Capítulo 1

Programação em GPU no Ambiente Google Colaboratory

Ricardo Ferreira [1], Michael Canesche[1], Westerley Carvalho[1]

Resumo

Este trabalho apresenta um minicurso sobre a programação de GPU no Google Colaboratory (Colab). Tem como objetivo fornecer um material introdutório para o ensino e pesquisa com GPU no ambiente Colab, democratizando e desmistificando o acesso às GPUs. O Colab disponibiliza 4 tipos de GPU (K80, P4, P100 e T4) de forma gratuita e pode ser acessado, inclusive, por celular. Inicialmente, uma introdução à programação GPU é apresentada, juntamente com a configuração do Colab. Em seguida, vários exemplos de código são apresentados para ilustrar a execução, os recursos e as técnicas básicas de programação para GPU. Os laboratórios do minicurso também incluem material adicional com exercícios e sugestões de atividades extras.

1.1. Introdução

Nosso objetivo é fornecer um material introdutório para ensino e pesquisa com GPU no ambiente *Colab*, democratizando e desmistificando o acesso às GPUs. As seções e subseções do minicurso estão organizadas em laboratórios no *Colab*. Os laboratórios serão numerados e rotulados com o nome da seção correspondente do minicurso. Laboratórios extras como extensões do material do minicurso serão disponibilizados e atualizados. Este texto no formato PDF contém os links para os laboratórios (Clique aqui). Iremos também fornecer um repositório para consulta com exemplos para os níveis intermediário e avançado. Como iremos trabalhar em um ambiente colaborativo, a comunidade poderá contribuir com material de forma incremental.

Este minicurso está organizado da seguinte forma. Primeiro, a seção 1.2 apresenta vários recursos que podem ser explorados no *Google Colab*. A seção 1.3 ilustra a programação em GPU no ambiente *Google Colab* com exemplos introdutórios. A seção 1.4

[1]Apoio Financeiro: FAPEMIG, Nvidia, CNPq, Funarbe. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

apresenta algumas técnicas de instrumentação de código para medir desempenho da GPU e as características das arquiteturas das GPUs do *colab*. A seção 1.5 ilustra quatro exemplos de problemas com duas ou mais implementações em GPU. A seção 1.6 apresenta como automatizar a geração de gráficos com *scripts* utilizando os códigos da seção 1.5. Finalmente, a seção 1.7 apresenta as considerações finais e trabalhos futuros.

1.2. Google Colaboratory

A Google criou o *Google Colaboratory* ou simplesmente *Colab* para facilitar e difundir o ensino e a pesquisa em aprendizado de máquina. O *Colab* é um ambiente virtual na nuvem da Google onde o programador tem acesso gratuito a um *Jupyter notebook*. O projeto tem o nome *Jupyter* pois foi originalmente criado para o ensino das linguagens JUlia, PYThon e R. Posteriormente, o projeto foi estendido e suporta várias linguagens. Este ambiente é executado em um navegador e oferece vários recursos interessantes para ensino e pesquisa.

Ao criar uma sessão no *Colab*, o usuário tem acesso a um processador com dois núcleos, 12 GBytes de memória RAM e cache L3 de 40-50 Mbytes. Além disso, o usuário pode ter acesso a uma TPU ou uma GPU. No caso de abrir uma nova sessão, o usuário deve configurar o ambiente de execução se for utilizar a TPU ou a GPU como acelerador. Este recurso é importante para executar os códigos de aprendizado de máquina. O usuário também tem acesso a um sistema de arquivos com 30 a 300 Gbytes de espaço em disco e um terminal *Linux*.

A proposta deste minicurso é ilustrar como podemos fazer uso do ambiente *Colab* no ensino e pesquisa com GPUs. Primeiro, iremos apresentar alguns recursos que podem ser utilizados como facilitadores em laboratórios virtuais. O laboratório 2 ilustra os exemplos descritos nas próximas seções. A seção 1.2.1 mostra como configurar o ambiente para execução nas linguagens C, C++ e Java. A seção 1.2.2 mostra como gerenciar arquivos, carregar e gravar arquivos da sua conta no *Google drive*, de uma pasta em seu computador ou do *github*. Mostraremos também como gravar o conteúdo de uma célula de código para depois compilar com linha de comando (ou até mesmo instalar um compilador). Finalmente, a seção 1.2.3 ilustra como gravar a saída do código em um arquivo *csv* e depois exibir os resultados de forma gráfica (linhas ou barras) e/ou histogramas usando os recursos dos pacotes em Python.

1.2.1. Linguagens

O Google Colaboratory é amplamente utilizado para executar código em Python com as bibliotecas e as ferramentas de aprendizado de máquina. O ambiente têm dois tipos de células, as de texto e as células de código. Nativamente, a célula de código interpreta e executa Python. Os resultados podem ser visualizados logo abaixo da célula ao usar o comando *print*. Antes de começar o ensino de GPU e com a finalidade de poder comparar diferentes implementações com as versões em GPU, primeiro, iremos mostrar como compilar em outras linguagens (C, C++ e Java).

Nossa primeira contribuição como facilitador foi criar um pacote para configurar

as células do *Colab* para executar outras linguagens. O pacote está disponível no material do minicurso no Github. O usuário executa uma célula de código que faz uma cópia do material para sua pasta local e depois faz a instalação do pacote com o comando *% load_ext*. O laboratório 2 ilustra um código que calcula o produto escalar como inserir, compilar e executar códigos em C, C++ e Java. A configuração proposta para as células é bem simples. Basta inserir na primeira linha *%%gcc, %%cpp* e *%%java* para codificar em C, C++ e Java, respectivamente, como ilustrado na figura 1.1.

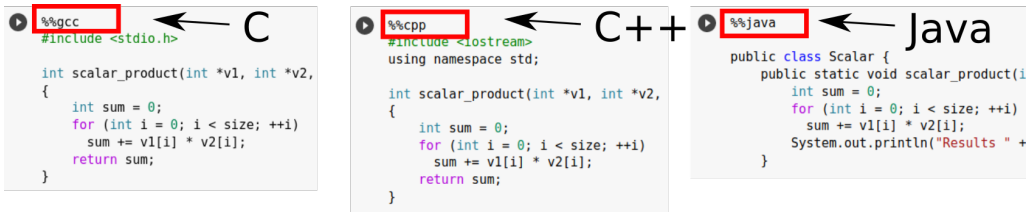

Figura 1.1. Como incluir código C, C++ e Java com a extensão proposta no minicurso.

Os compiladores utilizados para as linguagens C, C++ e Java foram, respectivamente, gcc na versão 7.5.0, g++ na versão 7.5.0 e o OpenJDK 11.0.8. Estes compiladores já estão instalados no *Colab*. Como o ambiente está sendo sempre atualizado, a versão atual de cada uma das linguagens pode ser verificada com a flag *-v* ou *–version* após os comandos de linhas *%%gcc, %%cpp* e *%%java*. Note que por questão de limitação do jupyter, uma célula mágica nunca pode ser vazia, sendo assim é necessário adicionar algum código ou comentário na célula.

