ressaltar que a recuperação do vetor X para qualquer capacidade encontrada não é feita em paralelo por ser uma tarefa inerentemente sequencial.

Caso opte-se por paralelizar o laço que varia sobre os n pesos, apenas n *threads* poderão computar paralelamente, comprometendo a eficiência da paralelização, já que $n \ll c$.

Uma terceira abordagem seria a de mesclar ambas as estratégias, computando apenas uma parcela dos itens ou capacidades. Por exemplo, variando os n itens no laço interno e paralelizando um grupo de $t \leq c$ capacidades com t *threads*. Assim, a cada n passos, t novas capacidades estariam computadas.

Dentre estas estratégias, foram analisadas apenas a primeira, com grau de paralelismo igual a c, e a última, com $t \leq c$ *threads* variando tanto os itens quanto as capacidades. A segunda abordagem foi abandonada, devido a limitação do paralelismo.

Devido ao fato de ambas estratégias paralelizarem as capacidades, todas as otimizações do **Capítulo 4** podem ser utilizadas. No início da computação, é realizada a ordenação dos pesos, com tempo $O(n \log n)$, seguida da verificação das soluções triviais (**Seção 4.4**), gastando tempo $O(n)$. Durante esta fase, a soma dos pesos é computada para o cálculo da capacidade a ser pesquisada, através da otimização de capacidade (**Seção 4.1**). Esta otimização limitará o tamanho do vetor g em $m = \min\{c, \sum_{i=1}^{n} w_i - c\}$.

Após alocar o vetor g, sua inicialização é realizada como nas linhas 2-8 do **Algoritmo 7** (**Seção 3.3**), gastando tempo $O(m)$ para definir seus elementos com o valor $n+1$ e $O(n)$ para as soluções de um único item. Com a inicialização do vetor g, não é necessário iniciar a computação dos laços aninhados a partir do primeiro item, pois não existirá combinações. Isto torna possível computar o vetor g em até $(n-1)m$ iterações.

A quantidade de iterações pode ser reduzida ainda mais, levando em conta a otimização de limitação (**Seção 4.1**), pois apenas as capacidades de $w_{i-1} + w_i$ a $\sum_{j=1}^{i} w_j$ são computadas em cada iteração i. Como $w_i \geq w_{\min}$ e a faixa de capacidades computadas

na iteração $i = 2$ é de $w_1 + w_2$ a $w_1 + w_2$, no máximo $(n-2)(m-2w_{\min})+1$ iterações serão computadas.

Caso a solução ótima não tenha sido encontrada após a computação do vetor g, o método apresentado na **Seção 4.1** deve ser utilizado, gastando tempo $O(w_{\max})$. Com a solução ótima encontrada, o vetor X é então recuperado, e caso a capacidade da mochila tenha sido trocada pela otimização de capacidade, seu complemento é computado, com tempo $O(n)$.

Somando todas as etapas anteriores, chega-se a complexidade temporal de $O(n \log n + m + (n-2)(m-2w_{\min}))$, entretanto, como a constante multiplicativa de $O(m)$, do laço que atribui o valor $n+1$ para os m elementos do vetor g, é menor ou igual à dos laços aninhados, pode-se reescrevê-la como $O(n \log n + (n-1)(m-2w_{\min}) + 2w_{\min})$, resultando nas complexidades temporal $O(n \log n + (n-1)(m-2w_{\min}) + w_{\min})$ e espacial $O(n+m-w_{\min})$.

5.1.2 Latência da memória

As implementações realizadas neste trabalho foram desenvolvidas para a arquitetura Fermi; mais especificamente, para a GPU NVIDIA GeForce GTX 560 com versão de capacidade 2.1. A versão de capacidade de uma GPU NVIDIA descreve uma coleção de recursos de *hardware* disponíveis, onde as mais recentes possuem versão de capacidade igual a 3.5.

Nas versões anteriores à capacidade 2.0, apenas uma *shared memory* de 16KB que poderia ser utilizada como memória *on-chip* para computação, porém, a partir desta versão, as GPUs passaram a ter acesso à memória global servidas por *cache* de 2 níveis

(L1 e L2), além de outros recursos e melhorias importantes, como mais do que o dobro de *cores* em relação a versão anterior (NVIDIA, 2009).

A escolha desta versão de GPU se deve ao fato acima, e por não dispor de equipamentos mais recentes durante o período de pesquisa.

Deste modo, similar aos outros algoritmos de programação dinâmica para o SSP, é possível verificar a existência de apenas dois testes principais (linha 6 do **Algoritmo 11**) para computar cada preenchimento e somente uma subtração de inteiros para o cálculo do índice da combinação complementar. Nenhum cálculo de ponto flutuante ou operações complexas são necessárias. Como a recursão proposta possui poucos cálculos e muitas verificações que necessitam de transações na memória, a falta de uma hierarquia de *cache* poderia tornar a implementação em CUDA muito ineficiente, pois, segundo (NVIDIA, 2012b), a latência para leituras na memória global varia de 400 à 800 ciclos para GPUs com capacidade 1.x ou 2.x.

Em (NVIDIA, 2012b), é descrito que para esconder a latência de L ciclos de máquina para GPUs com capacidade 2.1, são necessárias $2L$ instruções, considerando uma média de 22 ciclos por instrução. Por exemplo, se fosse levado em conta uma razão de 10 instruções com operandos de memória *on-chip* para 2 instruções com operandos de memória *off-chip*, ou seja, uma razão de 5:1, para esconder uma latência de acesso à memória global de 600 ciclos, seriam necessários $600/5 = 120$ *warps ativos* por SM.

As GPUs de capacidade 2.x possuem um limite máximo de 48 *warps* ativas por SM, o que pode tornar a paralelização um desafio nos casos descritos acima. Sendo assim, a paralelização realizada deve buscar ao máximo amenizar o problema da pouca quantidade de instruções com operandos de memória *off-chip* (linhas 6 e 7 do **Algoritmo 11**).

5.1.3 Configuração de *threads* e blocos

Nos algoritmos apresentados neste capítulo, as variáveis $threads_{\max}$ e $blocks_{\max}$ são utilizadas para representar, respectivamente, a quantidade máxima de *threads* ativas e blocos ativos. Embora seja possível chamar um *kernel* com mais blocos, foi preferido utilizar apenas blocos e *threads* ativos por conveniência.

Estes valores são específicos para cada modelo de GPU e podem ser calculados utilizando a *Calculadora de Ocupação da GPU CUDA* (NVIDIA, 2012a), presente no CUDA SDK. De acordo com esta ferramenta, as GPUs com capacidade 2.1 podem ter no máximo 48 *warps* ativas e 8 blocos ativos por SM.

Desta forma, para iniciar um *kernel* com a quantidade máxima e exata de *threads* ativos, é necessário garantir, por SM, que o número de blocos não exceda 8 e que a quantidade final de *warps* seja 48. Com este objetivo, neste trabalho, foi definida a quantidade de 768 *threads* por bloco, ou seja, 24 *warps* por bloco e 2 blocos por SM.

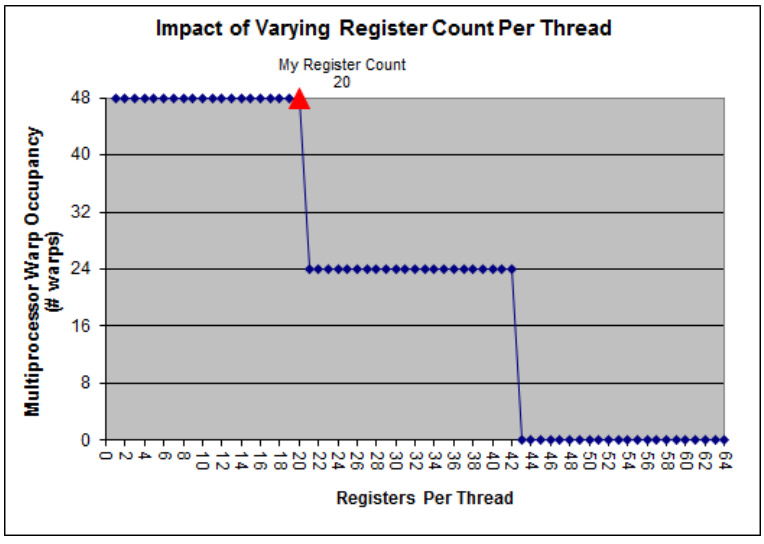

FIGURA 5.1 – Warps ativos por registradores/thread.

Como a GPU utilizada (NVIDIA GeForce GTX 560) possui 7 SMs de 48 *cores* cada, no total serão 7 x 2 x 24 *warps* ativas (10.752 *threads* ativas) sendo computadas paralelamente

por 336 *cores*.

A quantidade de registradores utilizados por *thread* é um fator importante que deve ser levado em conta nesta configuração, pois pode limitar a quantidade de *threads* ativas, não utilizando a capacidade máxima da GPU desta forma: conforme a **Figura 5.1**, para GPUs de capacidade 2.1, cada *kernel* deve utilizar no máximo 20 registradores por *thread*, para que cada SM tenha até 48 *warps* ativas.

5.2 Implementação 1: variando c

Nesta seção é apresentada a paralelização em que as capacidades variam no laço mais interno, enquanto o laço externo, que percorre os itens, é computado na CPU, realizando uma chamada ao *kernel* a cada item computado.

5.2.1 Estratégia adotada

Esta estratégia possui a vantagem de permitir que todas as capacidades sejam computadas em paralelo para cada item. Apenas uma sincronização é realizada, entre as computações dos itens, para garantir que todas as *threads* escrevam seus resultados antes da computação do próximo item, porém, como cada item é computado através de uma chamada ao *kernel*, este sincronismo é realizado de forma implícita pela plataforma CUDA.

Devido ao fato do *kernel* ser chamado com apenas 768 *threads* por bloco, como dito na **Subseção 5.1.3**, um laço deve ser utilizado para percorrer todas as capacidades necessárias. Isto pode ser feito de forma simples, tornando o incremento do laço igual ao número de *threads* no *kernel*.

Desta forma, os acessos à memória, tanto de $g[k - w_i]$ como de $g[k]$, são coalescentes, atendendo a observação do manual *CUDA C Best Practices Guide*, dito na **Seção 3.5**.

