Capítulo 4

Fundamentos de Computação Acelerada com CUDA C/C++

Arthur Francisco Lorenzon (UFRGS)

Abstract

This chapter highlights the fundamental tools and techniques for accelerating applications written in C/C++ languages to run on massively parallel architectures with CUDA. With CUDA, developers are able to dramatically accelerate the computation of applications on GPU (graphic processing unit) architectures. The Chapter has the following learning objectives: (I) writing parallel code to run on the GPU; (II) Expose and express data and instruction level parallelism in C/C++ applications using CUDA; (III) Use CUDA-managed memory and optimize memory migration using asynchronous prefetching; and (IV) Use concurrent streams for instruction-level parallelism.

Resumo

Este capítulo destaca as ferramentas e técnicas fundamentais para acelerar aplicações escritas em linguagens C/C++ para execução em arquiteturas massivamente paralelas com CUDA. Com CUDA, desenvolvedores são capazes de acelerar dramaticamente a computação de aplicações em arquiteturas GPU (graphic processing unit). O Capítulo tem os seguintes objetivos de aprendizagem: (I) escrever código paralelo para ser executado na GPU; (II) Expor e expressar paralelismo no nível de dados e instruções em aplicações C/C++ usando CUDA; (III) Utilizar o CUDA-managed memory e otimizar a migração de memória usando prefetching assíncrono; e (IV) Usar streams (fluxo de instruções) concorrentes para paralelismo no nível de instruções.

4.1. Introdução

Sistemas computacionais de alto desempenho são amplamente utilizados para executar aplicações de diversos domínios, que envolvem computação financeira, aplicações médicas, processamento de vídeo e imagem, entre outros. Esses sistemas geralmente são baseados em arquiteturas *multi-core* e *many-core* com memória e dispositivos de

computação altamente heterogêneos. Isso geralmente implica a existência de hierarquias de memória complexas e tecnologias que evoluem a cada ano. Assim, para continuar explorando o potencial dos sistemas computacionais de alto desempenho, as bibliotecas e interfaces de programação paralela se equipam com diversas políticas de gerenciamento de recursos [Navarro et al. 2020].

Neste cenário, a exploração do paralelismo em arquiteturas heterogêneas surge como um desafio para os desenvolvedores de *software* uma vez que estas arquiteturas são compostas de diferentes dispositivos computacionais. Um exemplo deste ambiente corresponde a combinação de CPUs de propósito geral (ou de alto desempenho) com unidades de processamento gráfico (*graphical processing units* - GPUs). As GPUs, também chamadas de aceleradores, fornecem um *throughput* de instruções e largura de banda de memória muito maiores que a CPU com similar consumo de energia. Assim, diversas aplicações aproveitam esses recursos para executar mais rapidamente na GPU do que na CPU. Adicionalmente, outros dispositivos de computação, como FPGAs (*field-programmable gate array*), também são muito eficientes em termos de energia, mas oferecem menor flexibilidade de programação do que GPUs [Knorst et al. 2021].

A diferença de recursos entre GPU e CPU existe porque elas são projetadas com objetivos diferentes em mente, conforme ilustrado na Figura 4.1. Enquanto a CPU é projetada para se destacar na execução de uma sequência de operações, chamada de *threads*, o mais rápido possível e pode executar algumas dezenas dessas *threads* em paralelo, a GPU é projetada para se destacar na execução de milhares de *threads* em paralelo, amortizando o desempenho mais lento fornecido por uma única por thread para obter maior *throughput*.

O interesse do uso de GPUs na computação de alto desempenho surgiu quando o potencial destes componentes foi combinado com uma linguagem de programação que tornou as GPUs mais fáceis de programar. Atualmente, o programador conta com diferentes interfaces de programação paralela que expõem o paralelismo em GPUs: CUDA (*Compute Unified Device Architecture*) [Sanders and Kandrot 2010] – criada pela NVI-

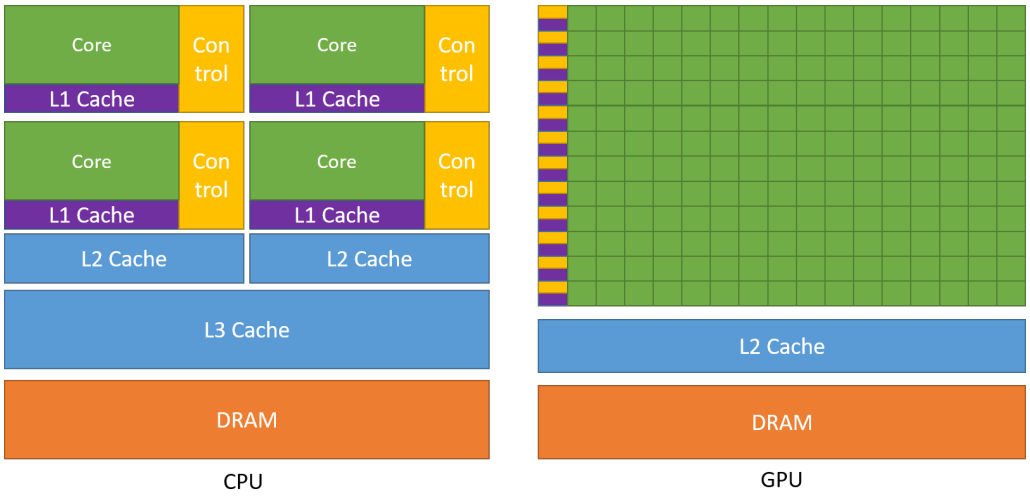

Figura 4.1. Comparação entre uma CPU e uma GPU - [Zone 2019]

DIA – e OpenCL (*Open Computing Language*) [Stone et al. 2010] fornecem maior flexibilidade no gerenciamento do paralelismo enquanto que OpenACC [Farber 2016] e, recentemente, OpenMP (*Open Multi-Processing*) [Chandra et al. 2001], são orientadas a diretivas de compilação permitindo o rápido desenvolvimento. Portanto, programadores de aplicações científicas e multimídia estão ponderando se devem usar CPUs ou GPUs.