1.2.2. Linha de comando e Sistema de Arquivos

Uma célula de código pode executar um comando no sistema operacional Linux, ou seja, a célula se comporta como um terminal e pode executar linhas de comando como listar os arquivos, instalar um pacote de biblioteca Python, compilar, etc. Para executar uma linha de comando basta adicionar o caracter "!"(ponto de exclamação) no início. Como por exemplo "!ls","!pwd"ou "!echo"para listar os arquivos, imprimir o diretório corrente ou imprimir uma mensagem na tela, respectivamente.

Entretanto, ao usar o caracter "!", a linha de comando é executada em uma sessão de terminal temporária. Alguns comandos como a navegação nos diretórios com o comando "cd" não surgem efeito se usados apenas localmente. Para esta situação devemos usar o caracter "%" no início com "%cd".

Além de linhas de comando, uma sessão do *Colab* conta com um sistema de arquivos e uma área em disco que pode variar de 30 GB a 300 GB. Um recurso interessante é montar sua pasta do *Google Drive* com um simples comando. Portanto é possível ler ou gravar diretamente na sua conta *Google Drive*. Outro comando permite carregar um arquivo do seu disco local na nuvem do *Colab*. Estes comandos são úteis para buscar um exemplo local ou armazenar os resultados da sua sessão. Além destas opções, um recurso que usaremos nos laboratórios é montar os exemplos em uma pasta *github* e com o comando *git clone* carregar na sua sessão *Colab*.

Finalmente, mostraremos também o comando "%%writefile filename" no início de uma célula de código. Este comando irá gravar o conteúdo da célula no arquivo "file-

name". Posteriormente, o arquivo pode ser compilado com uma linha de comando. Vários compiladores já estão pré-instalados no *Colab* como *gcc, g++, nvcc e java* dentre outros. Você também pode instalar um compilador na sua sessão e depois compilar seu arquivo na linguagem desejada.

O laboratório 2 apresenta vários exemplos com os recursos desta seção para execução de linhas de comando, acesso as suas pastas locais, a sua conta *Google Drive*, gravando arquivos e compilando como linha de comando como ilustrado na figura 1.2.

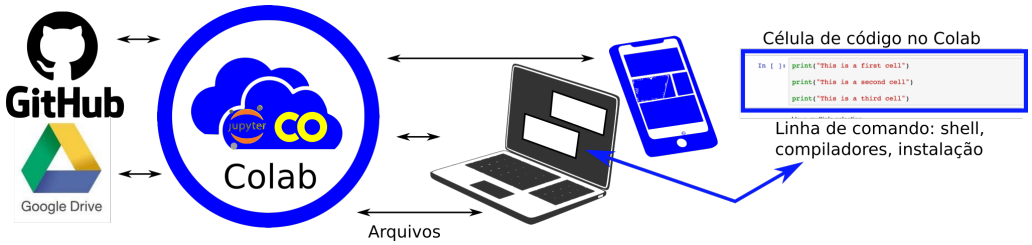

Figura 1.2. Colab, Github, Google Drive, sistema de arquivos locais.

1.2.3. Gráficos e Histogramas

As bibliotecas *Pandas* e *Matplotlib* do Python tem muitos recursos para manipulação de dados e visualização. Existem vários exemplos de uso disponíveis no Web. Esta seção apresenta de uma forma simples como exportar os resultados do seu código para posteriormente visualizá-los.

O fluxo proposto é bem direto. Primeiro, é necessário criar um arquivo texto no formato *csv (comma-separated values)* no seu código C, C++, CUDA, etc. O arquivo pode ter "," como separador ou outro caracter. Depois usaremos comandos em Python, onde temos que abrir o arquivo com a biblioteca *Pandas* com o método *read_csv*. Posteriormente, usando as funcionalidades da biblioteca *matplotlib* podemos facilmente gerar gráficos de barras, linhas, etc. Mostramos também como fazer um histograma. Usaremos estes recursos na seção 1.6 para mostrar como criar *scripts* para visualizar os resultados das suas execuções e outras métricas de desempenho.

O laboratório 2 ilustra os recursos desta seção como armazenar os resultados do seu código em um arquivo temporário para posteriormente gerar gráficos e visualizá-los como ilustrado na figura 1.3.

1.3. Introdução a programação GPU

Nesta seção apresentamos uma breve introdução à linguagem CUDA através do Google Colaboratory disposta em três subseções a seguir: a seção 1.3.1 apresenta os conceitos básicos do modelo de programação CUDA, a seção 1.3.2 aprofunda um pouco mais na organização das *threads* trazendo o conceito de warps e gerenciamento de memória, e na última seção 1.3.3 apresentamos um exemplo levemente mais elaborado que compila todos os conceitos básicos vistos até então.

Por fim, para iniciantes sugerimos a referência [Cheng et al. 2014] cujos códi-

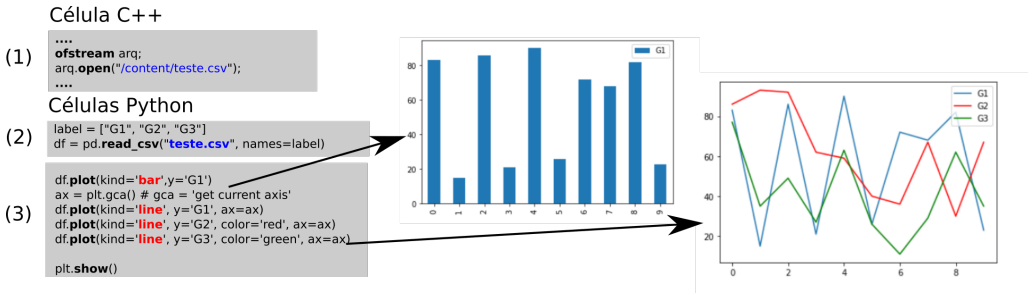

Figura 1.3. (1) Criar um arquivo no código C++, gerar resultados; (2) Importar o arquivo CSV em Python com Pandas; (3) Usar Matplotlib para apresentar os gráficos de barra, linha, etc.

gos estão disponíveis [2], o minicurso da SBC Técnicas de Otimização de Código para Placasde Processamento Gráfico do Prof. Fernando M.Q.Pereira [Pereira 2011], o repositório CUDA by practice de Edgar Garcia Cano que contém vários exemplos do nível introdutório que podem ser executados no *Colab*, o tutorial disponível no Blog de Jonathan Hui e o curso online da Udacity.