Entretanto, para cada item i e capacidade k, uma leitura seria realizada para verificar o valor de $g[k - w_i]$ e, caso fosse menor do que i, uma leitura de $g[k]$ seria realizada para verificar a condição de escrita, resultando em uma aproximação de duas leituras coalescentes à memória para cada capacidade computada. As escritas podem ser ignoradas uma vez que representam aproximadamente apenas até $\frac{1}{n}$ das leituras.

Os principais problemas desta estratégia são a pouca quantidade de instruções independentes e o grande volume de requisições à memória por capacidade computada, mesmo possuindo um padrão de acesso coalescente à memória.

5.2.2 Pseudocódigo da CPU

A CPU será responsável por computar as otimizações (**Capítulo 4**), inicializar o vetor g (linhas 2-8 do **Algoritmo 7** da **Seção 3.3**), transferir este vetor para a GPU e executar o laço que chama o *kernel* com os parâmetros apropriados. Deste modo, a CPU calcula os índices $k_{i\,\min}$ e $k_{i\,\max}$ (**Seção 4.2**), que limitam o intervalo de capacidades para cada peso w_i, e define a capacidade c da mochila a ser pesquisada (**Seção 4.1**).

No final da computação, a CPU transfere o vetor g já calculado da GPU para sua memória e recupera o vetor X.

Neste pseudocódigo, apresentado no **Algoritmo 12**, as linhas 7-9 são referentes a configuração de *threads* e blocos da **Subseção 5.1.3**; as linhas 16-20 são responsáveis por alocar menos blocos ativos, caso o máximo não seja necessário; e a linha 22 invoca o *kernel* com a configuração de *threads* resultante. Os parâmetros da invocação correspondem,

Algoritmo 12 Pseudocódigo da CPU para o *kernel* 1

1: // Ordenação decrescente dos pesos
2: // Otimização da capacidade
3: // Verificação de soluções triviais
4: // Inicialização de g
5: // Cópia de g da CPU para a GPU

6: // Configuração de *threads* e blocos
7: $qtd_{SMs} \leftarrow 7$
8: $blocks_{\max} \leftarrow 2 * qtd_{SMs}$
9: $threads_{\max} \leftarrow 768$

10: // Laço externo percorre todos os itens
11: **for** $i \leftarrow 2;\ i \leq n;\ i \leftarrow i + 1$ **do**
12: // Otimização de limitação com ordenação dos pesos
13: $k_{i\,\min} \leftarrow \max\{w_i + w_{i-1}, c - \sum_{j=i+1}^{n} w_j\}$
14: $k_{i\,\max} \leftarrow \min(c,\ \sum_{j=1}^{i} w_j)$

15: // Quantidade de blocos necessários
16: $blocks \leftarrow \lceil \frac{k_{i\,\max} - k_{i\,\min} + 1}{threads_{\max}} \rceil$
17: // Limitar para somente blocos ativos
18: **if** $blocks > blocks_{\max}$ **then**
19: $blocks \leftarrow blocks_{\max}$
20: **end if**

21: // Chamada do *kernel*
22: kernel1 $\lll \text{blocks}, \text{threads}_{\max} \ggg (i, k_{i\,\min}, k_{i\,\max}, w_i, g)$
23: **end for**

24: // Cópia de g, da GPU para a CPU
25: // Recuperação da solução ótima X de g

respectivamente, ao índice i do item analisado, às capacidades mínima e máxima $k_{i\,\min}$ e $k_{i\,\max}$, ao peso w_i do item i e a um ponteiro para o vetor g na memória global da GPU.

No final da computação, a CPU recupera o vetor X (linha 25) como descrito nas seções 4.1 e 4.2 do **Capítulo 4**. Também é importante notar que o laço (linha 11) inicia no passo 2, pois o primeiro item já é inserido em g durante a inicialização realizada pela CPU (linha 4).

5.2.3 Pseudocódigo do *kernel*

No **Algoritmo 13**, é apresentado o pseudocódigo do *kernel* 1, implementado com um laço que varia por todas as capacidades delimitadas por $k_{i\,\min}$ e $k_{i\,\max}$ (linha 12).

Algoritmo 13 Pseudocódigo do *kernel* 1

```
1:  i: Índice do item analisado
2:  k_{i min}: Menor índice da otimização de limitação
3:  k_{i max}: Maior índice da otimização de limitação
4:  w_i: Peso do item i
5:  g: Ponteiro para o vetor g

6:  function KERNEL1(i, k_{i min}, k_{i max}, w_i, g)
7:      // Identificador único da thread na grid
8:      tid ← blockIdx.x * blockDim.x + threadIdx.x
9:      // Quantidades de threads na grid
10:     inc ← gridDim.x * blockDim.x

11:     // Percorre todas as capacidades
12:     for k ← k_{i min} + tid; k ≤ k_{i max}; k ← k + inc do
13:         // Condição de escrita
14:         if g[k] > i ∧ g[k − w_i] < i then
15:             g[k] ← i
16:         end if
17:     end for
18: end function
```

Os cálculos das linhas 8 e 10 são realizados como comentado na **Subseção 2.1.3**. No laço, cada *thread* adiciona sua identificação ao índice $k_{i\,\min}$ (linha 12), fazendo com que a *thread* com identificação *tid* inicie computando a *tid*-ésima capacidade a partir de $k_{i\,\min}$ (a capacidade $k_{i\,\min}$ é computada pela *thread* com $tid = 0$), e a condição de escrita é realizada no interior do laço, como apresentado na **Subseção 5.1.1**.

5.2.4 Alternativas de implementação

O uso da *shared memory* para compartilhar dados entre as *threads* de um mesmo bloco não é viável nesta estratégia, pois cada *thread* computa um valor distinto do vetor

g. A única situação em que ocorre leitura de uma mesma posição é quando uma *thread* t_1 computa $g[k_1]$ e uma *thread* t_2 computa $g[k_2]$, com $k_2 - w_i = k_1$. Neste caso, as *threads* podem compartilhar o valor de $g[k_1]$.

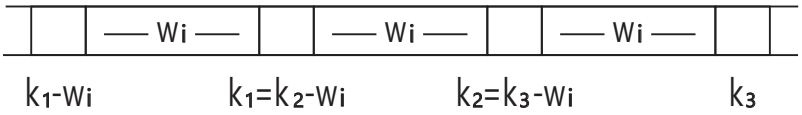

FIGURA 5.2 – Dados compartilhados entre w_i capacidades.

Uma forma simples de tirar vantagem desta estratégia é alterar o valor do incremento do laço interno para w_i, ao invés de utilizar a quantidade de *threads* no *kernel*. Neste caso, cada capacidade computada pelo *kernel* será, na próxima iteração, a combinação complementar, conforme mostra a **Figura 5.2**. Armazenando os valores em registradores, basta realizar apenas uma transição de memória a cada passo: aquela referente à próxima combinação.

Neste caso, a execução paralela de várias chamadas ao *kernel* é necessária quando w_i for maior do que a quantidade de *threads*. Caso contrário, a computação deverá ser distribuída entre as *threads* disponíveis, para que não fique limitada a apenas w_i *threads*.

Esta estratégia foi implementada durante a pesquisa, porém não resultou em ganho de desempenho para a maioria dos testes realizados (**Seção 6.1**), mesmo embora a quantidade de transações à memória tenha sido reduzida, segundo a ferramenta *Visual Profiler* da NVIDIA.

Outra melhoria analisada foi o uso de *unrolling* no laço interno, com o objetivo de aumentar a quantidade de instruções independentes, permitindo ao compilador agrupar todas as requisições de memória em registradores e depois as utilizar nos testes condicionais. Esta estratégia resultou no aumento do tempo de execução, sendo que menos de 20

registradores foram utilizados (**Subseção 5.1.3**).

5.3 Implementação 2: variando n e t

Nesta seção é apresentada uma paralelização em que um grupo de $t \leq c$ *threads* computam os n itens, antes de iniciar a computação das próximas capacidades.

5.3.1 Estratégia adotada

Nesta estratégia, $t \leq c$ *threads* são distribuídas entre as capacidades, devido ao baixo grau de paralelismo dos itens, já que $n \ll c$, e o laço interno percorre os n itens. Desta forma, cada *thread* poderá terminar a computação de uma capacidade precocemente ao encontrar a combinação, evitando transações de memória e a computação dos n passos, como dito na **Subseção 5.1.1**.

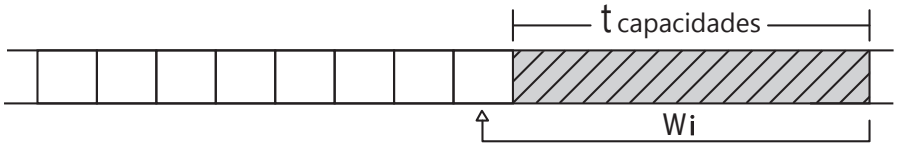

FIGURA 5.3 – Limite do valor os pesos.

Uma limitação existente, é que podem ser computados somente pesos iguais ou maiores do que t, já que as t capacidades estarão sendo computadas, como mostra a **Figura 5.3**. Desta forma, quanto maior for t ($SMs * 2 * 768$, **Subseção 5.1.3**), menos itens poderão ser computados. Os demais itens deverão ser computados através de outro método.

Ao final, antes de um bloco computar as próximas capacidades, todas as anteriores deverão estar concluídas para que possam ser utilizadas como combinações complementares, exigindo assim uma sincronização entre blocos. Embora a sincronização implícita

entre as chamadas ao *kernel* possa ser utilizada, foi preferido optar pelo método apresentado na **Subseção 2.1.6.1**, pelo fato de haver desbalanceamento de tarefas, uma vez que as *threads* podem terminar sua computação antecipadamente, permitindo à um bloco computar as próximas capacidades logo assim que as anteriores tenham sido terminadas. Este procedimento é seguido até que se chegue à capacidade da mochila.

Embora este método apresente a necessidade de muitas sincronizações entre blocos, elas somente serão realizadas após a computação de todos os pesos. Levando em conta a redução das transações de memória e computação, esta estratégia pode se tornar ainda mais promissora no futuro, com a implementação em *hardware* do sincronismo entre blocos.

Outra vantagem desta estratégia é a possibilidade do uso de texturas para leituras dos pesos (**Subseção 2.1.5**), uma vez que não sofrem escritas. Como os pesos são dados de 32 *bits*, em uma instância com 1.000 itens, até 4.000 *bytes* seriam poupados dos *caches* L1 e L2, além de liberar o barramento de memória das diversas requisições realizadas por cada *thread*.