Considerando a crescente demanda de desenvolvedores de software no aprendizado de tais interfaces de programação paralela para ambientes heterogêneos, este capítulo aborda os fundamentos de computação acelerada com CUDA C/C++. O material apresentado no decorrer do capítulo é baseado no *workshop* disponibilizado pelo instituto de *deep learning* (DLI) da NVIDIA: *Fundamentals of Accelerated Computing with CUDA C/C++*. Assim, uma breve fundamentação teórica sobre o modelo de programação e arquitetura de GPUs NVIDIA é destacada na Seção 4.2. A Seção 4.3 descreve fundamentos básicos de programação com CUDA, enquanto que as Seções 4.4 e 4.5 discutem técnicas de otimização que podem ser empregadas para acelerar a comunicação entre CPU-GPU e aumentar o nível de paralelismo. Por fim, a Seção 4.6 conclui este capítulo.

4.2. Fundamentação Teórica

GPUs são SIMD (*single instruction, multiple data*) *engines underneath*, onde existe um *pipeline* de instruções que opera como um pipeline SIMD (e.g., um processador vetorial). No entanto, a grande diferença comparado a processadores vetoriais é que o programador não precisa escrever código com instruções vetoriais, mas sim, programar as GPUs usando *threads*, o que é uma grande vantagem na perspectiva de programação. Assim, é possível programar de maneira mais eficiente, pois o programador só precisa se preocupar com a execução de múltiplas *threads* individuais ao invés de se preocupar com detalhes de baixo nível, e.g., *packing* de dados, instruções vetoriais e despachar as instruções no *hardware*.

Existem dois conceitos principais que são relacionados, mas que devem ser distinguidos: *modelo de programação* (*software*), que refere-se a como o programador escreve o código, podendo ser de maneira sequencial (e.g., *von Neuman*), explorando o paralelismo de dados (SIMD), *dataflow* e *multi-threaded*, por exemplo; E, *modelo de execução* (*hardware*), que refere-se em como o *hardware* executa o código, podendo ser de diferentes maneiras, como por exemplo, *out-of-order*, processador vetorial, processador *dataflow*, multiprocessador e processador *multithreaded*. Para melhor entende-los, as seguintes subseções descrevem brevemente as principais características do modelo de programação CUDA e da arquitetura de GPUs NVIDIA.

4.2.1. Modelo de Programação

O paralelismo explorado na GPU pode ser classificado como SPMD (*single program*, *multiple data*), onde múltiplas *threads* são criadas para executar partes do mesmo código de maneira concorrente, porém, operando sob diferentes dados. Cada *thread* tem seu próprio contexto, podendo ser tratada, reiniciada, executada de maneira independente das outras. Assim, com esta maneira de exploração do paralelismo, é dito que a GPU é chamada de máquina SIMT: *Single instruction, Multiple Threads*. A grande diferença para uma máquina SIMD é que em uma máquina SIMT, o programador não desenvolve o código

usando instruções vetoriais (SIMD) e sim, usando *threads* no modelo de programação SPMD. De uma maneira geral, podemos dizer que GPUs são máquinas SIMD onde o paralelismo no nível de operações vetoriais não é exposto ao programador. Existem duas grandes vantagens do modelo de execução SIMT empregado pelas GPUs: (*i*) Cada *thread* é tratada de maneira separada, o que é visto no processamento MIMD; e (*ii*) múltiplas *threads* podem ser agrupadas em threads de instruções SIMD, isto é, grupos de *threads* que supostamente irão executar a mesma instrução são dinamicamente agrupadas para obter o máximo dos benefícios oferecidos pelo processamento SIMD.

Em um modelo de execução SIMD, um único fluxo sequencial de instruções SIMD são executadas, onde cada instrução específica múltiplas entradas de dados. Por outro lado, no modelo de execução SIMT, múltiplos fluxos de instruções escalares (*threads*) são agrupadas dinamicamente em *threads* de instruções SIMD (*warps*, termo oficial CUDA/NVIDIA). As *threads* de instruções SIMD correspondem a um conjunto de *threads* que estão executando a mesma instrução (i.e., no mesmo PC). Tais *threads* de instruções SIMD são criadas de maneira transparente para o usuário, no sentido de que o programador não precisa se preocupar com esta definição no momento que está explorando o paralelismo no código. Nas arquiteturas atuais da NVIDIA, o tamanho do *warp* é de 32 *threads*, enquanto que em arquiteturas AMD, este número é de 64 *threads*. No entanto, as GPUs não possuem hardware disponível para executar simultaneamente todas as *threads* de instruções SIMD criadas em uma aplicação. Portanto, a GPU aplica *fine-grained multithread*, onde estas *threads* são intercaladas no mesmo *pipeline*.

Existem diferentes interfaces de programação e bibliotecas disponíveis para auxiliar o programador no desenvolvimento de código paralelo para executar na GPU, como por exemplo, CUDA e OpenACC. CUDA fornece um paradigma de codificação que estende linguagens como C, C++, Python e FORTRAN, possibilitando a execução de código massivamente paralelo nas GPUs NVIDIA. CUDA acelera as aplicações com pouco esforço, possuindo um ecossistema de bibliotecas altamente otimizadas para *deep neural networks*, BLAS, análise de grafos, FFT e muito mais, além de fornecer ferramentas de *profiling*. A Seção 4.3 destaca os fundamentos da programação paralela com CUDA, enquanto que as Seções 4.4 e 4.5 discutem características avançadas da interface.

4.2.2. Arquitetura e Organização da GPU

Uma arquitetura GPU é normalmente composta de dois componentes principais: (*i*) Memória Global, que é análoga a RAM em um servidor CPU, sendo acessível por ambos GPU e CPU; e (*ii*) processadores SIMD *multithreaded*, onde a computação é realizada. Estes processadores são chamados de *streaming multiprocessors* (SMs) nas arquiteturas NVIDIA e cada SM tem sua própria unidade de controle, registradores, *pipelines* de execução, *caches*, entre outros componentes de *hardware*. Considerando que a nomenclatura que envolve a discussão de GPUs pode variar de acordo com a bibliografia e empresa (e.g., NVIDIA e AMD utilizam nomenclaturas diferentes para se referir a mesma informação), o resto deste texto considera a nomenclatura utilizada pela NVIDIA.

SMs são processadores de propósito geral porém com uma frequência de operação mais baixa. Um SM é composto basicamente dos seguintes componentes (no entanto, é importante destacar que, a cada nova versão da arquitetura da NVIDIA, diferenças signi-

Figura 4.2. SM da arquitetura Ampere da NVIDIA - [Zone 2019]

ficativas são realizadas nos componentes internos a um SM). Estas diferenças são discutidas abaixo e podem ser acompanhadas na Figura 4.2, que apresenta um SM da arquitetura Ampere da NVIDIA.