1.3.1. Kernels, Blocos e Threads

O laboratório 1.3.1 contém exemplos para os tópicos abordados nessa seção. O modelo de programação CUDA foi criado de forma que programadores já acostumados à linguagem C tenham facilidade para se adaptar. CUDA estende a linguagem C permitindo ao programador criar funções para executar na GPU. Estas funções são comumente denominadas pelo termo *kernels*. Neste texto iremos usar este termo para deixar claro qual parte do código é executado na GPU.

A figura 1.4 mostra de forma simplificada o modelo de computação heterogênea com GPU. A parte do código mais sequencial será executada na CPU (trechos em vermelho). Os trechos com paralelismo serão executados na GPU (trechos em azul). A CPU tem poucas unidades de execução e registradores, enquanto a GPU tem muitas unidades de execução (ALU) e registradores. Uma arquitetura complementa a outra formando um modelo heterogêneo com alto desempenho.

A grande vantagem da GPU é que ao disparar um kernel, o programador especifica, de forma simples, a execução concorrente de X cópias da sua função. O modelo de execução é chamado SIMT, da sigla em inglês *Single Instruction Multiple Threads*, onde um conjunto de threads executa a mesma instrução em paralelo.

Para criar um kernel é necessário utilizar o especificador *__global__* antes da declaração normal da função. Ao compilar, este trecho de código será mapeado para executar na GPU. A CPU controla o processo, portanto a GPU será chamada pela CPU. Para a chamada de um kernel no código da CPU temos que passar pelo menos 2 paramêtros específicos. A chamada do kernel segue o padrão da linguagem C com o nome e os parametros da função entre parênteses. Porém, entre o nome da função e os parâmetros, especificamos com uma tupla (B,T) entre os delimitadores $<<<$ e $>>>$ a quantidade de threads a serem executadas, onde B é o número de blocos e T o tamanho dos blocos em

[2] Clique aqui para os códigos do livro CUDA C professional programming

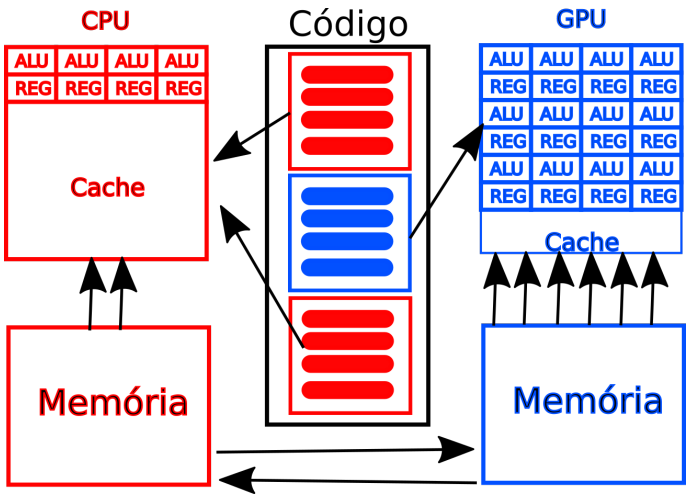

Figura 1.4. Computação Heterogênea CPU e GPU

threads. Por exemplo, a chamada *meu_kernel* $<<<2,4>>>(A,C)$ irá disparar 2 blocos de 4 threads cada, totalizando $4*2=8$ threads com os parâmetros A e C.

Cada *thread* tem um identificador único composto pela sua tupla B_i, T_i, onde B_i é número do bloco e t_i é o número da *thread*. A próxima seção apresenta mais exemplos sobre este tema.

Para exemplificar a definição de um kernel, apresentamos um programa simples que imprime "Hello World" na tela. Nele é possível ver que a chamada do kernel com 6 threads que foram organizados em 2 blocos de três threads cada. A figura 1.5(a) mostra o kernel que apenas imprime a mensagem "Hello...from GPU". A figura 1.5(b) mostra o código da CPU que inclui a chamada para o kernel com 2 blocos de 3 threads. A figura 1.5(c) mostra a saída da execução onde a mensagem é impressa 6 vezes, pois disparamos um total de 6 threads.

```
__global__ void helloFromGPU(){
    printf("Hello World from GPU!\n");
    return;
}
```

(a)

```
int main(){
    //Chamada do Kernel
    helloFromGPU<<<2,3>>>();
```

(b)

```
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
```

(c)

Figura 1.5. (a) Definição do kernel. (b) Chamada na função main(); (c) Saída do programa.

Como já mencionado, as *threads* são organizadas em blocos, que podem ser unidimensionais, bidimensionais ou tridimensionais. Nosso primeiro exemplo usou blocos unidimensionais. Dentro destes, as *threads* podem ser acessadas usando o identificador *threadIdx.x*. Cada bloco possui um limite de threads que ele pode conter, uma vez que todas as *threads* que pertencem a esse bloco devem estar no mesmo núcleo do processador e compartilham memória. Dependendo da placa utilizada, o tamanho do bloco pode variar entre 512, 1024 e 2048 threads. Todos os blocos serão do mesmo tamanho.

A figura 1.6(a) ilustra o kernel do exemplo anterior com uma pequena modifica-

ção. Fazendo acesso as variáveis *thread.Idx* e *block.Idx*, cada *thread* irá imprimir sua identificação única composta pelo número do bloco e o número da *thread*.

```
__global__ void helloFromGPU(){
    printf("Hello World from GPU! Block number: %d Thread number: %d\n", blockIdx.x ,threadIdx.x);
    return;
}
```

(a)

```
Hello World from GPU! Block number: 0 Thread number: 0
Hello World from GPU! Block number: 0 Thread number: 1
Hello World from GPU! Block number: 0 Thread number: 2
Hello World from GPU! Block number: 1 Thread number: 0
Hello World from GPU! Block number: 1 Thread number: 1
Hello World from GPU! Block number: 1 Thread number: 2
```

(b)

Figura 1.6. (a) A função kernel com impressão do número da *thread* e do bloco; (b) Saída impressa.

Apesar dessa limitação do número de threads por bloco, como podemos disparar milhares de blocos, uma GPU pode executar bilhões ou mesmo trilhões de threads em uma única chamada. Na nomeclatura da Nvidia, os blocos, por sua vez, são organizados em um grid que também pode ter de uma a três dimensões. O tamanho do grid (número de blocos) e o tamanho do bloco (seu número de threads), como já mencionamos, são as informações adicionais passadas junto à chamada do kernel dentro da estrutura $<<<$ $numBlocos, numThreads$ $>>>$. A figura 1.7 demonstra graficamente como os blocos são organizados no grid e como as *threads* são organizadas nos blocos.