5.3.2 Pseudocódigo da CPU

O código da CPU para invocar este *kernel* é o menor, uma vez que toda a computação é realizada na GPU. O pseudocódigo é apresentado no **Algoritmo 14**.

Duas novas inicializações são requeridas, em relação à implementação anterior: na linha 6 a textura é inicializada com os pesos (**Subseção 2.1.5**), e na linha 7, o vetor *Clock* é inicializado com o valor 1 (**Subseção 2.1.6.1**).

A chamada ao *kernel* é realizada na linha 13, com dois parâmetros diferentes: um

Algoritmo 14 O pseudocódigo da CPU para o *kernel* 2

1: // Ordenação decrescente dos pesos
2: // Otimização da capacidade
3: // Verificação de soluções triviais
4: // Inicialização de g
5: // Cópia de g da CPU para a GPU
6: // Inicializar textura tex
7: // Inicializar $Clock$

8: // Configuração de *threads* e blocos
9: $qtd_{SMs} \leftarrow 7$
10: $blocks_{\max} \leftarrow 2 * qtd_{SMs}$
11: $threads_{\max} \leftarrow 768$

12: // Chamada do *kernel*
13: kernel2 $\lll \text{blocks}_{\max}, \text{threads}_{\max} \ggg (c, n, \sum_{j=1}^{n} w_j, g, Clock)$

14: // Percorre os itens restantes
15: **for** $i \leftarrow 2;\ i \leq n;\ i \leftarrow i + 1$ **do**
16: // Somente computar os itens restantes
17: **if** $w_i < blocks_{\max} * threads_{\max}$ **then**
18: // Otimização de limitação com ordenação dos pesos
19: $k_{i\,\min} \leftarrow \max\{w_i + w_{i-1}, c - \sum_{j=i+1}^{n} w_j\}$
20: $k_{i\,\max} \leftarrow \min(c,\ \sum_{j=1}^{i} w_j)$

21: // Quantidade de blocos necessários
22: $blocks \leftarrow \lceil \frac{k_{i\,\max} - k_{i\,\min} + 1}{threads_{\max}} \rceil$
23: // Limitar para somente blocos ativos
24: **if** $blocks > blocks_{\max}$ **then**
25: $blocks \leftarrow blocks_{\max}$
26: **end if**

27: // Chamada do *kernel*
28: kernel1 $\lll \text{blocks}, \text{threads}_{\max} \ggg (c, i, k_{i\,\min}, k_{i\,\max}, w_i, g)$
29: **end if**
30: **end for**

31: // Cópia de g, da GPU para a CPU
32: // Recuperação da solução ótima X de g

ponteiro para o vetor *Clock* na GPU e a soma de todos os pesos, para o cálculo da otimização de limitação, uma vez que o *kernel* é responsável pela computação do laço externo.

Após a computação, o *kernel* 1 é utilizado para computar os itens restantes (linhas

15-30). O teste que verifica se o item já foi computado é realizado na linha 17.

5.3.3 Pseudocódigo do *kernel*

O pseudocódigo do *kernel* 2 é apresentado em **Algoritmo 15** e **Algoritmo 16**.

Algoritmo 15 Pseudocódigo do kernel 2 - Parte 1

```
1:  c: Capacidade da mochila
2:  n: Quantidade de itens
3:  sumWi: Soma de todos os pesos
4:  g: Ponteiro para o vetor g
5:  Clock: Ponteiro para o vetor Clock inicializado

6:  function KERNEL2(c, n, sumWi, g, Clock)
7:      // Identificador único da thread na grid
8:      tid ← blockIdx.x * blockDim.x + threadIdx.x
9:      // Quantidades de threads na grid
10:     inc ← gridDim.x * blockDim.x

11:     // Menor capacidade computada
12:     k ← tex1Dfetch(tex, n) + tex1Dfetch(tex, n − 1) + tid

13:     //  Laço externo percorre todas as capacidades
14:     for ; k ≤ c; k ← k + inc do

15:         //  Recupera primeiro peso
16:         w_i ← tex1Dfetch(tex, 1)
17:         //  Inicializa a soma dos pesos restantes
18:         sumComp ← sumWi − w_i
19:         //  Inicializa a soma dos pesos
20:         sumWi ← w_i

21:         //  Percorre todos os itens
22:         for i ← 2; i ≤ n; i ← i + 1 do
23:             //  Recupera peso anterior da iteração anterior
24:             w_{i−1} ← w_i
25:             //  Recupera peso atual
26:             w_i ← tex1Dfetch(tex, i)

27:             //  Caso os pesos restantes forem menores que a quantidade de threads,
28:             //  ou a capacidade já tenha sido encontrada na inicialização de g
29:             if w_i < inc ∨ w_i = k then
30:                 //  Continuar na próxima capacidade
31:                 break
32:             end if
```

Na linha 12, a capacidade inicial é calculada como a soma dos últimos pesos $w_n + w_{n-1}$, através da textura tex (**Subseção 2.1.5**), pois os últimos itens são os menores devido à ordenação decrescente.

A computação prossegue no laço externo (linha 14), até chegar à capacidade máxima c. A cada iteração, as variáveis $sumComp$ e $sumWi$ são inicializadas (linhas 18 e 20) para o cálculo da otimização de limitação, representando, respectivamente, a soma dos pesos restantes e a soma dos primeiros i pesos. Como o laço interno (linha 22) inicia no segundo peso, o primeiro w_1, lido na linha 16, deve ser considerado nesta inicialização.

No laço interno da linha 22, a cada iteração, o peso analisado é recuperado (linha 26) através da textura tex e o peso anterior é copiado da iteração anterior (linha 24). Em seguida, a condição da linha 29 verifica se a capacidade computada já foi encontrada na inicialização do vetor g ou se os pesos restantes não podem ser calculados, como descrito na **Subseção 5.3.1**, interrompendo a computação desta capacidade em caso afirmativo.

O **Algoritmo 16** apresenta a continuação do pseudocódigo, onde nas linhas 34 e 35 são atualizadas as variáveis $sumComp$ e $sumWi$, assim como realizado nas linhas 18 e 20, para o cálculo da otimização de limitação (linhas 37 e 38).

A condição da linha 40 salta para a próxima capacidade caso a atual esteja fora dos limites, e a condição de escrita é analisada na linha 44. É importante ressaltar que esta condição não analisa se $g[k] > i$, pois uma vez que a capacidade k é encontrada, ela não é mais analisada, pois a *thread* salta para a próxima capacidade (linha 47). Porém, a cada capacidade computada, o sincronismo entre blocos é realizado na linha 51.

Algoritmo 16 Pseudocódigo do kernel 2 - Parte 2

```
33:             // Atualiza somas
34:             sumComp ← sumComp − w_i
35:             sumWi ← sumWi + w_i
36:             // Otimização de limitação com ordenação dos pesos
37:             k_{i min} = max{w_i + w_{i−1}, c − sumComp}
38:             k_{i max} = min(c, sumWi)

39:             // Computar somente as capacidades necessárias
40:             if k < k_{i min} ∨ k > k_{i max} then
41:                 continue
42:             end if

43:             // Condição de escrita
44:             if g[k − w_i] < i then
45:                 g[k] ← i
46:                 // Salta para próxima capacidade
47:                 break
48:             end if
49:         end for

50:         // Sincronização entre blocos, a cada capacidade computada
51:         __cycleSyncBlocks(Clock)
52:     end for
53: end function
```

5.3.4 Alternativas de implementação

Várias outras abordagens alternativas, variando tanto as capacidades quanto os itens, foram analisadas durante a pesquisa, porém nenhuma se demonstrou mais promissora para implementação.

Uma alternativa interessante é o uso da *shared memory* para compartilhamento de dados entre as *threads* de um bloco, por ser uma memória *on-chip*. Duas abordagens foram analisadas: compartilhar as combinações complementares e as capacidades. Ambos os casos foram abandonados devido a diversos motivos. A seguir, alguns deles são destacados.

Com o compartilhamento das capacidades na *shared memory*, as *threads* estariam distribuídas entre as combinações complementares, uma vez que não faz sentido utilizar

registradores para armazenar os mesmos valores. Neste caso, cada *thread* deveria adicionar os pesos em sua combinação complementar para chegar nas novas capacidades. Isto pode ser feito realizando a seguinte troca de variáveis na recursão proposta: $k' = k - w_i$ (**Subseção 3.5.1**), tornando a computação idêntica ao algoritmo de Yanasse & Soma.

Neste caso, condições de corrida podem ocorrer, uma vez que cada *thread* poderá encontrar qualquer capacidade da *shared memory*, necessitando de sincronismos a cada capacidade computada. Outro problema é que cada *thread* poderá apenas computar novas capacidades que estejam na *shared memory*. Devido a isto, com l capacidades carregadas na *shared memory*, até $\frac{c}{l}$ destas computações podem ser necessárias. Como, geralmente, a quantidade de memória *on-chip* é baixa, ou seja, l é pequeno, o paralelismo poderia ser comprometido.

Com o compartilhamento das combinações complementares na *shared memory*, da mesma forma, as *threads* estariam limitadas a computar somente capacidades e pesos, cujas combinações complementares estejam na *shared memory*, e com l combinações complementares, $\frac{c}{l}$ destas computações seriam necessárias. Outro problema, é que as *threads* ficam limitadas a computar capacidades a $w_{\max}$ posições distantes das combinações da *shared memory*.

5.4 Implementação 3: variando c

Nesta seção, uma paralelização muito próxima da implementação 1 (**Seção 5.2**) é apresentada, porém, ao invés da forma tradicional, a recursão proposta é computada através de um vetor binário.

5.4.1 Estratégia adotada

Caso não seja necessário retornar o vetor solução X, é possível computar a recursão proposta com um vetor binário. Ao variar as capacidades no laço mais interno, na etapa i, apenas pesos w_{i-1} podem estar adicionados ao vetor g. Independente da configuração dos índices no vetor, apenas é necessário saber se existe ou não um preenchimento em determinada combinação complementar $k - w_i$ para que seja possível realizar a combinação k com o peso w_i, ou seja, é possível utilizar um vetor binário para representar esta condição.

Desta forma, como são necessários pelo menos $\log_2(n)$ *bits* para representar os índices do vetor g, as leituras da memória poderão ser reduzidas até uma fração de $\frac{1}{\log_2(n)}$, em relação ao método tradicional (**Subseção 5.1.1**). É importante citar que, nesta computação, não é possível realizar a inicialização do vetor binário, sendo necessária a computação da combinação w_i somente durante o passo i.