- **Núcleos de processamento**: podendo ser sub-divididos em *Cuda cores, Ray-Tracing cores* e *Tensor cores*. Cada GPU NVIDIA contém centenas ou dezenas de *CUDA cores*, onde um *CUDA core* não pode buscar ou decodificar instruções. Ele apenas faz requisições, computa sobre os dados e responde com o resultado. Assim, *CUDA cores* contém unidades de ponto flutuante de precisão simples, dupla, unidades funcionais especiais, unidades lógicas, unidades de *branch*, entre outros componentes. Por outro lado, *Tensor Cores* apresentam computação de multi-precisão para inferência eficiente em inteligência artificial e aprendizado de máquina. Por fim, *Ray-Tracing Cores* são unidades aceleradoras dedicadas para realizar operações de *ray-tracing* e são exclusivos para placas gráficas NVIDIA RTX.
- Milhares de registradores de 32 bits;
- ***Warp schedulers***: unidades de escalonamento e despacho para conjuntos de threads;
- ***Caches***: *Cache L0 de instrução*, encontrada em arquiteturas mais atuais, que é privada a cada bloco de processamento; *L1 data cache*, que é privada a cada SM,

com baixa latência de acesso e alta largura de banda, podendo ser reconfigurada; *Constant cache*, para comunicar as leituras de uma memória somente leitura;

Adicionalmente, externo ao SM, podem ser encontradas as seguintes memórias: *L2 data cache*, que é compartilhada entre todos os SMs da GPU, usada para compartilhar dados, constantes e instruções; e *RAM*, consistindo da memória global.

O número de *warps* que podem ser executadas em paralelo é dependente da quantidade de *CUDA cores* disponíveis na arquitetura. Por exemplo, se existirem 8 *CUDA cores* em cada SM, então as 32 *threads* do *warp* levarão 4 ciclos para concluir a execução (8 *threads* ativas por ciclo). Como esta é uma definição utilizada pelo *hardware*, o número de *warps* não pode ser mudado pelo programador. Assim, para fornecer maior flexibilidade ao programador, a sequencia de operações SIMD (*CUDA threads*) são agrupadas em blocos de *threads* (e no nível de hardware elas são dinamicamente distribuídas em *warps*). Portanto, os programadores podem definir o número de *CUDA threads* por bloco assim como o número de blocos que são criados na chamada da função que irá executar na GPU.

Um bloco de *threads* consiste de uma parte do laço vetorizado que será executado em SMs, composto de um ou mais *warps* onde a comunicação acontece via memória local. Um conjunto de blocos de threads é denominado *Grid* (*loop* vetorizado). Assim, para oferecer mais oportunidades para o *hardware* da GPU gerenciar a execução e explorar o paralelismo disponível, um *grid* é decomposto em múltiplos blocos de *threads*. Um bloco de *thread* é então atribuído pelo escalonador de *warps* ao SM, que executa este código. O programador informa ao escalonador do *warp*, que é implementado em hardware, quantos blocos de *threads* devem ser executados.

Uma vez que SMs são processadores completos com PCs separados, uma GPU pode ter de uma a várias dezenas de SMs. Por exemplo, o sistema Pascal P100 tem 56, enquanto que chips menores podem ter apenas um ou dois. Portanto, para fornecer escalabilidade transparente entre modelos de GPUs com um número diferente de SMs, o escalonador de blocos de *threads* é responsável por atribuir os blocos de *threads* aos SM. Uma vez que o bloco de *threads* foi escalonado para um SM, o escalonador do *warp* sabe quais *warps* estão prontos para executar e os envia para uma unidade de despacho. Assim, a GPU tem dois níveis de escalonadores de *hardware*: (*i*) o *escalonador de blocos de threads* que atribui blocos de threads a SMs e (*ii*) o *escalonador de warps* interno ao SM, que escalona os *warps* quando estiverem prontos para execução.

Como por definição os *warps* são independentes um dos outros, o escalonador de *warps* pode escolher qualquer um que esteja pronto para execução. Para tanto, ele inclui um algoritmo de *scoreboard* para manter informação de até 64 *warps* e decidir quais estão prontos para execução. Uma vez que a latência da memória de instruções é variável devido aos *hits* e *misses* nas *caches* e TLB (*translation lookaside buffer*), uma *scoreboard* é usada para determinar quando as instruções estão completas. A suposição dos projetistas de GPU é que as aplicações que rodam na GPU têm tantos *warps* que o *multithreading* pode ocultar a latência de acesso a DRAM e aumentar a utilização de processadores SIMD *multithreaded*.

4.3. Acelerando Aplicações com CUDA C/C++

4.3.1. Escrevendo Códigos para a GPU

CUDA fornece extensões para as principais linguagens de programação, o que é o caso deste mini-curso, C/C++. Estas extensões da linguagem permitem que os desenvolvedores modifiquem o código fonte para executar em uma GPU. O trecho de código mostrado no Algoritmo 1 contém duas funções além da `main`: `CPUFunction`, que irá executar na CPU; e `GPUFunction`, que irá executar na GPU. Normalmente, o código executado na CPU é referenciado como código do *host*, enquanto que o código que executará na GPU é denominado como código do *dispositivo*. A partir do Algoritmo 1, pode-se destacar algumas características da programação paralela com CUDA:

- A palavra chave `__global__` (linha 6) indica que a respectiva função irá executar na GPU e pode ser invocada globalmente, isto é, pela CPU ou pela GPU. Note que as funções definidas com esta palavra chave devem retornar tipo `void`.
- Ao executar `GPUFunction`$<<<NUM\_BLOCOS, NUM\_THREADS>>>$();"(linha 13), dizemos que esta função é um *kernel* que será despachado para execução na GPU. Em conjunto com a chamada do *kernel*, o programador deve fornecer uma configuração de execução, que é feita usando a sintaxe $<<<$ ... $>>>$. Esta configuração permite a especificação da hierarquia de *threads* para a execução do respectivo *kernel*: o primeiro parâmetro informa o número de blocos que serão criados; o segundo parâmetro representa o número de *threads* que serão criadas por bloco. Parâmetros adicionais e opcionais podem ser informados, os quais serão discutidos na Seção 4.5.
- Diferentemente de muitos códigos paralelos escritos em C/C++, o despacho de *kernels* é assíncrono, isto é, o código da CPU continuará executando sem aguardar o término da execução do *kernel* na GPU. Portanto, a função `cudaDeviceSynchronize()` (linha 14), fornecida pelo *runtime* do CUDA, é usada para que o código da CPU aguarde até que o *kernel* finalize sua execução na GPU para então continuar a execução.