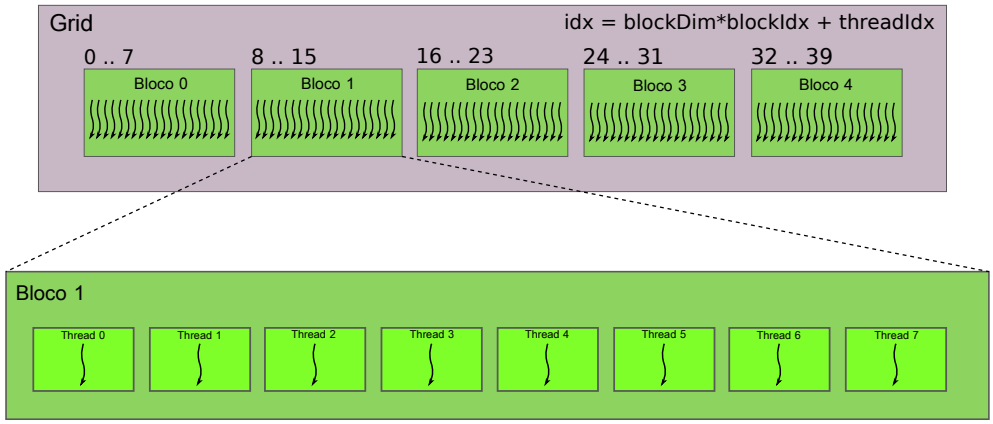

Figura 1.7. Disposição dos blocos no grid e das *threads* no bloco.

Os blocos de threads podem executar em qualquer ordem a priori. Para o exemplo da figura 1.6 podemos fazer a chamada com $<<<$ $32, 128$ $>>>$, onde $32 * 128 = 4096$ threads serão disparados. Cada execução poderá imprimir as mensagens em uma ordem diferente.

1.3.2. Gerenciamento de Memória e Warps

Exemplos para essa seção podem ser encontrados no laboratório 3.2. Cada *thread* possui uma memória local própria acessada apenas por ela. As variáveis locais serão armaze-

nadas em registradores com acesso rápido. Somente se o número de registradores disponíveis não for suficiente que estas variáveis irão para memória global, o que deve ser evitado ao máximo.

Cada bloco de threads também tem acesso a uma memória compartilhada (*shared memory*), que é visível para todas as *threads* dentro de um mesmo bloco, cujo tempo de vida é o mesmo do bloco. A memória compartilhada é declarada dentro do kernel. Além disso, todas as *threads* tem acesso à mesma memória global. As variáveis globais são passadas como parâmetros na chamada do kernel.

O gerenciamento de memória em CUDA é feito com funções similares às que são utilizadas na linguagem (malloc/free). Para alocar memória para variáveis que virão a ser acessadas pelo GPU, devemos usar então as funções cudaMalloc() e cudaFree(), que recebem como parâmetro um ponteiro para a variável e o tamanho do bloco de memória a ser alocado para ela. A figura 1.4 mostra que a CPU e GPU tem espaços diferentes de memória.

Como agora há uma distinção entre variáveis da CPU e variáveis da GPU, é usual utilizar os identificadores *host* e *device* para que seja possível distingui-las mais facilmente. Além disso, quando é necessário fazer a cópia do conteúdo de uma variável device para um variável host, é utilizada a função cudaMemcpy(). A figura 1.8(a) mostra um exemplo da chamada de cudaMemcpy(), note que essa função recebe um ponteiro para a posição de memória a ser copiada, um ponteiro para o destino, o tamanho do bloco de memória a ser copiado e uma tag que indica se o sentido da cópia é da CPU para a GPU (cudaMemcpyHostToDevice) ou da GPU para CPU (cudaMemcpyDeviceToHost). As variáveis d_A, d_B e d_C são variáveis da GPU e h_A, h_B e h_C são variáveis da CPU.

Na Figura 1.8(b) temos um exemplo simples de um programa que calcula a soma de dois vetores. Para isso, passamos para o kernel os dois vetores a serem somados (A e B), o vetor destino (C) e o tamanho deles (N). Para calcular o índice da *thread* vamos utilizar a fórmula que se encontra na figura 1.7. Observe que o índice será o número do bloco vezes o tamanho do bloco mais o identificador da *thread* dentro do bloco.

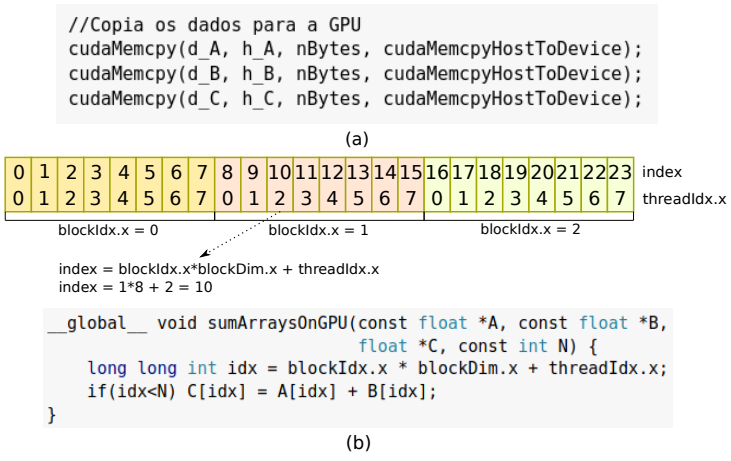

Figura 1.8. (a) Chamada da função cudaMemcpy(); (b) Exemplo soma de vetores.

Outro conceito introduzido nessa seção são os Warps. Dizemos previamente que as *threads* são organizadas dentro dos blocos, que têm tamanho que pode ser escolhido pelo programador. Além disso, dentro dos blocos elas são dispostas em grupos de 32 chamados Warps. Todas as *threads* em um mesmo warp realizam a mesma instrução porém cada uma sobre seus dados individuais. Vale ressaltar que o hardware sempre irá alocar um número inteiro de warps para um bloco e o warp nunca é dividido entre blocos. Por esse motivo, devemos atentar para o tamanho do bloco ao fazer a chamada do kernel, é interessante que esse tamanho seja múltiplo de 32 para evitar que existam threads inativas no final.

É importante ressaltar também que pelo motivo de todas as *threads* no warp executarem em conjunto, pode ser que algumas *threads* executem coisas que não servirão de nada, diminuindo a eficiência do código. Na seção 1.5.2, veremos o exemplo do código de redução que perde desempenho devido as divergências das *threads*. Por esse motivo é interessante sempre que possível evitar comandos condicionais e situações de divergência dentro de um warp. A figura 1.9 ilustra essa situação.