Contudo, ainda é possível recuperar o vetor solução X. Isto é feito utilizando o vetor binário em conjunto com o vetor g: durante a computação do passo i, caso seja encontrada uma combinação k, além de setar o *bit* correspondente no vetor binário, a posição k do vetor g é definida com o índice i.

Na abordagem paralela, além do vetor g, dois vetores binários de c *bits* podem ser utilizados. Isto é necessário, pois caso uma *thread* encontre uma combinação k no passo i, ela não poderá setar esta posição no vetor binário, pois outra *thread* poderá ler este valor e interpretá-lo como sendo uma combinação alcançada durante os passos anteriores. Assim, no passo i, um vetor binário armazenará as capacidades encontradas nos passos anteriores e o outro servirá de *buffer* para armazenar as capacidades encontradas adicionando o peso

w_i.

Ao final da computação do passo i, após sincronismo entre os blocos, as capacidades encontradas no *buffer* deverão ser setadas no vetor binário que contém as capacidades encontradas, antes de iniciar o próximo passo $i+1$. O vetor g pode ser configurado ao se descobrir a nova capacidade ou durante esta última etapa.

Como o vetor g é utilizado apenas para escrita e cada capacidade pode ser definida uma única vez, o recurso de memória mapeada (**Subseção 2.1.4**) pode ser utilizado. Pelo fato do vetor g não estar alocado na memória da GPU, este método se torna vantajoso em relação à implementação 1 (**Seção 5.2**), pois além de reduzir as transações de memória e computação, também reduz a quantidade de memória alocada na GPU para uma fração de $\frac{2}{\log_2(n)}$, em relação ao anterior.

5.4.2 Pseudocódigo da CPU

O código da CPU para invocar o *kernel* possui poucas diferenças em relação à implementação 1 (**Subseção 5.2.2**): apenas dois parâmetros a mais são passados ao *kernel 3*, referentes aos vetores binários, e após sua computação, $kernel3Flush$ é chamado para limpar o *buffer* e atualizar o vetor binário b. O pseudocódigo é apresentado no **Algoritmo 17**.

O mapeamento do vetor g, citado na **Subseção 2.1.4**, é realizado na linha 4, retornando o ponteiro para uso na GPU. A inicialização dos vetores binários (linha 5) e do vetor g, serve para limpá-los, ou seja, não se refere à inicialização tradicional. Apenas a primeira capacidade w_1 é atualizada nos vetores b e g, permitindo com que o laço externo (linha 11) inicie a computação na segunda iteração. Ao final da computação, não é

necessária a cópia do vetor g, uma vez que o vetor já está na memória do *host*.

5.4.3 Pseudocódigo do *kernel*

O pseudocódigo da GPU com os vetores binários implementados com elementos de 16-bit é apresentado em **Algoritmo 18**, **Algoritmo 19**, **Algoritmo 20** e **Algoritmo 21**.

Como cada *thread* irá computar uma coleção de 16 capacidades subsequentes, as va-

Algoritmo 17 Pseudocódigo da CPU para o *kernel* 3

1: // Ordenação dos pesos
2: // Otimização da capacidade
3: // Checagem de soluções triviais
4: // Mapeamento do vetor g da CPU para a GPU
5: // Inicialização dos vetores b e $buffer$ da GPU, e do vetor g da CPU

6: // Configuração de *threads* e blocos
7: $qtd_{SMs} \leftarrow 7$
8: $blocks_{\max} \leftarrow 2 * qtd_{SMs}$
9: $threads_{\max} \leftarrow 768$

10: // Laço externo percorre todos os itens
11: **for** $i = 2;\ i \leq n;\ i = i + 1$ **do**
12: // Otimização de limitação com ordenação dos pesos
13: $k_{i\min} \leftarrow \max\{w_i + w_{i-1}, c - \sum_{j=i+1}^{n} w_j\}$
14: $k_{i\max} \leftarrow \min(c,\ \sum_{j=1}^{i} w_j)$

15: // Quantidade de blocos necessários
16: $blocks \leftarrow \lceil \frac{k_{i\max} - k_{i\min} + 1}{threads_{\max}} \rceil$
17: // Limitar para somente blocos ativos
18: **if** $blocks > blocks_{\max}$ **then**
19: $blocks \leftarrow blocks_{\max}$
20: **end if**

21: // Chamada do *kernel*
22: $kernel3 \lll blocks, threads_{\max} \ggg (c, i, k_{i\min}, k_{i\max}, w_i, g, b, buffer)$

23: // Atualização do vetor binário e limpeza do *buffer*
24: $kernel3Flush \lll blocks, threads_{\max} \ggg (k_{i\min}, k_{i\max}, w_i, b, buffer)$
25: **end for**

26: // Recuperação da solução ótima X de g

riáveis *tid* e *inc* são multiplicadas por 16 (linhas 15 e 17), ou seja, é como se existissem 16 vezes a quantidade de *threads* na *grid*.

Algoritmo 18 Pseudocódigo do *kernel* 3 - Parte 1

1: $getCapacity(k, v)$: Retorna o valor binário referente à capacidade k do vetor v
2: $setCapacity(k, v)$: Seta a capacidade k do vetor binário v
3: $keepFirst(p)$: Retorna um valor com apenas os primeiros p *bits* setados
4: $getIdx(k)$: Retorna o índice de um vetor binário correlacionado à capacidade k

5: c: Capacidade da mochila
6: i: Índice do item analisado
7: $k_{i\,\min}$: Menor índice da otimização de limitação
8: $k_{i\,\max}$: Maior índice da otimização de limitação
9: w_i: Peso do item i
10: g: Ponteiro para o vetor g mapeado
11: b: Vetor binário com capacidades encontradas
12: $buffer$: Vetor binário temporário

13: **function** KERNEL3($c, i, k_{i\,\min}, k_{i\,\max}, w_i, g, b, buffer$)
14: // Identificador único da *thread* na grid multiplicado por 16
15: $tid \leftarrow (blockIdx.x * blockDim.x + threadIdx.x) \ll 4$
16: // Quantidades de *threads* na grid multiplicado por 16
17: $inc \leftarrow (gridDim.x * blockDim.x) \ll 4$

18: // Última *thread* seta a combinação w_i,
19: // caso ela ainda não tenha sido encontrada
20: **if** $tid = inc - 16 \wedge getCapacity(w_i, b) = 0$ **then**
21: // Seta o *buffer*
22: $setCapacity(w_i, buffer)$
23: // Configura o vetor mapeado
24: $g[w_i] \leftarrow i$
25: **end if**

Já que não é possível realizar a inicialização do vetor binário com as soluções de um único item, a capacidade w_i deve ser verificada por alguma *thread* durante o passo i. A condição da linha 20 realiza esta tarefa, setando o *buffer* e escrevendo no vetor mapeado g, caso a capacidade w_i não tenha sido preenchida nos passos anteriores. Isto é feito com a ajuda de duas funções auxiliares: *getCapacity* e *setCapacity*.

A função auxiliar $getCapacity(k, v)$ é responsável por retornar o valor binário correspondente à capacidade k do vetor binário v, ou seja, retorna 0 caso a capacidade k não

esteja preenchida e 1 caso contrário. A função $setCapacity(k, v)$ seta o *bit* referente à capacidade k no vetor binário v.

Algoritmo 19 Pseudocódigo do *kernel* 3 - Parte 2

```
26:   // Calcula a primeira capacidade a ser computada
27:   k ← k_{i min} + tid − ((k_{i min} + tid)&15)
28:   if w_i > k then
29:       // Recupera o elemento que contém a capacidade w_i
30:       cap ← b[getIdx(w_i)]
31:       // Recupera as combinações complementares e as desloca
32:       //  para seus bits correspondentes às capacidades
33:       comp ← (cap&keepFirst(16 − w_i&15)) ≪ (w_i&15)

34:       // 16 condições de escrita são verificadas de uma única vez
35:       comp ← comp&(∼ cap)
36:       // Caso seja encontrada novas combinações
37:       if comp > 0 then
38:           // Seta as combinações no buffer
39:           buffer[getIdx(w_i)] ← cap|comp

40:           // Atualiza as capacidades encontradas no vetor mapeado g
41:           for bit = 0; comp > 0; bit ← bit + 1, comp ← comp ≫ 1 do
42:               if comp&1 > 0 then
43:                   g[w_i + bit] ← i
44:               end if
45:           end for
46:       end if

47:       // Adiciona o incremento para continuar no laço
48:       k ← k_{i min} + tid − ((k_{i min} + tid)&15) + inc
49:   end if
```

O **Algoritmo 19** apresenta a segunda parte do pseudocódigo, onde, na linha 27, uma correção é aplicada no cálculo da capacidade inicial, fazendo com que as *threads* computem 16 capacidades contidas apenas em um único elemento do vetor b. Isto facilita a computação, uma vez que será necessário requisitar apenas um elemento do vetor b.

Entretanto, devido a este ajuste, pode existir o caso do peso w_i ser maior que a capacidade computada k, resultando em uma combinação complementar $k - w_i$ negativa. Como isto não pode ocorrer, este caso é tratado isoladamente no início da computação

(linhas 28-49).

Esta computação é similar à realizada no laço que percorre as capacidades (linhas 51-81), explicada a seguir, porém, neste caso, as combinações complementares e as capacidades estarão contidas no mesmo elemento do vetor binário, ou seja, somente uma requisição à memória será necessária; e apenas as capacidades $k \leq w_i$ serão tratadas. Ao final desta computação, na linha 48, a capacidade inicial recebe o incremento do laço *inc* (linha 51), como se pertencesse a ele.

No início do laço da terceira parte do pseudocódigo (**Algoritmo 20**), as capacidades analisadas e suas respectivas combinações complementares são recuperadas com o auxílio de duas funções auxiliares: $getIdx$ e $keepFirst$ (linhas 53-67).

A função $getIdx(k)$ é utilizada para retornar o índice do elemento de 16-bit, de um vetor binário, que contém o *bit* referente à capacidade k. Já a função $keepFirst(p)$, retorna um valor com apenas os primeiros p *bits* setados, e é utilizada em conjunto com a operação binária "&" (E lógico) para manter apenas os respectivos p *bits* de interesse.