Algorithm 1 Primeiro programa em CUDA

```
 1:
 2: function VOID CPUFUNCTION(...)
 3:     printf("Esta função está definida para executar na CPU.");
 4: end function
 5:
 6: function __GLOBAL__ VOID GPUFUNCTION(...)
 7:     printf("Esta função está definida para executar na GPU.");
 8:     printf("Thread %d do bloco %d.", threadIdx.x, blockIdx.x);
 9: end function
10:
11: function INT MAIN(...)
12:     CPUFunction();
13:     GPUFunction<<<NUM_BLOCOS,NUM_THREADS>>>();
14:     cudaDeviceSynchronize();
15: end function
```

A plataforma CUDA é fornecida com o compilador *nvcc*, que é utilizado para compilar aplicações aceleradas com CUDA através do comando `nvcc -arch=sm_70 -o out app-cuda.cu`. Onde, a *flag* `arch` indica para qual arquitetura o arquivo deve ser compilado de maneira otimizada.

4.3.2. Hierarquia de *Threads* em CUDA

A configuração de execução permite a especificação de detalhes referentes a execução do *kernel* na GPU. Mais precisamente, ela permite ao programador definir quantos grupos de *threads* (i.e., blocos) e quantas *threads* serão criadas em cada bloco. A sintaxe para definição da configuração é a seguinte: `<<<NUM_BLOCOS, NUM_THREADS>>>`. O código do respectivo *kernel* será executado por todas as *threads* criadas no despacho do *kernel*. Assim, o total de *threads* que serão criadas corresponde ao produto do número de blocos e *threads* por bloco.

Cada *thread* criada recebe um índice interno ao seu bloco, iniciando pelo identificador 0. Adicionalmente, cada bloco recebe um índice, iniciando em 0. Da mesma maneira que as *threads* são agrupadas em um bloco, os blocos também são agrupados em uma *grid*, que é a mais alta entidade na hierarquia de *threads* em CUDA. De uma maneira resumida, os *kernels* de um programa CUDA são executados em uma *grid* de um ou mais blocos, cada um deles contendo o mesmo número de uma ou mais *threads*. A Figura 4.3 exemplifica a hierarquia de *Threads* em CUDA. Uma CUDA *Thread* é executada em um *CUDA Core*. Um bloco de CUDA *threads* é atribuído para execução por um SM. E, um *kernel* é encaminhado para execução em uma GPU. Durante a execução de um *kernel*, as *threads* têm acesso a variáveis especiais para identificar os seus índices interno ao bloco (`threadIdx.x`) e o índice do bloco na *grid* (`blockIdx.x`), conforme descrito no Algoritmo 1.

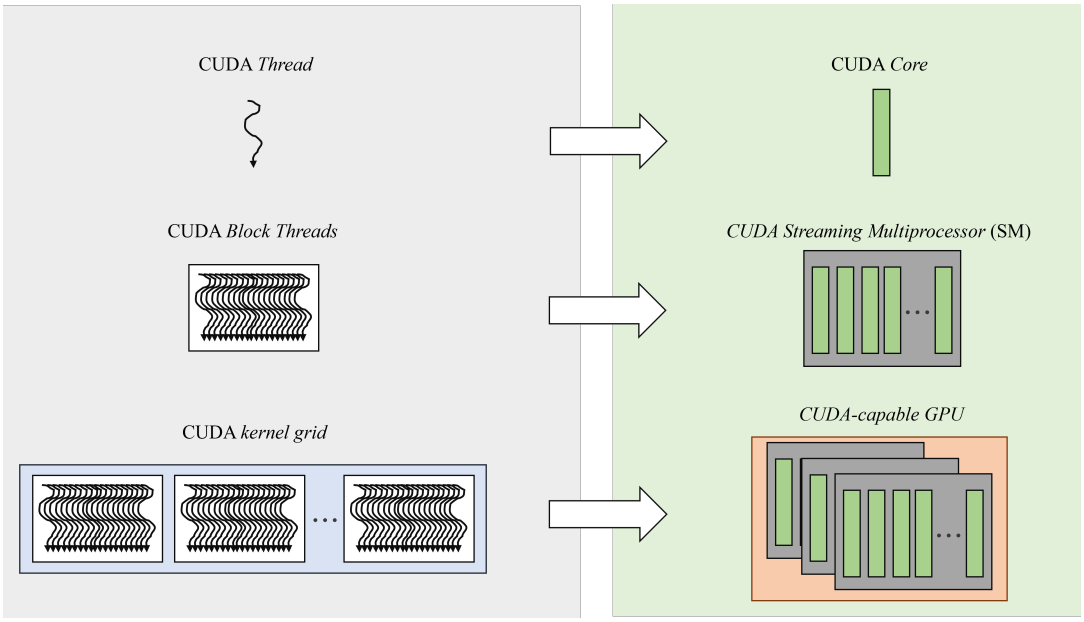

Figura 4.3. Execução de um Kernel na GPU - [Zone 2019]

Ao programar com CUDA, existe um limite de 1024 *threads* que podem ser criadas em um bloco. Assim, para aumentar o grau de exploração de paralelismo, há a necessidade de coordenar a execução de múltiplos blocos de *threads*. Para tanto, as *threads* executando um *kernel* escrito em CUDA têm acesso a variável especial `blockDim.x`, que fornece o número total de *threads* em um bloco. Esta variável pode ser usada em conjunto com as descritas anteriormente para aumentar o grau de paralelismo via a organização da execução paralela entre múltiplos blocos de múltiplas *threads* com a expressão: `threadIdx.x + blockIdx.x * blockDim.x`.

4.3.3. Alocando Memória Unificada para ser Acessada na GPU e CPU

As versões mais recentes de CUDA (a partir da 6.0, inclusive) facilitam a alocação de memória disponível tanto para a CPU quanto para a GPU através da memória unificada (*unified memory* - UM) [Chien et al. 2019]. Embora existam diversas técnicas intermediárias e avançadas para o gerenciamento de memória que podem fornecer desempenho ótimo, a técnica básica de gerenciamento de memória em CUDA é capaz de fornecer desempenho satisfatório em muitas aplicações com quase nenhuma sobrecarga de programação ao desenvolvedor. Assim, para alocar e liberar memória, além de obter um ponteiro que pode ser referenciado tanto no código do *host* quanto no dispositivo, as funções `cudaMallocManaged` e `cudaFree` são utilizadas, respectivamente. Um exemplo da aplicabilidade destas duas funções pode ser observado no Algoritmo 2 na utilização dos vetores `a` e `b`. A partir da alocação de memória, nas linhas 13 e 14, as duas variáveis podem ser acessadas tanto na CPU quanto na GPU, sem a necessidade do uso de funções específicas para transferência de dados entre a CPU e GPU (e.g., `cudaMemCpy`). Por fim, a liberação de memória ocorre nas linhas 19 e 20.