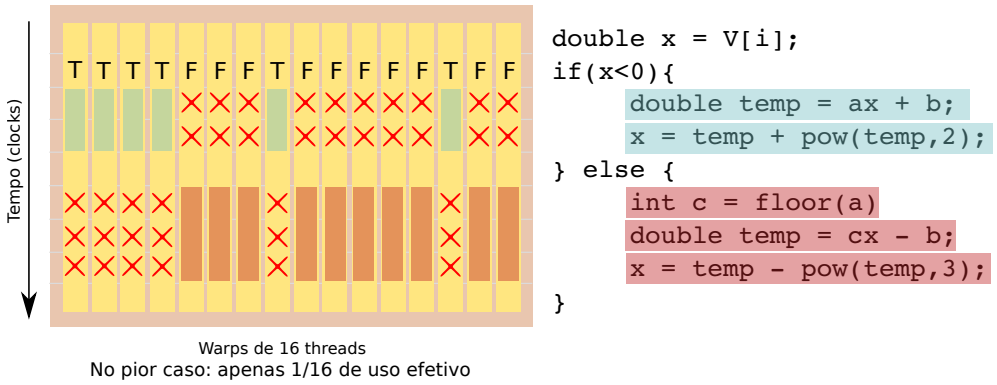

Figura 1.9. Divergência de threads em um warp devido ao uso de comando condicional.

1.3.3. Soma de Matrizes

Nessa última seção de introdução, vamos compilar os conceitos aprendidos e os aplicá-los ao problema da soma de matrizes quadradas. Até agora trabalhamos com o grid e o bloco como sendo unidimensionais, dessa vez vamos utilizar ambos com 2 dimensões. Vamos considerar o caso da soma de duas matrizes quadradas de tamanho N. O código para esse exemplo está disponível no laboratório 3.3.

O procedimento é parecido com o que utilizamos para a soma de vetores, mas agora vamos passar as dimensões na chamada do kernel de maneira diferente como indicado na figura 1.10(a).

A declaração do tamanho do grid, representado pela variável gridSize, e a do tamanho do bloco, representado pela variável blockSize recebem as dimensões N/2 e N/2. Note que agora precisamos de 2 índices para nos orientar pelo grid. Um para a dimensão x e outro para a dimensão y como mostrado na figura 1.10(b).

Para modificar os elementos do vetor no exemplo anterior utilizamos apenas o

```
dim3 blockSize( N/2,N/2 );
dim3 gridSize( N/2,N/2 );
additionMatricesKernel<<<gridSize,blockSize>>>(d_a, d_b, d_c);
```

(a)

```
int i = threadIdx.y+blockIdx.y*blockDim.y;
int j = threadIdx.x+blockIdx.x*blockDim.x;
d_c[i*N+j] = d_a[i*N+j] + d_b[i*N+j];
```

(b)

Figura 1.10. (a) Declaração das dimensões do grid e do bloco. (Ambos de tamanho N/2 x N/2) (b) Cálculo dos índices para o grid bidimensional

idx calculado diretamente. Porém, como agora temos uma matriz, precisamos utilizar o índice i*N+j, como mostrado na figura 1.11.

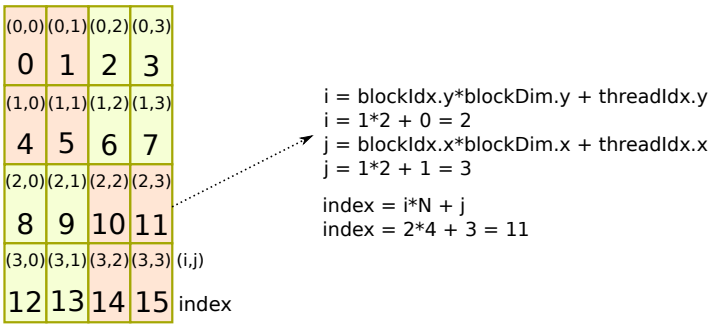

Figura 1.11. Cálculo do índice dos elementos da matriz

1.4. Instrumentação do Código

Nesta seção, nosso objetivo é instrumentar o código para analisar o seu desempenho nas GPUs [Mei and Chu 2016, Serpa et al. 2019, Jia et al. 2019, Arafa et al. 2019]. Esta seção está organizada da seguinte forma. Primeiro, a seção 1.4.1 utiliza as primitivas de eventos do *CUDA* para medir o tempo de execução com precisão. Depois, a seção 1.4.2 mostra como utilizar a ferramenta *nvprof* para coletar e apresentar várias métricas medidas durante a execução do seu código. Finalmente, a seção 1.4.4 ilustra como incluir medidores a nível de ciclo de relógio em trechos de código para coletar informações durante a execução. Os exemplos referentes a essa seção se encontram no laboratório 4.

1.4.1. Eventos

Primeiro iremos mostrar uma maneira simples e precisa de medir o tempo de execução de um *kernel* que irá executar na GPU. A GPU organiza as execuções em filas de execução chamadas de *streams*. Neste minicurso introdutório iremos trabalhar apenas com uma única fila de execução. Para sincronizar as filas e suas tarefas, a API *CUDA* oferece o objeto evento e algumas primitivas de controle de execução e espera. Podemos criar eventos como marcadores de início e fim de tarefas. Como iremos usar apenas uma única fila, os eventos servirão para delimitar o início e fim da execução de uma função ou *kernel*.

O procedimento padrão para medir o tempo de um ou mais chamadas que executam na GPU tem as seguintes fases. Primeiro, na função *time_start()* dois objetos *start* e

```
float elapsed_time;
cudaEvent_t start, stop;      // Declara dois eventos

void time_start(){            // Iniciar a contabilizar os
   eventos
  cudaEventCreate(&start);    // Irá marcar o inicio da execucao
  cudaEventCreate(&stop);     // Irá  marcar o final da execucao
  cudaEventRecord(start, 0);  // insere na fila
}

void time_end() {                      // Finaliza os eventos
  cudaEventRecord(stop, 0);            // insere na fila
  cudaEventSynchronize(stop);          // espera terminar
  cudaEventElapsedTime(&elapsed_time,
                       start, stop);   // calcula o tempo
}
```

Figura 1.12. Funções que calculam o tempo inicial e final da execução de um kernel

stop são declarados como ilustrado na figura 1.12. Depois, os eventos são inicializados com *cudaEventCreate*. Posteriormente, o evento *start* é inserido na fila de execução da GPU. Neste caso foi inserido na fila "0", que é a opção padrão. Na segunda fase, a função *time_end()*, a chamada irá inserir o *kernel* na fila de execução, onde inserimos logo em seguida o marcador *stop* para sincronizarmos quando a execução terminar com a chamada do *cudaEventSynchronize*. Finalmente, fazendo a diferença entre dos tempos dos eventos com a chamada do *cudeEventElapseTime* fornecerá o tempo em milisegundos.