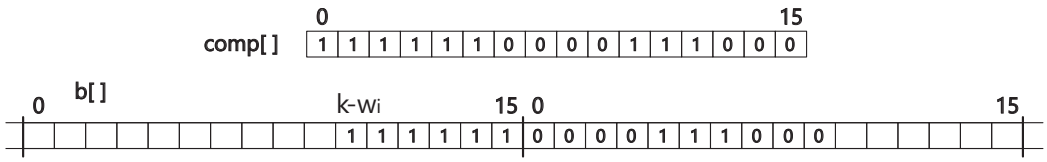

FIGURA 5.4 – Leitura das capacidades complementares binárias.

As capacidades analisadas são recuperadas na linha 53 e seus respectivos complementos são recuperados nas linhas 56-67. Este conjunto de instruções para a recuperação das combinações complementares é necessário, pois os *bits* correspondentes a estas combinações podem estar em duas posições consecutivas do vetor b, como mostra a **Figura 5.4**. O que estas instruções fazem é deslocar as primeiras combinações para os primeiros *bits* da variável *comp* e adicionar as restantes nos *bits* mais altos.

Algoritmo 20 Pseudocódigo do *kernel* 3 - Parte 3

```
50:     // Percorre todas as capacidades
51:     for ; k ≤ k_{i max}; k ← k + inc do
52:         // Recupera o elemento que contém a capacidade k
53:         cap ← b[getIdx(k)]
54:         // Recupera a primeira parte das combinações complementares
55:         // nos bits baixos da variável comp
56:         comp ← b[getIdx(k − w_i)] ≫ ((k − w_i)&15)
57:         // Limpa os bits restantes
58:         comp ← comp&keepFirst(16 − ((k − w_i)&15))

59:         // Recupera a segunda parte
60:         if (k − w_i)&15 > 0 then
61:             // Requisita o próximo elemento do vetor b
62:             high ← b[getIdx(k − w_i) + 1]&keepFirst((k − w_i)&15)
63:             // Desloca para os bits altos
64:             high ← high ≪ (16 − ((k − w_i)&15))
65:             // Combina na variável comp
66:             comp ← high|comp
67:         end if

68:         // Condição de escrita é verificada de uma única vez
69:         comp ← comp&(∼ cap)
70:         // Caso seja encontrada novas combinações
71:         if comp > 0 then
72:             // Seta as combinações no buffer
73:             buffer[getIdx(k)] ← cap|comp

74:             // Atualiza as capacidades encontradas no vetor mapeado g
75:             for bit = 0; comp > 0; bit ← bit + 1, comp ← comp ≫ 1 do
76:                 if comp&1 > 0 then
77:                     g[k + bit] ← i
78:                 end if
79:             end for
80:         end if
81:     end for
82: end function
```

Na linha 69, a condição de escrita é realizada para as 16 capacidades de uma única vez, com o operador de negação binária "∼" e o operador binário "&" (E lógico). Em seguida, na linha 71, é verificado se alguma nova combinação foi encontrada. Caso exista, a nova configuração é setada no *buffer* (linha 73).

Em seguida, o vetor mapeado g é preenchido com as capacidades encontradas, nas

linhas 75-79. O laço destas linhas, desloca cada *bit* analisado da variável *comp* para a posição mais baixa, e, caso esteja setado, atualiza o vetor g.

Algoritmo 21 Pseudocódigo do *kernel* 3 Flush

```
1:  function KERNEL3FLUSH(k_{i min}, k_{i max}, w_i, b, buffer)
2:      tid ← (blockIdx.x * blockDim.x + threadIdx.x) ≪ 4
3:      inc ← (gridDim.x * blockDim.x) ≪ 4

4:      // Última thread analisa a combinação w_i
5:      if tid = inc − 16 ∧ buffer[getIdx(w_i)] > 0 then
6:          // Atualiza o vetor b com o novo valor
7:          b[getIdx(w_i)] ← buffer[getIdx(w_i)]
8:          // Limpa o buffer
9:          buffer[getIdx(w_i)] ← 0
10:     end if

11:     // Percorre todas as capacidades
12:     for k ← k_{i min} + tid − ((k_{i min} + tid)&15); k <= k_{i max}; k+ ← inc do
13:         // Atualiza, caso exista uma nova combinação
14:         if buffer[getIdx(k)] > 0 then
15:             b[getIdx(k)] ← buffer[getIdx(k)]
16:             buffer[getIdx(k)] ← 0
17:         end if
18:     end for
19: end function
```

No **Algoritmo 21** é apresentado o código referente ao *kernel* que atualiza os vetores, após o sincronismo. Assim como anteriormente, é atualizada a capacidade w_i separadamente (linhas 5-10) e, em seguida, são atualizadas as outras capacidades encontradas (linhas 12-18), através do mesmo laço que o pseudocódigo anterior.

Basicamente, o *kernel* encontra os elementos do *buffer* diferentes de 0 e copia a nova configuração de capacidades para o vetor b (linhas 7 e 15). Em seguida, o *buffer* é limpo para a próxima computação (linhas 9 e 16). Vale ressaltar que, assim como no *kernel* anterior, 16 capacidades são atualizadas de uma única vez, pelo fato das atualizações serem feitas nos elementos dos vetores binários.

5.4.4 Alternativas de implementação

Como dito na **Seção 2.1**, a arquitetura Fermi possui registradores de 32 bits. Caso os elementos dos vetores binários fossem de 32-bit ao invés de 16-bit, o dobro de operações sobre seus elementos poderiam ser calculadas de uma única vez.

Este pseudocódigo não é apresentado neste trabalho, por ser quase idêntico ao algoritmo com elementos de 16-bit; basta trocar o tipo dos vetores binários e as constantes correlacionadas aos elementos. Os resultados desta abordagem também são apresentados no **Capítulo 6**.

6 Resultados computacionais

Neste capítulo é realizado *benchmarking* com os algoritmos citados no estado da arte, junto com algumas análises sobre o potencial dos algoritmos paralelos propostos.

6.1 Benchmarking

O computador utilizado nos testes foi um AMD FX-6100 3.3 GHz com 8 GB RAM, 1 GPU NVIDIA GeForce GTX 560 não dedicada com 7 SMs de 48 *cores* de 1.8 GHz, 1 GB de memória global e Ubuntu 64 bits 12.04.2 LTS. O NVCC 5.0.35 Release 1 foi utilizado em conjunto com o GCC versão 4.6.3 para compilação dos códigos.

As mesmas entradas foram utilizadas para todos os algoritmos com a finalidade de reduzir as variações temporais, e os tempos transcorridos apresentados neste capítulo são sempre a média de 10 amostragens. Os pesos também foram restringidos para inteiros de apenas 32 *bits*, pelo fato dos outros algoritmos ocuparem muito espaço de memória.

Foram testados os seguintes códigos:

BALSUB: Algoritmo sequencial Balsub

DECOMP: Algoritmo sequencial Decomp

YS: Algoritmo sequencial de Yanasse & Soma

BBE: Algoritmo paralelo de Boyer, Baz & Elkihel, cedido gentilmente

K1: Algoritmo paralelo da implementação 1

K2: Algoritmo paralelo da implementação 2

K3-16: Algoritmo paralelo da implementação 3, com variáveis de 16-bit para o vetor binário

K3-32: Algoritmo paralelo da implementação 3, com variáveis de 32-bit para o vetor binário

O algoritmo BBE foi gentilmente cedido pelos atores e sua implementação corresponde à versão original.

Como o objetivo deste trabalho é o de elaborar algoritmos capazes de retornar soluções ótimas de casos difíceis, onde não é possível utilizar otimizações que limitem a solução ótima à capacidade da mochila, o *benchmarking* foca em instâncias que não possuem soluções exatas. Baseado em (MARTELLO; TOTH, 1990) e (BOYER; BAZ; ELKIHEL, 2012), três tipos de *benchmarks* são utilizados neste trabalho:

Half: $w_i = 10k$, k randomicamente distribuído em $[1, 10^5]$, e $c = \left\lfloor \frac{\sum_{i=1}^{n} w_i}{20} \right\rfloor 10 + 5$.

P: w_i par, randomicamente distribuído em $[2, 10^6]$, e $c = \left\lfloor \frac{n10^6}{8} \right\rfloor + (1)$ (ímpar), onde n é a quantidade de pesos.

Uniform: w_i par, randomicamente distribuído em $[2, c-1]$, e $c = 10^6 + 1$.

O *benchmark Half* tem o objetivo de analisar o pior caso dos algoritmos, quando a otimização de capacidade não reduz a capacidade que será analisada e o vetor g possui o maior tamanho para os itens analisados.

O *benchmark P*, baseado nas instâncias *Even-odd* e *P6* de (MARTELLO; TOTH, 1990), aborda casos em que a solução ótima é menor que a metade da soma dos itens. Embora a otimização de capacidade não reduza o tamanho do vetor g neste caso, ele será pequeno em relação a soma dos pesos. A capacidade c deverá ser sempre ímpar, caso contrário, o valor 1 (em parênteses) é adicionado para garantir esta exigência.

O *benchmark Uniform* apenas analisa o caso particular em que os pesos são distribuídos uniformemente até a capacidade da mochila, similar a (BOYER; BAZ; ELKIHEL, 2012).

6.1.1 Resultados

As seguintes tabelas apresentam os tempos em segundos gastos pelos códigos, em relação ao número de itens.

TABELA 6.1 – Código Balsub (em segundos).

n	Half	P	Uniform
10	0,06	0,06	0,08
30	0,19	0,17	0,19
100	0,66	0,64	0,68
300	1,93	1,75	2,01
1000	6,45	5,48	8,99
3000	$	$	$
10000	$	$	$

Na **Tabela 6.1** é possível ver os resultados do código Balsub. No *benchmark P*, por exemplo, foram gastos 5,48 segundos para computar a instância com 1.000 itens, e para as instâncias com 3.000 ou mais itens, a memória principal de 8 GB do *host*, além da memória virtual, não foi suficiente (representado pelo caractere $).

Os resultados do código Decomp são apresentados na **Tabela 6.2** e embora seja possível computar instâncias maiores, os tempos são substancialmente piores que os do Balsub, com exceção para o *benchmark Uniform*. Nas instâncias *P* com 3.000 itens e *Half* com

TABELA 6.2 – Código Decomp (em segundos).

n	Half	P	Uniform
10	0,00	0,00	0,00
30	0,00	0,01	0,00
100	0,14	4,82	0,08
300	2,34	59,42	0,86
1000	37,22	714,80	3,75
3000	338,96	>1800	11,66
10000	>1800	$	38,30

10.000 itens, os processos foram abortados após 2 horas de computação, e para a instância *P* com 10.000 itens, também houve a necessidade de mais memória no *host*.