Em palavras simples, a UM fornece ao usuário a visão de um único espaço de memória que é acessível por todas as GPUs e CPUs no sistema, conforme ilustrado na Figura 4.4. Quando a memória unificada é alocada, os dados ainda não residem no dispositivo nem no *host*. Quando um deles tentar acessar tais dados, ocorrerá um *page fault*, momento em que o *host* ou dispositivo irá migrar os dados necessários em lotes. De modo similar, a qualquer momento em que a CPU ou GPU tentar acessar uma posição de memória que ainda não reside nela, *page faults* irão ocorrer, acionando a migração dos dados.

A capacidade de migrar os dados da memória sob demanda é extremamente útil para facilitar o desenvolvimento de aplicações. Além disso, ao trabalhar com dados que exibem padrões de acesso esparsos, por exemplo, quando não é possível saber quais dados devem ser trabalhados até que a aplicação seja executada e para cenários em que os dados

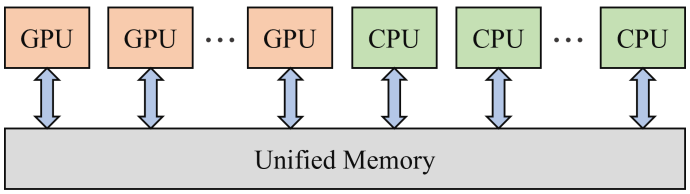

Figura 4.4. Diagrama de Memória Unificada - [Han and Sharma 2019]

podem ser acessados por várias GPUs em um sistema multi-GPU, a migração de memória sob demanda é extremamente benéfica. Por outro lado, em aplicações onde os dados necessários são conhecidos a priori e onde grandes blocos contíguos de memória são necessários, o sobrecusto do *page-fault* e migração de dados sob demanda incorre em um alto custo no desempenho que é melhor evitar. Este assunto é discutido com maior profundidade na Seção 4.4.

4.3.4. Acelerando Estruturas de Repetição

Estruturas de repetição representadas através de laços `for` em aplicações otimizadas para a CPU normalmente estão prontas para serem aceleradas na GPU: em vez de executar cada iteração do laço de maneira serial, uma após a outra, cada iteração do laço pode ser executada em paralelo por sua própria *thread*. Assim, para paralelizar um laço de repetição com CUDA, duas etapas devem ser seguidas: (*i*) re-escrever o *kernel* para fazer o trabalho de uma única iteração do laço; e (*ii*) como o *kernel* será independente de outros *kernels* em execução, a configuração de execução deve garantir que o *kernel* execute o número correto de vezes, por exemplo, o número de vezes que o laço de repetição deveria iterar.

Dependendo da aplicação que está sendo paralelizada, a configuração de execução pode não ser capaz de criar o número exato de *threads* necessários para a paralelização do laço. Considere o seguinte cenário como exemplo: a paralelização de um laço de repetição com 1000 iterações. Conforme mencionado anteriormente, blocos que possuem um número múltiplo de 32 *threads* tendem a apresentar melhor desempenho devido ao tamanho do *warp* e da maneira como estes são escalonados nos SMs. Assumindo que queremos despachar blocos com 256 *threads* (múltiplo de 32), não existirá um número de

Algorithm 2 Acelerando um laço de repetição em CUDA

```
1:
2: function __GLOBAL__ VOID MULTIPLICA(int valor, int *a, int *b, int N)
3:     int id = threadIdx.x + blockIdx.x * blockDim.x;
4:     int gridStride = gridDim.x * blockDim.x;
5:     for (int i = id; i < N; i += gridStride) do
6:         a[i] = valor * b[i];
7:     end for
8: end function
9:
10: function INT MAIN(...)
11:     int N = 10000, *a, *b, valor = 5;
12:     size_t size = N * sizeof(int);
13:     cudaMallocManaged(&a, size);
14:     cudaMallocManaged(&b, size);
15:     size_t num_threads = 256;
16:     size_t num_blocks = 32;
17:     multiplica<<<num_blocos, num_threads>>>(valor, a, b, N);
18:     cudaDeviceSynchronize();
19:     cudaFree(a);
20:     cudaFree(b);
21:     return 0;
22: end function
```

blocos que produzirá o total exato de 1000 *threads* na *grid* (e.g., 4 blocos resultarão no total de 1024 *threads*).

Este cenário pode ser tratado da seguinte maneira:

1. Escrever uma configuração de execução que cria mais *threads* do que o necessário para computar o *kernel*.

2. Passar um valor `N` como um argumento para o *kernel*, que representa o tamanho total dos dados que serão processados, ou o total de *threads* que são necessárias para finalizar o trabalho.

3. Após, calcular o índice de cada *thread* na *grid* (usando `threadIdx.x + blockIdx.x * blockDim.x`), e apenas realizar o trabalho se o índice não exceder `N`.

De maneira similar, há situações em que o número total de *threads* criadas para executar um *kernel* é menor do que o tamanho do conjunto de dados. Como um simples exemplo, considere um vetor com 1000 elementos e uma *grid* com 250 *threads*. Neste cenário, cada *thread* na *grid* deverá ser usada quatro vezes para garantir a execução correta do *kernel*. Para tratar tal cenário, o método de *grid-stride loop* é utilizado dentro do *kernel*.

Em um *grid-stride loop*, cada *thread* irá: (*i*) calcular com seu identificador único dentro da *grid*; (*ii*) realizar a operação no elemento representado pelo índice da *thread*; e (*iii*) adicionar a este índice o número total de *threads* que existem na *grid* e repetir até que o índice esteja fora do intervalo do vetor. O Algoritmo 2 destaca este cenário na função `multiplica`: na linha 3, calcula-se o índice de cada *thread* na *grid*; na linha 4, obtém-se o deslocamento do *grid-stride loop*, o qual será utilizado como incremento no laço de repetição da linha 5.