Na figura 1.13 é apresentado como podemos encapsular as chamadas de eventos e simplificar a instrumentação do código. O laboratório *colab* apresenta detalhes desta implementação. É possível monitorar também o tempo de transferência dos dados entre as memórias da CPU e da GPU que são executados com os comandos *cudaMemcpy*.

```
....
time_start(); // inicializa a medição
Seu_kernel<<<grid, block>>>(....); // <- Seu KERNEL
time_end();   // finaliza a medição
printf("Tempo de execucao %.6fms\n",
                    elapsed_time );  // imprime o tempo
....
```

Figura 1.13. Exemplo de medição do tempo de execução via eventos

1.4.2. Ferramentas de Profile

A *Nvidia* oferece duas ferramentas para medir várias métricas de desempenho sem a necessidade de instrumentar seu código. As ferramentas são o *nvprof* e o *nsight*. Além da opção por linha de comando, ambas as ferramentas também possuem interface grá-

fica. Desde 2019, a *Nvidia* está migrando as novas versões da API CUDA para o *nsight*. Atualmente, o ambiente *Colab* oferece suporte apenas para o *nvprof*.

Para simplificar o uso do *nvprof*, a extensão proposta para este minicurso oferece o recurso de começar um código CUDA com *%%nvprof*. Todo o código desta célula será executado e monitorado com o *nvprof*. Na sua opção padrão, a saída do código irá imprimir as mensagem do seu código seguidas das informações de medidas de tempo. Os resultados são impressos em uma lista ordenada com as funções que você executou seguido das API CUDA como *cudaMalloc*, etc. Para cada função serão exibidas as seguintes informações: a percentagem do tempo total que a função utilizou, o tempo total, o número de vezes que a função foi chamada, seguido do tempo mínimo, médio e máximo e na última coluna o nome da função.

A figura 1.14 mostra um pequeno trecho da saída após a execução com *nvprof*. As três primeiras linhas mostrando as medidas para a chamada da função *cudaMemcpy* para copiar os dados da CPU para a GPU (*Host to device* ou *H2D*), depois para a função *cudaMemcpy* no sentido inverso, copiando da GPU para a CPU e finalmente o kernel *sum* que executa a soma dos vetores. Neste exemplo, o tempo de execução foi dominado pelo tempo de transferências de dados entre a GPU e a CPU.