TABELA 6.3 – Código YS (em segundos).

n	Half	P	Uniform
10	0,00	0,01	0,00
30	0,00	0,05	0,00
100	0,07	1,64	0,02
300	0,88	15,42	0,29
1000	10,32	170,94	1,32
3000	92,04	>1800	4,72
10000	>1800	>1800	16,08

Os resultados do código YS, na **Tabela 6.3**, apresentam tempos melhores do que o Decomp em todas as instâncias, e melhores que o Balsub na instância *Uniform*, sendo possível computar todas as instâncias.

TABELA 6.4 – Código BBE (em segundos).

n	Half	P	Uniform
10	0,07	0,07	0,06
30	0,08	0,14	0,08
100	0,17	0,71	0,09
300	0,95	3,47	0,15
1000	9,61	#	0,38
3000	#	#	1,02
10000	#	#	3,10

A **Tabela 6.4** apresenta os resultados do código BBE, onde é possível observar uma melhora nos *benchmarks* *P* e *Uniform*, em relação aos códigos Decomp e YS. Já para o

código Balsub, apenas apresenta melhora para o *benchmark Uniform*, com redução do tempo de execução de até 23 vezes para a instância de 1.000 itens. Entretanto, como já dito anteriormente (**Seção 3.2**), esta comparação não deve ser feita sem esquecer que o algoritmo apenas computa soluções ótimas iguais à capacidade da mochila, ou seja, caso o algoritmo seja alterado para sempre encontrar a solução ótima, seu desempenho pode ser pior.

As instâncias acima de 1.000 itens para o *benchmark Half* e acima de 300 itens para o *benchmark P* excederam a memória global de 1 GB da GPU (representado pelo caractere #).

TABELA 6.5 – Código K1 (em segundos).

n	Half	P	Uniform
10	0,00	0,06	0,06
30	0,00	0,07	0,06
100	0,07	0,15	0,06
300	0,18	0,82	0,07
1000	1,61	9,39	0,08
3000	16,28	84,62	0,10
10000	194,08	0,00	0,21

A **Tabela 6.5** apresenta os resultados do código K1, superando o BBE em todas as instâncias e reduzindo o tempo computacional em até 76 vezes em relação ao YS e 14 vezes em relação ao BBE, para a instância *Uniform* com 10.000 itens. O baixo tempo na instância *P* com 10.000 itens se refere à verificação dos casos triviais (**Seção 4.4**), que encontra rapidamente a solução.

Os resultados da **Tabela 6.6** demonstram que o código *K2* é pior do que o *K1* em todas as instâncias. Nas instâncias de 3.000 e 10.000 itens do *benchmark Half* ainda foi excedida a memória global de 1 GB da GPU. Isto ocorre devido ao fato deste algoritmo também alocar o vetor de pesos na memória, necessitando de 4 *bytes* por peso.

TABELA 6.6 – Código K2 (em segundos).

n	Half	P	Uniform
10	0,00	0,06	0,06
30	0,00	0,08	0,06
100	0,08	0,21	0,06
300	0,30	1,28	0,09
1000	2,76	12,27	0,15
3000	#	108,15	0,32
10000	#	0,00	1,00

TABELA 6.7 – Código K3-16 (em segundos).

n	Half	P	Uniform
10	0,00	0,05	0,05
30	0,00	0,06	0,05
100	0,05	0,14	0,05
300	0,09	0,49	0,05
1000	0,32	2,45	0,06
3000	2,13	13,76	0,07
10000	22,16	0,00	0,12

A **Tabela 6.7** apresenta os tempos para o *kernel* 3, onde o vetor g é implementado com elementos de 16-bit. É possível verificar que este *kernel* possui o melhor tempo em relação aos anteriores, reduzindo o tempo de execução em até 30 vezes em relação ao BBE, 154 em relação ao Balsub e 135 em relação ao YS.

TABELA 6.8 – Código K3-32 (em segundos).

n	Half	P	Uniform
10	0,00	0,05	0,05
30	0,00	0,06	0,05
100	0,05	0,11	0,05
300	0,08	0,32	0,05
1000	0,24	1,33	0,06
3000	1,30	7,02	0,07
10000	12,57	0,00	0,11

A **Tabela 6.8** apresenta os tempos para o *kernel* 3, onde o vetor g é implementado com elementos de 32-bit. Nos melhores casos, ele reduz o tempo de execução pela metade em relação ao K3-16, pelo fato dos registradores da GPU serem de 32-bit, o que permite

a computação dos 32 valores de um elemento do vetor com o mesmo custo temporal de 16 valores. Assim, este *kernel* atinge tempos até 162 vezes melhores em relação ao Balsub, 336 em relação ao Decomp, 141 em relação ao YS e 40 em relação ao BBE.

6.2 Análise de memória

Os códigos propostos possuem complexidade espacial $O(n + m - w_{\min})$, onde $m = \min\{c, c - \sum_{i=1}^{n} w_i\}$, pelo fato de calcularem o vetor g com a otimização de capacidade. Assim, mesmo se considerada a melhor taxa de compressão (*cf.* (BOYER; BAZ; ELKIHEL, 2012)), a memória utilizada nos trabalhos de paralelização (BOKHARI, 2012) e (BOYER; BAZ; ELKIHEL, 2012) ainda se mantém muito alta ($O(nc)$).

TABELA 6.9 – Memória requirida pela tabela de decisão compactada (em GB).

	c			
n	10^6	10^7	10^8	10^9
10^4	$< 0,1$	$< 0,1$	$< 0,1$	0,4
10^5	$< 0,1$	$< 0,1$	0,4	3,6
10^6	$< 0,1$	0,4	3,6	36,1
10^7	0,4	3,6	36,1	360,9
10^8	3,6	36,1	360,9	3.608,9
10^9	36,1	360,9	3.608,9	36.088,7

A **Tabela 6.9** apresenta a memória requirida pelo código de Boyer, Baz & Elkihel para representação apenas da tabela de decisão, considerando a melhor taxa de compressão (0,00031), sem contar o vetor auxiliar necessário ao algoritmo de Bellman com matriz binária e o vetor compactado utilizado por Boyer, Baz & Elkihel. Como o vetor auxiliar possui tamanho igual a capacidade da mochila e armazena os índices dos pesos, isto equivale exatamente na mesma memória gasta para representação do vetor g.

Desconsiderando a otimização de capacidade, a representação do vetor g ocupa as

TABELA 6.10 – Memória requirida pelo vetor G (em GB).

n	c 10^6	10^7	10^8	10^9
10^4	$< 0,1$	$< 0,1$	0,4	3,7
10^5	$< 0,1$	$< 0,1$	0,4	3,7
10^6	$< 0,1$	$< 0,1$	0,4	3,7
10^7	$< 0,1$	$< 0,1$	0,4	3,7
10^8	$< 0,1$	$< 0,1$	0,4	3,7
10^9	$< 0,1$	$< 0,1$	0,4	3,7

quantidades apresentadas pela **Tabela 6.10**. No maior problema, é possível observar uma redução de 9.687 vezes.

TABELA 6.11 – Memória requirida pela GPU para o *kernel* 3 (em GB).

n	c 10^6	10^7	10^8	10^9
10^4	$< 0,1$	$< 0,1$	$< 0,1$	0,2
10^5	$< 0,1$	$< 0,1$	$< 0,1$	0,2
10^6	$< 0,1$	$< 0,1$	$< 0,1$	0,2
10^7	$< 0,1$	$< 0,1$	$< 0,1$	0,2
10^8	$< 0,1$	$< 0,1$	$< 0,1$	0,2
10^9	$< 0,1$	$< 0,1$	$< 0,1$	0,2

Em relação ao consumo de memória da GPU, os algoritmos baseados na matriz binária podem apresentar menor consumo de memória já que alocam dois vetores binários de tamanho igual à capacidade da mochila, enquanto que o vetor g necessita de um vetor com elementos de 16-bit e mesmo tamanho. Entretanto, o *kernel* 3 também utiliza apenas dois vetores binários, implicando no mesmo consumo de memória, ou inferior nos casos em que for computada a capacidade complementar. Desta forma, embora o *kernel* 3 necessite das quantidades descritas na **Tabela 6.10** no lado do *host*, a memória necessária na GPU é muito inferior, como apresentado na **Tabela 6.11**.

6.3 Análise de eficiência

Nesta seção, é analisada a eficiência do código K3-32 em número de *threads* ativas, *i.e.* a relação do *speed-up* obtido pelo número de *threads* ativos, por ser o código com melhores resultados obtidos. Para isto, foi limitada a quantidade de *threads* disponíveis: se uma GPU possui somente t *threads* para computar, então no máximo t serão responsáveis por computá-las.

Somente é considerada a solução da instância P com 1.000 itens, gastando 1,33 segundos quando utilizada a capacidade máxima da GPU, e a quantidade de *threads* varia de 1 a 10.752, uma vez que a GPU somente aceita esta quantidade de *threads* ativas.

A **Figura 6.1** apresenta o tempo relativo da computação do código K3-32 para as partes CPU e GPU. A parte da CPU inclui as inicializações, otimizações e a recuperação da solução ótima.

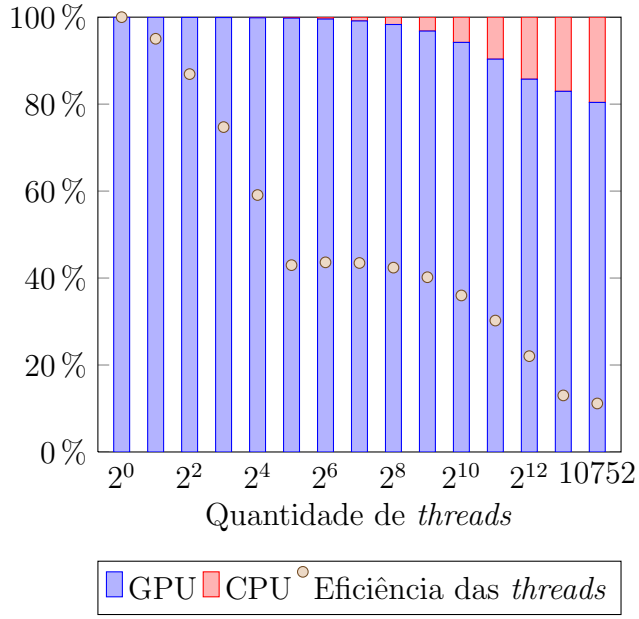

FIGURA 6.1 – Eficiência em relação a quantidade de *threads* do *kernel* K3-32.