4.3.5. Tratamento de Erros

Como em qualquer aplicação, o tratamento de erros em códigos implementados com CUDA é essencial. A grande maioria das funções (e.g., funções de gerenciamento de

Algorithm 3 Tratamento de Erros

```
1: ...
2: cudaError_t err1, err2;
3: err1 = cudaMallocManaged(&a, size);
4: if (err1 != cudaSuccess) then
5:    printf("Error:  %s", cudaGetErrorString(err1));
6: end if
7:
8: GPUFunction<<<1, 1>>>();
9: err2 = cudaGetLastError();
10: if (err2 != cudaSuccess) then
11:    printf("Error:  %s", cudaGetErrorString(err2));
12: end if
13: cudaDeviceSynchronize();
14: ...
```

memória) retorna um valor do tipo `cudaError_t`, que pode ser utilizado para verificar a ocorrência de erro durante a execução da função. Um exemplo deste cenário é demonstrado no Algoritmo 3 para a execução da função `cudaMallocManaged`, na linha 3. O retorno da função é então armazenado na variável `err1` para então ser verificada na linha 4. Caso a função não tenha sido executada corretamente, a função `cudaGetErrorString` é utilizada para obter informações do erro.

Um outro cenário ocorre quando a função a ser executada é do tipo `void`, a qual não retorna um valor do tipo `cudaGetLastError`. Este cenário, descrito nas linha 8-12 do Algoritmo 3, acontece no despacho dos *kernels* que irão executar na GPU. Assim, para verificar se aconteceu algum erro durante o despacho do *kernel*, por exemplo, CUDA fornece a função `cudaGetLastError`, que é utilizada para obter informações do último erro que ocorreu durante a execução.

4.3.6. Obtendo Informações da GPU

Conforme destacado na Seção 1.2, as GPUs na qual as aplicações CUDA executam possuem unidades de processamento chamadas de *streaming multiprocessors*, ou SMs. Durante a execução do *kernel*, os blocos de *threads* são alocados nos SMs para execução. Assim, para dar suporte à capacidade da GPU de executar o maior número possível de operações em paralelo, os maiores ganhos de desempenho normalmente podem ser obtidos escolhendo um tamanho de *grid* que tenha um número de blocos múltiplo do número de SMs. Além disso, os SMs criam, gerenciam, escalonam e, executam grupos de 32 *threads* dentro de um bloco (*warp*). Assim, é importante destacar que escolher um tamanho de bloco que seja múltiplo de 32 tende a fornecer um desempenho melhor para a aplicação.

Neste sentido, uma vez que o número de SMs pode variar de acordo com a GPU que está sendo usada e para fornecer portabilidade às aplicações, o número de SMs e outras informações úteis podem ser obtidas através de uma *struct* em C. O trecho de código no Algoritmo 4 destaca a obtenção de algumas propriedades da GPU: as linhas 2 e 3 são responsáveis por obter o identificador da GPU no sistema; nas linhas 5 e 6, a *struct* `props` é declarada e utilizada para receber as propriedades da respectiva GPU. Por fim, nas linhas 8 a 11 são destacadas algumas propriedades, como por exemplo a capacidade computacional da GPU, o número de SMs e o tamanho do *warp*. A lista completa de

Algorithm 4 Obtendo informações da GPU

```
 1: ...
 2: int deviceId;
 3: cudaGetDevice(&deviceId);
 4:
 5: cudaDeviceProp props;
 6: cudaGetDeviceProperties(&props, deviceId)
 7:
 8: int computeCapabilityMajor = props.major;
 9: int computeCapabilityMinor = props.minor;
10: int numberSMs = props.multiProcessoCount;
11: int warpSize = props.warpSize;
12: ...
```

propriedades pode ser obtida diretamente na documentação do CUDA [1].

4.4. Otimizando a Troca de Dados entre CPU e GPU

Quando a memória unificada é utilizada, a GPU pode acessar qualquer página da memória do sistema e, ao mesmo tempo, migrar os dados sob demanda para sua própria memória para acesso de alta largura de banda. No entanto, para obter o melhor desempenho possível com a memória unificada, é importante entender como funciona a migração de página sob demanda. Para tanto, considere o pseudocódigo descrito no Algoritmo 2. Nele, os dados dos vetores `a` e `b` são inicializados na CPU (função omitida do algoritmo por simplicidade). Então, os dados são migrados da memória da CPU para a GPU e acessados na GPU durante a computação do *kernel*.

No entanto, trabalhos da literatura têm mostrado que a memória unificada introduz um complexo mecanismo para tratamento de *page-fault* [Landaverde et al. 2014, Chien et al. 2019, Knap and Czarnul 2019]. Além disso, o *trashing* de memória que move as mesmas páginas entre as memórias se torna um gargalo no desempenho. A natureza da GPU em explorar o paralelismo massivo de dados exacerba ainda mais a sobrecarga de *page-fault*, uma vez que os processos param quando um *page fault* está sendo resolvido e múltiplas *threads* em diferentes *warps* acessando a mesma página podem causar várias falhas duplicadas.

Desta maneira, a interface CUDA introduziu um mecanismo de pré-busca assíncrona (*assynchronous prefetching*) de página. Esta técnica poderosa pode ser utilizada para reduzir o *overhead* das migrações de memória sob demanda entre CPU-GPU e GPU-CPU quando ocorrer um *page-fault*. Ao utilizar este mecanismo (i.e., `cudaMemPrefetchAsync()`, a migração de dados ocorre em segundo plano através de um *stream* CUDA para evitar a interrupção das *threads* que estão computando. Ao fazer isso, o desempenho dos *kernels* da GPU e funções da CPU pode ser melhorado devido à redução de *page fault* e consequente *overhead* da migração de dados sob demanda.

Um exemplo da aplicabilidade da pré-busca assíncrona de página é destacado no Algoritmo 5 para a computação dos vetores `a` e `out` na GPU. Inicialmente, os dados devem ser corretamente alocados através da função `cudaMallocManaged` (linhas 10 e 11). Então, para cada um dos vetores, deve-se adicionar uma chamada à função `cudaMemPrefetchAsync` para habilitar a pré-busca assíncrona de páginas. Esta função é chamada com os seguintes parâmetros nas linhas 12 e 13 para os vetores `a` e `out`, respectivamente: o primeiro parâmetro é a variável alvo da migração; o segundo parâmetro consiste no tamanho da estrutura que será migrada; por fim, o identificador do dispositivo GPU ativo é indicado.