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   67.30%  229.05ms         2  114.52ms  114.05ms  115.00ms  [CUDA memcpy HtoD]
                   30.82%  104.90ms         1  104.90ms  104.90ms  104.90ms  [CUDA memcpy DtoH]
                    1.87%  6.3711ms         1  6.3711ms  6.3711ms  6.3711ms  sum(float*, float*, float*, int)
      API calls:   63.53%  334.69ms         3  111.56ms  105.26ms  115.23ms  cudaMemcpy
                   34.11%  179.73ms         3  59.909ms  531.13us  178.65ms  cudaMalloc
                    1.21%  6.3980ms         1  6.3980ms  6.3980ms  6.3980ms  cudaEventSynchronize
```

Figura 1.14. Trecho impresso para execução do código de soma de vetores monitorado pelo *nvprof*.

1.4.3. Características das GPUs

As GPUs evoluíram aos longo dos anos com o lançamento de novas gerações. Algumas características são fundamentais para compreender o desempenho da GPU. Primeiro, precisamos saber qual é a quantidade de multiprocessadores e quais são os seus recursos. Segundo, precisamos saber qual é o desempenho no acesso ao sistema de memória. Terceiro, precisamos saber quais são os recursos dedicados que uma geração de GPU oferece e em quais situações podemos explorá-los.

Primeiro, a arquitetura de uma GPU é organizada em multiprocessadores ou *Stream Multiprocessors*. Estes multiprocessadores também são conhecidos pelo sigla *SM* ou *SMX*. Cada multiprocessador tem um conjunto de unidades de execução que são denominadas pelo termo *CUDA cores*. Os *Cores* são unidades de leitura/escrita em memória, unidades de execução para inteiros com 32 bits e unidades de execução para ponto flutuante que podem ser para 32 ou 64 bits. As gerações mais novas, a partir da GPU Volta (Turing e Ampere), incluem também unidades para processamento de tensores que são multiplicadores sistólicos de matrizes e recursos para manipular números com 16,8,4 e até 1 bit que são comuns em aplicações e modelos de aprendizado de máquina e redes neurais.

O modelo lógico de threads e blocos que vimos na seção 1.3 deixa o código da GPU independente da arquitetura alvo. Se a arquitetura tem mais recursos, como mais

multiprocessadores, irá executar mais rápido. A figura 1.15 ilustra um exemplo com 8 blocos mapeados em duas arquiteturas com 4 e 8 multiprocessadores.

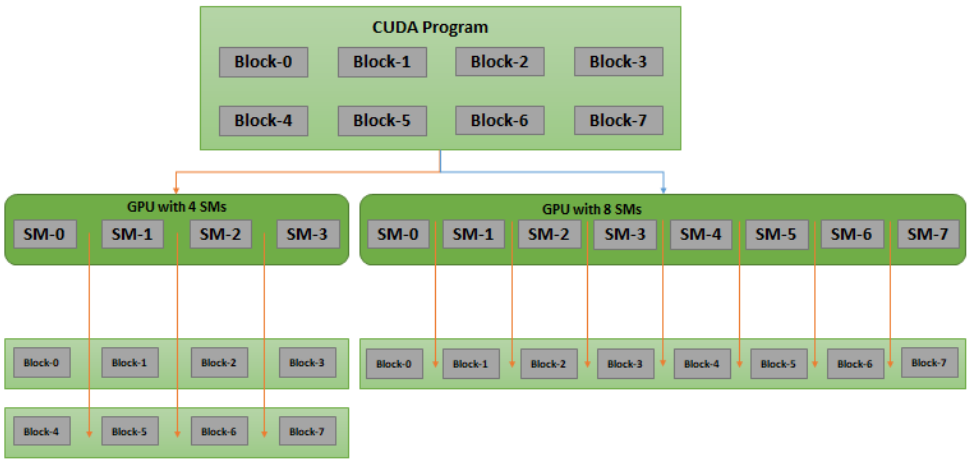

Figura 1.15. Mapeamento de um bloco de threads em duas arquiteturas com 4 e 8 multiprocessadores, respectivamente. Figura extraída do Manual CUDA programming Guide da Nvidia

O desempenho de pico de uma GPU pode ser calculado multiplicando o número de multiprocessadores pela quantidade de unidades do multiprocessador vezes a frequência de relógio. Por exemplo, a K80 tem 13 multiprocessadores com 192 unidades ou *CUDA Cores*, totalizando 2496 CUDA Cores executando a 824 MHz, onde duas operações são realizadas por ciclo. Portanto, o seu desempenho de pico será 2 * 2496 * 0,824 Gops/s = 4 Tera operações por segundo.

Tabela 1.1. Mutliprocessadores nas quatro GPUs disponíveis no Google Colab

	K80	P4	P100	T4
Número de Multiprocessadores	13	20	56	40
Cores por Multiprocessador	192	128	64	64
Total de CUDA Cores	2496	2560	3584	2560
Frequência de Relógio (MHz)	824	1114	1480	1590
Desempenho de Pico (TFLOPS)	4.1	5.7	10.6	8.1

Outro ponto importante é que uma aplicação terá que executar muitos *threads* simultaneamente para obter o desempenho de pico ou maximizar o seu desempenho. Na GPU K80 serão necessários no mínimo 2496 *threads* em um cenário ideal sem atrasos nos acesso aos dados e operações sem dependências entre elas. Como esta situação raramente ocorre, em caso de dependências entre as instruções, supondo uma latência de 5 à 10 ciclos para ter os dados necessários para executar uma nova operação, teremos que ter pelo menos 5 à $10 \times 2496 = 12480$ à 24960 *threads* executando paralelamente. Portanto, a GPU passa a ser vantajosa em cenários com muitos *threads* paralelos, senão

será sub-utilizada. Para explorar este potencial ao máximo, temos que entender bem o seu funcionamento.

O segundo ponto é o sistema de memória da GPU. Uma grande inovação da GPU é a memória com grande largura de banda. As primeiras GPUs já eram capazes de atingir 150 GB/s de taxa de leitura. Atualmente, as últimas gerações já atingem 1 TB/s. A partir da arquitetura Pascal, com a inclusão de memória 3D, a taxa de transferência pulou de 300 GB/s para 700 GB/s. Entretanto, o acesso deve ser aglutinado, onde as *threads* do mesmo *warp* deve fazer acesso ao mesmo bloco de memória e muitos acessos devem ser disparados para esconder a latência. A figura 1.16(a) mostra dois exemplos onde as *threads* 0 à 15 fazem um acesso contíguo ou aglutinadona memória que é um bom padrão de acesso. Na figura 1.16(b), apenas os 4 primeiras *threads* já geram um padrão ruim que requer o movimento de 4 blocos inteiros para acessar apenas um elemento em cada bloco. Este seria o padrão de acesso ao percorrer uma coluna em uma matriz. Como veremos adiante, não é um bom padrão e deve ser otimizado para obter desempenho da GPU.

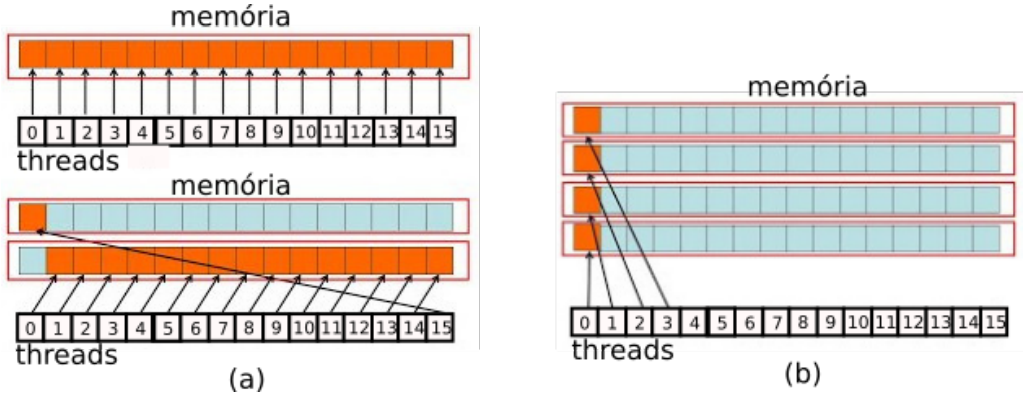

Figura 1.16. (a) Acesso aglutinado (*coalesced*) na memória global; (b) as *threads* 0,1, 2 e 3 acessam blocos distintos de memória.

Outra inovação da GPU foi popularizar as memórias *scratpad* ou de rascunho. Estas memórias já eram usadas desde dos anos 70, introduzidas no processador *Fairchild F8* de 1974. Entretanto, a GPU simplificou o seu uso e adicionou funcionalidades interessantes para o contexto de alto desempenho. As primeiras GPUs não tinham *cache* mas tinham um módulo de memória compartilhada, a *shared memory*, dentro de cada multiprocessador. A *shared* é mais rápida que a memória global e possui um acesso por bancos. De forma simplificada, a *shared* é uma *cache* controlada em software. Além disso, a *shared* pode ser usada para as *threads* do mesmo bloco se comunicarem e executarem um trabalho conjunto. As caches L1 foram introduzidas a partir da geração Fermi e a partir da geração Kepler, recursos para controlar o tamanho da cache e da *shared* foi incluídos. As novas gerações incluem também cache L2.