Nos resultados, é possível notar que, com o aumento exponencial do paralelismo, o tempo computacional da CPU cresce em relação ao da GPU, atingindo um máximo de

19,5% do tempo total.

Analisando o código foi possível identificar que aproximadamente 95% do tempo da CPU é dedicado às otimizações e inicialização dos vetores, ou seja, antes de iniciar a computação na GPU. Os 5% restantes são responsáveis por encontrar a solução ótima e o vetor solução X. Outro fato é que este tempo se manteve aproximadamente constante em 0,25 segundos para todas as quantidades de *threads* ativas.

A eficiência final do código K3-32 também é apresentada na **Figura 6.1** pelos pontos. É obtida uma eficiência de 11% quando o poder máximo da GPU é utilizado. Na faixa de 2^5 a 2^{10} *threads* ativas, há uma eficiência próxima de 40%.

A redução da eficiência é causada devido ao tempo gasto na CPU, baixa quantidade de *cores* (336) para computar 10.752 *threads*, e pelo fato da otimização de limitação reduzir o nível de paralelismo a cada etapa do algoritmo ao aumentar a capacidade inicial.

6.4 Análise do sincronismo cíclico

Com o intuito de verificar o desempenho do método de sincronismo proposto, o mesmo *benchmark* utilizado em (XIAO; FENG, 2010) foi testado: uma multiplicação simples de ponto flutuante é realizada 1.000.000 vezes.

Algoritmo 22 Kernel para *benchmarking* dos sincronismos.

```
1: for i ← 1; i ≤ 1.000.000; i ← i + 1 do
2:     n ← n * threadIdx.x
3:     Sincronismo()
4: end for
```

No **Algoritmo 22**, é apresentado o *kernel* referente ao cálculo com sincronismo, chamando a respectiva função em cada iteração. A computação do *kernel* sem o sincronismo também é realizada, com o objetivo de medir o tempo gasto apenas com o laço e o cálculo

de ponto flutuante.

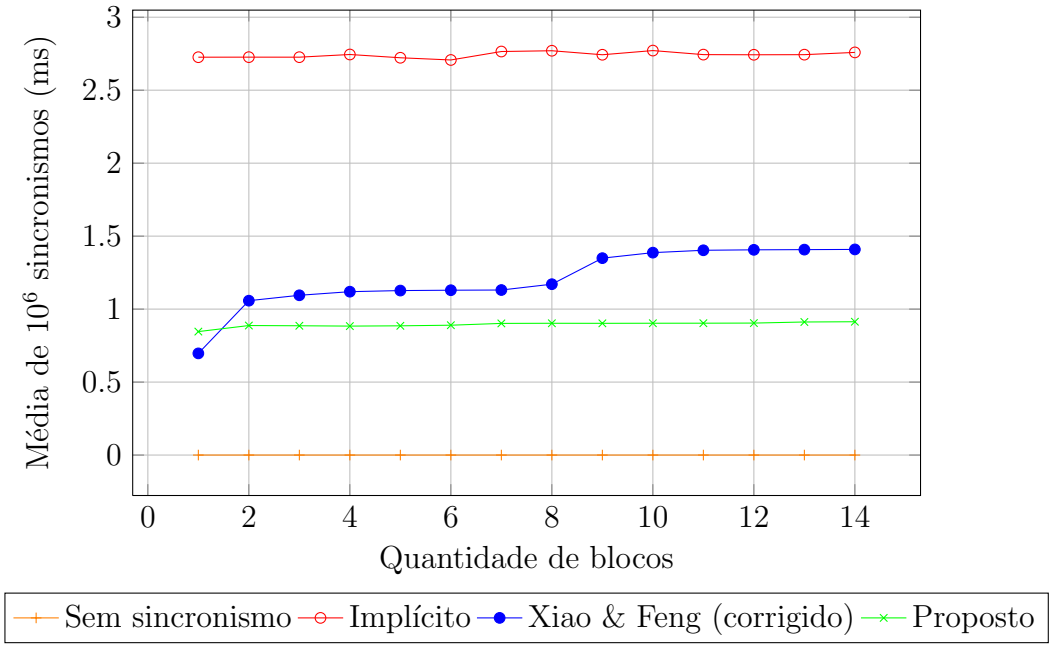

FIGURA 6.2 – Tempo das funções de sincronismo entre blocos.

A comparação com o sincronismo implícito da plataforma CUDA é realizada invocando 1.000.000 vezes um *kernel* que apenas contém o cálculo de ponto flutuante.

A **Figura 6.2** apresenta os resultados obtidos variando a quantidade de blocos até o máximo de 14 blocos ativos, por ser o limite máximo da GPU utilizada neste trabalho (**Seção 6.1**).

No teste realizado, a quantidade de computação aumenta proporcionalmente com a quantidade de blocos ativos utilizados, de tal forma que resultem em um tempo constante uma vez que todos os blocos são computados paralelamente, ou seja, a chamada ao *kernel* é realizada como: $kernel \lll blocos,\ threads \ggg()$, onde *blocos* representa a quantidade de blocos ativos e *threads* é a quantidade máxima de *threads* ativas por bloco, discutida na **Subseção 5.1.3**.

É possível verificar que o tempo do sincronismo proposto se mantêm praticamente constante, apresentando uma melhora de 54% em relação ao algoritmo corrigido de Xiao

& Feng, para 14 blocos, e uma melhora no tempo de execução de até 2,9 vezes em relação ao sincronismo implícito da plataforma CUDA. Próximo a 8 blocos, é possível identificar um deslocamento no tempo do algoritmo de Xiao & Feng, provavelmente causado pelo fato da GPU ter 7 SMs.

No *kernel* 2, foi possível identificar uma melhora de aproximadamente 15% no tempo final de computação, em relação ao método de Xiao & Feng.

7 Conclusão

Este trabalho apresenta basicamente três algoritmos (K1, K2, K3) testados em uma maior variedade de *benchmarks* em relação aos trabalhos (BOYER; BAZ; ELKIHEL, 2012) e (BOKHARI, 2012), resolvendo grandes instâncias do SSP em GPU com tempos de execução até 40 vezes melhores (código K3 de 32-bit) em relação ao melhor resultado prático paralelo já conhecido (BOYER; BAZ; ELKIHEL, 2012), na mesma arquitetura. Isto foi alcançado graças ao alto grau de paralelismo e baixa necessidade de memória requerida, concluindo ser a arquitetura GPU uma ótima opção para solução do SSP.

O código K3 demonstrou ser a melhor das implementações, tomando os registradores como mais um nível de paralelismo, uma vez que cada *bit* pode ser utilizado para computar uma capacidade. Este código, implementado com elementos de 32-bit, atingiu uma eficiência mínima de 11%, em relação à quantidade máxima de *threads* ativas, e próxima ou acima de 40% para 1.024 ou menos *threads* ativas. Foi possível também atingir, nos melhores casos, tempos de execução 141 vezes melhores que aos algoritmos sequenciais Balsub, Decomp e YS, além de reduções no tempo de execução acima de 10 vezes em uma grande quantidade de instâncias sobre o algoritmo de Boyer, Baz & Elkihel.

O código K2 demonstrou baixo desempenho devido à necessidade de sincronismos entre *threads* de blocos distintos, porém, com a implementação de sincronismos entre

blocos em *hardware* ou outro mecanismos que permita coordenar as dependências, pode ser que esta estratégia ainda apresente bom desempenho. O código K1, por ser uma versão simplificada do K3, apresentou maior tempo computacional que sua versão otimizada.

Com a recursão proposta para o cálculo do vetor g de Yanasse & Soma, foi possível generalizar algumas formas de computação, incluindo a forma tradicional apresentada em seu algoritmo, através de uma substituição de variáveis (**Subseção 5.3.4**), além de tornar possível a criação de um algoritmo paralelo com tempo $O(n)$ e espaço $O(n+c)$, caso existam c processadores (**Subseção 5.1.1**), baixando o *upperbound* teórico do problema.

Um erro foi identificado no método de sincronismo entre blocos de Xiao & Feng, e foram apresentados sua correção e um novo método de sincronismo, com a finalidade de evitar a espera de todos os blocos.

O novo método de sincronismo obteve melhora de 54% em relação ao tempo de Xiao & Feng (**Seção 6.4**), com tempo de execução 2,9 vezes melhor que o sincronismo implícito da plataforma CUDA, o que torna seu uso extramente interessante em qualquer aplicação que faça uso intenso deste sincronismo, além de permitir o uso dos dados lidos nos registradores e na *shared memory* entre os sincronismos. No *kernel* 2, foi possível identificar uma melhora de aproximadamente 15% no tempo final de computação, em relação ao método de Xiao & Feng.

Novas otimizações foram apresentadas, reduzindo tanto o tempo quanto o espaço gasto, e algumas otimizações foram adaptadas tornando-as possível de computar soluções ótimas diferentes da capacidade da mochila, o que permite resolver o SSP em espaço $O(n + m - w_{\min})$ e tempo $O(n \log n + (n-1)(m - 2w_{\min}) + w_{\min})$, onde $m = \min\{c, c - \sum_{i=1}^{n} w_i\}$.

7.1 Trabalhos futuros

Os seguintes trabalhos futuros são recomendados como potenciais melhorias:

- Aumento do nível de paralelismo, ao computar as linhas e colunas da tabela de Bellman concorrentemente.
- Implementação em GPU, com vetor g maior do que a memória global disponível, através de cópias entre CPU e GPU.
- Uso do paralelismo dinâmico da arquitetura CUDA 3.x para redução dos tempos de sincronismo.
- Implementação dos algoritmos com vetores esparsos.
- Implementação do KP01 com as novas otimizações apresentadas.
- Teste do novo método de sincronismo em algoritmos clássicos que tenham grande uso prático.

Referências

BELLMAN, R. E. **Dynamic programming**. Princeton, NJ: Princeton University Press, 1957.

BOHMAN, T. A sum packing problem of Erdös and the Conway-Guy sequence. **Proceedings of the American Mathematical Society**, v. 124, n. 12, p. 3627–3636, Dec. 1996.