Com os dados sendo migrados em segundo plano para a memória da GPU, no momento que o *kernel* `GPUFunction` for despachado para execução, os dados necessários para execução já estarão presentes na GPU. O respectivo *kernel* (omitido por simplicidade) irá produzir como saída o vetor `out`, que será então utilizado pela CPU. Assim, para otimizar a cópia dos dados do vetor `out` para a memória da CPU através da pré-busca assíncrona, a função `cudaMemPrefetchAsync` é inserida na linha 21. Cabe notar

[1] https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.htmlstructcudaDeviceProp

que o último parâmetro contém a variável disponível na interface do CUDA que define o dispositivo CPU como sendo o destino da migração dos dados (`cudaCpuDeviceId`).

4.5. Sobrepondo Computação com Comunicação através de CUDA *Streams*

Em CUDA, um *stream* é definido como uma série de comandos que serão executados em ordem. Diversas operações, como por exemplo, execução de *kernel* e algumas transferências de dados ocorrem em CUDA *streams*. Embora até este momento o texto não tenha entrado no mérito de *streams*, os códigos descritos anteriormente executam seus *kernels* interno ao *default stream*.

Um exemplo da execução de vários *kernels* é ilustrado na Figura 4.5. Quando o programador não define explicitamente um *stream*, o *default* será utilizado. Nele, nenhuma operação irá iniciar até que todas as operações despachadas anteriormente no dispositivo sejam concluídas. Além disso, qualquer operação no *default stream* deve ser concluída antes de qualquer outra operação começar. Assim, este modo não permite a sobreposição de computação com comunicação, limitando os potenciais ganhos de desempenho.

Neste sentido, para aumentar o nível de concorrência durante a execução de uma aplicação CUDA na GPU, os programadores CUDA podem criar e utilizar *non-default CUDA streams* em adição ao *default stream*. Ao fazer isto, múltiplas operações podem

Algorithm 5 Exemplo da pré-busca assíncrona de memória

```
1: function INT MAIN(...)
2:     int N = 1000, *a, *out, deviceId, numberSMs;
3:     size_t size = N * sizeof(int);
4:
5:     cudaGetDevice(&deviceId);
6:     cudaDeviceProp props;
7:     cudaGetDeviceProperties(&props, deviceId);
8:     numberSMs = props.multiProcessorCount;
9:
10:    cudaMallocManaged(&a, size);
11:    cudaMallocManaged(&out, size);
12:    cudaMemPrefetchAsync(a, size, deviceId);
13:    cudaMemPrefetchAsync(out, size, deviceId);
14:
15:    size_t num_threads = 256;
16:    size_t num_blocos = 32 * numberSMs;
17:
18:    GPUfunction<<<num_blocos, num_threads>>>(a, out, ...)
19:
20:    cudaDeviceSynchronize();
21:    cudaMemPrefetchAsync(out, size, cudaCpuDeviceId);
22:
23:    checkElements(c,N);
24:
25:    cudaFree(a);
26:    cudaFree(out);
27: end function
```

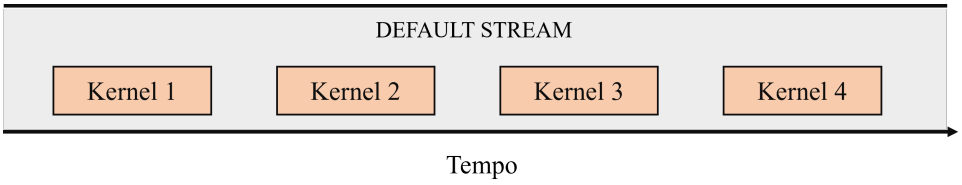

Figura 4.5. Comportamento da execução de diferentes *kernels* no *default stream*

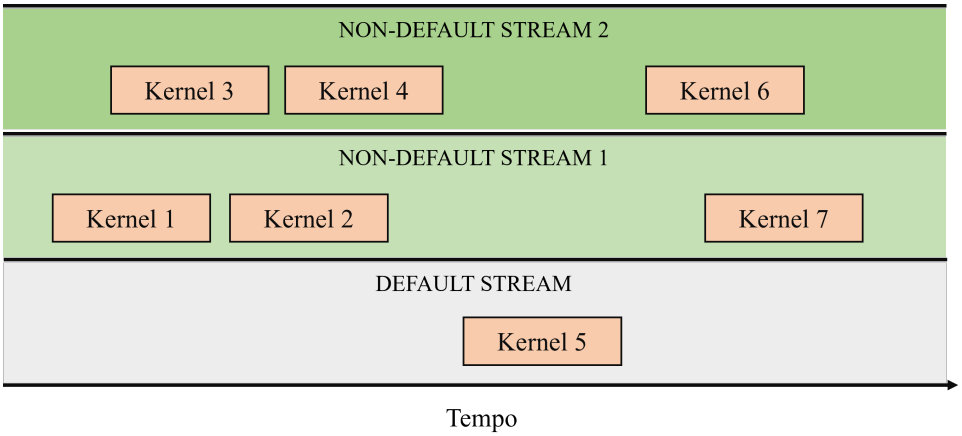

Figura 4.6. Explorando concorrência no nível de *streams*

ser realizadas de maneira concorrente, em *streams* diferentes, conforme ilustrado na Figura 4.6. Usando múltiplos *streams* pode adicionar uma camada extra de paralelização à aplicação, e oferecer mais oportunidades para otimização da aplicação. Existem algumas regras considerando o comportamento de CUDA *streams* que devem ser aplicados de modo a utiliza-los eficientemente:

- As operações realizadas internas a um *stream* ocorrem em ordem.
- Não existe a garantia de que as operações em diferentes *non-default streams* seja realizada em qualquer ordem específica relativa a cada uma.
- O *default stream* é bloqueante e irá aguardar todos os outros *streams* completar a execução antes de iniciar, e, irá bloquear a execução de outros *streams* até que seja finalizada sua execução.