BOKHARI, S. S. Parallel solution of the subset-sum problem: an empirical study. **Concurrency and Computation**, v. 24, n. 18, p. 2241–2254, Dec. 2012.

BOYER, V.; BAZ, D. E.; ELKIHEL, M. Solving knapsack problems on GPU. **Computers and Operations Research Journal**, v. 39, n. 1, p. 42–47, Jan. 2012.

DONGARRA, J. **On the future of high performance computing**. Utah, Salt Lake City: University of Utah, Feb. 2012. Disponível em: <http://www.netlib.org/utk/people/JackDongarra/SLIDES/uu-sci-0212.pdf>. Acesso em: 1 mar. 2013.

GAREY, M. R.; JOHNSON, D. S. **Computers and intractability**: a guide to the theory of np-completeness. 1st. ed. New York: W. H. Freeman, 1979. 340 p.

GEMMELL, P.; JOHNSTON, A. M. **Analysis of a subset sum randomizer**. [S.l.]: International Association for Cryptologic Research, Feb. 2001. Disponível em: <http://eprint.iacr.org/2001/018>. Acesso em: 21 fev. 2013. (IACR Cryptology ePrint Archive, Report 2001/018).

HILL, M. D.; MARTY, M. R. Amdahl's law in the multicore era. **IEEE Computer Journal**, v. 41, n. 7, p. 33–38, Jul. 2008.

HONG, S. *et al.* Accelerating CUDA graph algorithms at maximum warp. In: SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 16., 2011, San Antonio. **Proceedings...** New York: ACM, 2011. p. 267–276.

HOROWITZ, E.; SAHNI, S. Computing partitions with applications to the knapsack problem. **Journal of ACM**, v. 21, n. 2, p. 277–292, Apr. 1974.

IRVINE, S. A.; CLEARY, J. G.; RINSMA-MELCHERT, I. **The subset sum problem and arithmetic coding**. Hamilton: University of Waikato, 1995. 24 p. (Computer Science Working Papers, v. 7). Disponível em: <http://researchcommons.waikato.ac.nz-/bitstream/handle/10289/1085/uow-cs-wp-1995-07.pdf>. Acesso em: 21 fev. 2013.

KATE, A.; GOLDBERG, I. Generalizing cryptosystems based on the subset-sum problem. **International Journal of Information Security**, v. 10, n. 3, p. 189–199, Jun. 2011.

KAWANAMI, K.; FUJIMOTO, N. GPU accelerated computation of the longest common subsequence. In: FACING THE MULTICORE, 2., 2011, Karlsruhe. **Proceedings...** Heidelberg: Springer Verlag, 2012. p. 84–95. (Lecture Notes in Computer Science, v. 7174).

KELLERER, H.; PFERSCHY, U.; PISINGER, D. **Knapsack problems**. Heidelberg: Springer Verlag, 2004. 566 p.

LIN, C.-H. *et al.* Accelerating string matching using multi-threaded algorithm on GPU. In: IEEE GLOBAL TELECOMMUNICATIONS CONFERENCE, 2010, Miami. **Proceedings...** Piscataway: IEEE, 2010. p. 1–5.

MARTELLO, S.; TOTH, P. **Knapsack Problems**: algorithms and computer implementations. New York: John Wiley & Sons, 1990.

MERRILL, D.; GARLAND, M.; GRIMSHAW, A. Scalable GPU graph traversal. In: SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 17., 2012, New Orleans. **Proceedings...** New York: ACM, 2012. v. 47, n. 8, p. 117–128.

MEUER, H. *et al.* (*Ed.*). **Top500 List**: Nov. 2012. [S.l.], 2012. Disponível em: <http://www.top500.org/list/2012/11/>. Acesso em: 1 mar. 2013.

NVIDIA. **CUDA compute architecture**: Fermi. Versão 1.1. [S.l.], 2009. 22 p. Disponível em: <http://www.nvidia.com/content/PDF/fermi_white_papers-/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Acesso em: 21 fev. 2013.

_____. **CUDA GPU occupancy calculator**: Versão 5.1. [S.l.], 2012a. Disponível em: <http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator-.xls>. Acesso em: 21 fev. 2013.

_____. **NVIDIA CUDA C programming guide**: Versão 5.0. [S.l.], 2012b. 175 p. Disponível em: <http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf>. Acesso em: 21 fev. 2013.

_____. **NVIDIA CUDA C best pratices guide**: Versão 5.0. [S.l.], 2012c. 63 p. Disponível em: <http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf>. Acesso em: 21 fev. 2013.

O'NEIL, T.; KERLIN, S. A simple $o(2^{\sqrt{x}})$ algorithm for partition and subset sum. In: INTERNATIONAL CONFERENCE ON FOUNDATIONS OF COMPUTER SCIENCE, 2010, Las Vegas. **Proceedings...** Las Vegas: CSREA Press, 2010. p. 55–58.

PISINGER, D. **An O(nr) algorithm for the subset-sum problem and other balanced knapsack algorithms**. Copenhagen: University of Copenhagen, 1995. Disponível em: <http://www.diku.dk/~pisinger/95-6.ps>. Acesso em: 21 fev. 2013.

PÓLYA, G. On picture-writing. **The American Mathematical Monthly**, v. 63, n. 10, p. 689–697, Dec. 1956.

SANCHES, C.; SOMA, N.; YANASSE, H. An optimal and scalable parallelization of the two-list algorithm for the subset-sum problem. **European Journal of Operational Research**, v. 176, n. 2, p. 870–879, Jan. 2007.

_____. Parallel time and space upper-bounds for the subset-sum problem. **Theoretical Computer Science**, v. 407, n. 1-3, p. 342–348, Nov. 2008.

SOMA, N. Y.; TOTH, P. An exact algorithm for the subset sum problem. **European Journal of Operational Research**, v. 136, n. 1, p. 57–66, Jan. 2002.

SOMA, N. Y.; YANASSE, H. An exact pseudopolynomial algorithm for the value independent knapsack problem. In: SIMPÓSIO BRASILEIRO DE PESQUISA OPERACIONAL, 20., 1987, Salvador. **Proceedings...** Rio de Janeiro: SOBRAPO, 1987. v. 1, p. 710–719.

SØRENSEN, H. H. B. High-performance matrix-vector multiplication on the GPU. In: EUROPEAN CONFERENCE ON PARALLEL PROCESSING, 17., 2011, Bordeaux. **Proceedings...** Heidelberg: Springer Verlag, 2012. p. 377–386. (Lecture Notes in Computer Science, v. 7155).

STEARNS, R. E.; HUNT, H. B. Power indices and easier hard problems. **Mathematical Systems Theory**, v. 23, n. 4, p. 209–225, Dec. 1990.

SUN, Z.-W. Unification of zero-sum problems, subset sums and covers of z. **Electronic Research Announcements of The American Mathematical Society**, v. 9, p. 51–60, Jul. 2003.

SUTTER, H. **A fundamental turn toward concurrency in software**. San Francisco: Dr. Dobb's, Mar. 2005. Disponível em: <http://www.drdobbs.com/web-development/a-fundamental-turn-toward-concurrency-in/184405990>. Acesso em: 1 mar. 2013.

TOTH, P. Dynamic programming algorithms for the zero-one knapsack problem. **Computing Journal of the Springer Verlag**, v. 25, n. 1, p. 29–45, Mar. 1980.

XIAO, S.; FENG, W. chun. Interblock GPU communication via fast barrier synchronization. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, 24., 2010, Atlanta. **Proceedings...** Piscataway: IEEE, 2010. p. 1–12.

YUNG, R.; RUSU, S.; SHOEMAKER, K. **Future trend of microprocessor design**. Firenze: Intel, Sep. 2002. Disponível em: <http://www.imec.be/esscirc/ESSCIRC2002/presentations/Slides/C00.06.pdf>. Acesso em: 11 mar. 2013.

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. REGISTRO Nº	4. Nº DE PÁGINAS
DM	12 de junho de 2013	DCTA/ITA/DM-016/2013	101

5. TÍTULO E SUBTÍTULO:

Algoritmos para o problema *Subset-Sum* em GPU

6. AUTOR(ES):

Vitor Venceslau Curtis

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica. Divisão de Ciência da Computação – ITA/IEC

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

GPU; CUDA; Algoritmo Paralelo; Subset-sum; Problema da Mochila

9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Problema da soma de subconjuntos; Problema da mochila; Algoritmos; Coprocessadores; Programação paralela; Programação matemática; Computação.

10. APRESENTAÇÃO: (**X**) **Nacional** ( ) **Internacional**

ITA, São José dos Campos. Curso de Mestrado. Programa de Pós-Graduação em Engenharia Eletrônica e Computação. Área de Informática. Orientador: Carlos Alberto Alonso Sanches. Defesa em 11/06/2013. Publicada em 2013

11. RESUMO:

Este trabalho utiliza o problema *subset-sum* (SSP) como estudo de caso, com o objetivo de analisar a complexidade de paralelização em Unidades de Processamento Gráficas (GPU). O SSP foi escolhido por pertencer à classe dos problemas NP-Completo, possuir grande necessidade de memória e não ter cálculo de ponto flutuante, além de ser amplamente estudado na área acadêmica devido a sua importância prática e teórica. Estas características representam um desafio para paralelização em GPUs, pelo fato de serem especialistas em cálculos de ponto flutuante e por possuir pouca quantidade de memória em relação ao grande número de núcleos. Basicamente, são apresentados 3 novos algoritmos, implementados em linguagem CUDA C, com baixo consumo de memória: somente $O(n + m - w_{\min})$, onde $m = \min\{c, c - \sum_{i=1}^{n} w_i\}$, c é a capacidade da mochila, n é a quantidade de itens, w_i é o i-ésimo peso e $w_{\min}$ é o menor peso, ao invés de $O(nc)$ do paradigma de Bellman, referentes aos algoritmos do estado da arte implementados na mesma arquitetura. Esta característica permite um ganho significativo na quantidade de instâncias solucionáveis, além do melhor tempo computacional. Para uma variedade de *benchmarks*, obteve-se bons tempos de execução em comparação com os melhores resultados práticos conhecidos até agora. Isto foi possível graças a um novo método para a solução do SSP, permitindo sua computação em tempo $O(n)$ e mesmo espaço, caso c processadores sejam utilizados.

12. GRAU DE SIGILO:

(**X**) **OSTENSIVO** ( ) **RESERVADO** ( ) **CONFIDENCIAL** ( ) **SECRETO**