Considerando que as operações em diferentes *streams* podem executar de maneira concorrente e ser intercaladas, o desempenho de uma aplicação pode ser melhorado de maneira significativa. Para exemplificar esta situação, considere a Figura 4.7 que destaca três operações: cópia assíncrona de dados do *host* para o dispositivo; a execução do *kernel* na GPU; e a cópia dos dados do dispositivo para o *host*. Quando apenas o *default stream* é utilizado (serial), não existe concorrência entre as operações. Ao adicionar múltiplos *non-default streams*, a concorrência pode ser explorada de diferentes maneiras. Na concorrência *2-way*, duas operações irão ocorrer no mesmo instante de tempo, como por exemplo, a execução de um *kernel* em um *non-default stream* sobreposta com

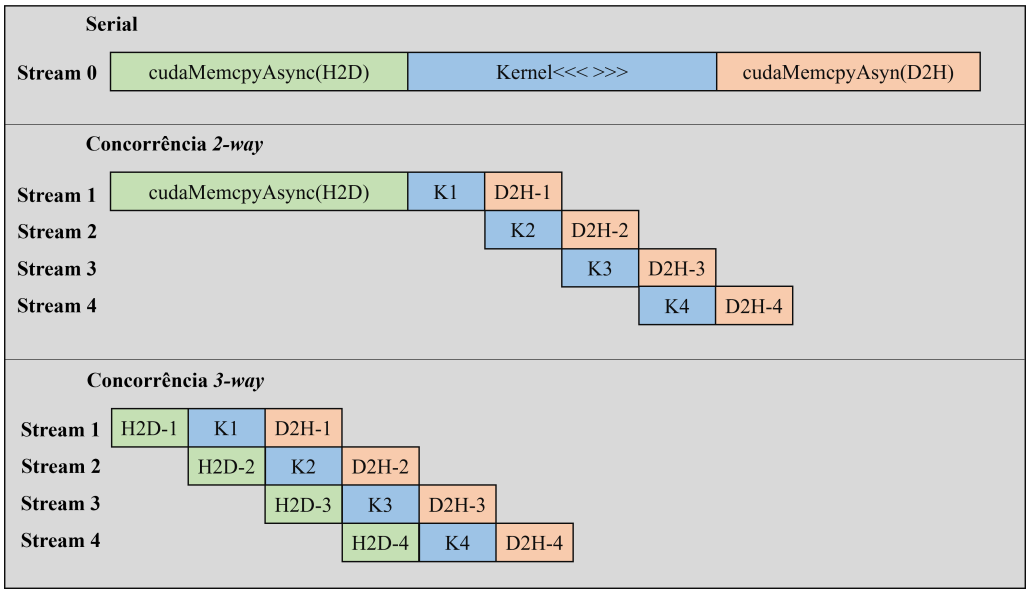

Figura 4.7. Quantidade de concorrência em múltiplos *streams*

a comunicação do dispositivo para o *host* em outro *stream*. Ao adicionar um novo nível de concorrência, (*3-way*), a cópia de dados do *host* para o dispositivo também passa a ser realizada de maneira sobreposta a computação.

Um exemplo da aplicabilidade de múltiplos *streams* em uma aplicação é destacado no Algoritmo 6. Inicialmente, um *stream* deve ser declarado com o tipo `cudaStream_t`. A criação de um *stream* de maneira assíncrona se dá através da função `cudaStream-Create`, que recebe como parâmetro um ponteiro para o identificador do *stream*. Então, para garantir que o despacho do *kernel* seja realizado em diferentes *streams*, o quarto parâmetro de configuração de execução deve conter o identificador do respectivo *stream*. Isto irá lançar o *kernel* para ser executado num *non-default stream*. O terceiro argumento da configuração de execução permite o programador fornecer o número de *bytes* na memória compartilhada que serão dinamicamente alocados por bloco para o respectivo *kernel*. Para deixar o número padrão de *bytes* alocado por bloco na memória compartilhada, basta incluir o valor 0 neste argumento. Por fim, o *stream* é finalizado através da função `cudaStreamDestroy`.

Algorithm 6 Explorando múltiplos *streams*

```
1: ...
2: for (int i = 0; i < totalStream; i++) do
3:    cudaStream_t stream;
4:    cudaStreamCreate(&stream);
5:
6:    GPUFunction<<<num_blocos, num_threads, 0, stream>>>();
7:
8:    cudaStreamDestroy(stream);
9: end for
10: ...
```

4.6. Conclusão

Considerando o avanço iminente das arquiteturas GPUs em sistemas de alto desempenho, *desktops* e até mesmo em dispositivos móveis, se torna extremamente importante o aprendizado de técnicas para tirar o maior proveito possível de tais dispositivos. Neste sentido, este capítulo de livro apresentou fundamentos relacionados a programação massivamente paralela com CUDA, cobrindo aspectos principais, como por exemplo, exploração do paralelismo em laços, otimização da troca de dados entre CPU-GPU, sobreposição de comunicação e computação para melhorar a concorrência e a utilização de múltiplos *streams* para elevar o nível de exploração de paralelismo. Por fim, recomenda-se a leitura de material fornecido pela NVIDIA em seus *blogs*, zona do desenvolvedor e documentação do CUDA para enriquecer ainda mais seus conhecimentos em programação paralela em ambientes heterogêneos.

Referências

[Chandra et al. 2001] Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., and McDonald, J. (2001). *Parallel programming in OpenMP*. Morgan kaufmann.

[Chien et al. 2019] Chien, S., Peng, I., and Markidis, S. (2019). Performance evaluation of advanced features in cuda unified memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 50–57. IEEE.

[Farber 2016] Farber, R. (2016). *Parallel Programming with OpenACC*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Han and Sharma 2019] Han, J. and Sharma, B. (2019). *Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++*. Packt Publishing.

[Knap and Czarnul 2019] Knap, M. and Czarnul, P. (2019). Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on nvidia pascal and volta gpus. *The Journal of Supercomputing*, 75(11):7625–7645.

[Knorst et al. 2021] Knorst, T., Jordan, M. G., Lorenzon, A. F., Rutzig, M. B., and Beck, A. C. S. (2021). Etcf – energy-aware cpu thread throttling and workload balancing framework for cpu-fpga collaborative environments. In *2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–8.

[Landaverde et al. 2014] Landaverde, R., Zhang, T., Coskun, A. K., and Herbordt, M. (2014). An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE.

[Navarro et al. 2020] Navarro, A., Lorenzon, A. F., Ayguadé, E., and Beltran, V. (2020). Enhancing resource management through prediction-based policies. In Malawski, M. and Rzadca, K., editors, *Euro-Par 2020: Parallel Processing*, pages 493–509, Cham. Springer International Publishing.

[Sanders and Kandrot 2010] Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[Stone et al. 2010] Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engg.*, 12(3):66–73.

[Zone 2019] Zone, N. D. (2019). Cuda toolkit documentation. *NVIDIA,[Online]. Available: http://docs. nvidia. com/cuda/cuda-c-best-practices-guide/# occupancy.[Acedido em Fevereiro 2015]*.