A figura 1.17(a) mostra as 32 threads de um warp fazendo acesso paralelo sem conflitos, onde cada *thread* faz acesso a um banco diferente da memória compartilhada. Caso haja conflito, onde 2 ou mais threads fazem acesso ao mesmo banco, irá ocorrer uma perda de desempenho e o acesso será serializado. Mas a GPU resolve isto em hardware. Caso seja necessário evitar condições de corrida no acesso aos dados, a GPU oferece o recurso de operações atômicas na memória global ou compartilhada. A figura 1.17(b) mos-

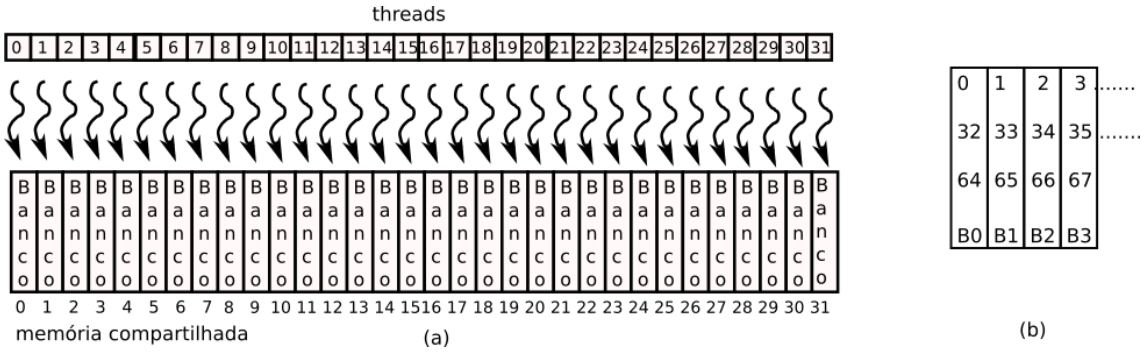

Figura 1.17. (a) Acesso por bancos na memória compartilhada sem conflitos; (b) Os dados são organizados nos bancos de forma: banco B_0 armazena as posições $0, 32, 64, \ldots, j*32$, ou seja o banco B_i armazena $i, i+32, i+64, \ldots, j*32+i$.

tra como os dados dos vetores armazenados na memória compartilhada são distribuídos nos bancos. O banco B_0 armazena as posições $0, 32, 64, \ldots, j*32$, o banco B_1 armazena as posições $1, 33, 65, \ldots, 1+j*32$, ou seja, o banco B_i armazena $i, i+32, i+64, \ldots, j*32+i$.

Outro grande diferencial entre a CPU e a GPU é a quantidade de registradores. A GPU faz uso massivo de registradores para esconder a latência da memória principal, denominada memória global. O termo global é devido ao fato de ser acessível a todas as *threads*. Já a memória compartilhada só é acessível entre as *threads* do mesmo bloco e só permanece "viva"enquanto o bloco da *thread* está em execução. Já os registradores são locais sendo acessíveis apenas a *thread*. A instrução *shuffle*, introduzida na geração Kepler, permite troca de valores dos registradores dentro do mesmo *warp*. O desempenho na GPU em geral é graças ao uso dos registradores que conseguem fornecer rápido os dados para alguns milhares de *CUDA cores*.

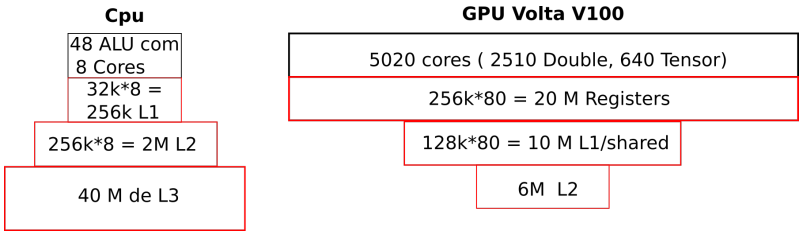

Figura 1.18. Hierarquia de memória na CPU multicore e na GPU Volta V100

A figura 1.18 ilustra o espaço total ocupado na hierarquia de memória em um CPU com 8 núcleos em comparação com uma GPU Volta V100. Podemos observar uma pirâmide invertida na GPU. O maior espaço é ocupado pelos registradores, seguido das caches L1 que são 80 unidades de 128KB cada, uma por multiprocessador. No último nível temos a cache L2 que é compartilhada por todos os multiprocessadores com capacidade de 6 MB. Apesar da tendência de uma pirâmide invertida nas GPUs, a nova GPU A100 mudou um pouco o cenário. A A100 inclui uma cache L2 de 40 MB que possui uma conexão direta com a memória compartilhada para alimentar as unidades de tensores. Esta operação é fundamental no treinamento e inferência das redes neurais e algoritmos de aprendizado de máquina. A figura 1.19 mostra resumidamente as características da arquitetura de memória da GPU A100 que tem 108 multiprocessadores.

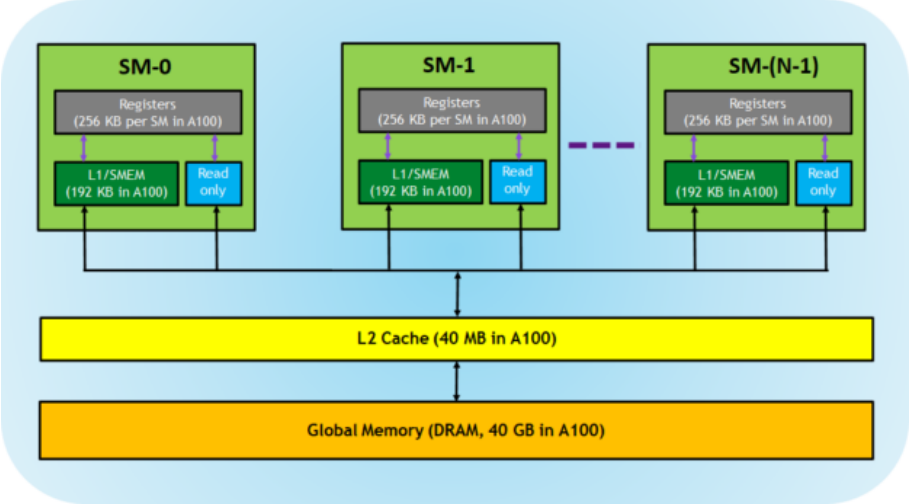

Figura 1.19. Hierarquia de memória na GPU Ampere A100. Figura extraída CUDA Refresher: The CUDA Programming Model

Tabela 1.2. Recursos de memória por Mutliprocessadores e totais nas quatro GPUs disponíveis no Google Colab. A primeira linha contêm links para os relatórios técnicos (*White Paper*) com mais detalhes das GPUs.

	K80	P4	P100	T4
Número de Multiprocessadores	13	20	56	40
Cores por Multiprocessador	192	128	64	64
Cache L_1 por Multiprocessador (KB)	16	48	24	48
Shared por Multiprocessador (KB)	48	96	64	64
Registradores por Multiprocessador (KB)	256	256	256	256
Cache L_1 por CUDA Core (Bytes)	85	384	384	1024
Shared por CUDA Core (Bytes)	256	768	768	1024
Registradores por CUDA Core	341	512	1024	1024
Total Cache L_1 (KB)	208	960	1344	1920
Total de Shared (KB)	624	1920	3584	2560
Total de Registradores (MB)	3.25	5	14	10
Cache L_2 (MB)	1.5	2.0	4.0	4.0
Vazão da memória principal (GB/s)	240.6	192.2	732.2	320.1

A tabela 1.2 mostra de forma resumida a quantidade de recursos disponíveis por multiprocessador e a quantidade total. Importante ter em mente que os recursos são compartilhados, por isso mostramos também a quantidade média de recursos para cada CUDA core dentro do multiprocessador. A penúltima e última linhas mostram a quantidade de cache L_2 e o desempenho da GPU para ler na memória global medido com o código *bandwith_test* dos exemplos disponibilizados pela Nvidia.

1.4.4. Ciclos de Relógio

Como já mencionado, a memória da GPU tem uma alta vazão, porém, tem uma grande latência de 400 à 800 ciclos. Entretanto, como é possível disparar milhares de threads concorrentemente, enquanto várias *threads* estão esperando suas requisições de leitura de dados, outros estão sendo atendidos. Além dos eventos e dos recursos do nvprof, a GPU permite instrumentar o código e medir quantos ciclos foram gastos em um determinado trecho ou até mesmo qual foi a latência de uma instrução.

Nesta seção iremos mostrar dois kernels e como podemos medir a latência de uma instrução em ciclos de relógio. A figura 1.20 mostra dois trechos de código instrumentados para medir a latência das instruções *div.f32* e *add.f32* que executam a divisão e a adição de números de ponto flutuante de 32 bits. Os dois kernels recebem três vetores como parâmetros: A, B e C. Além disso, tem uma variável local X para o cálculo temporário, uma variável Idx para índice e duas variáveis inteiras c1 e c2 para medir o número de ciclos.

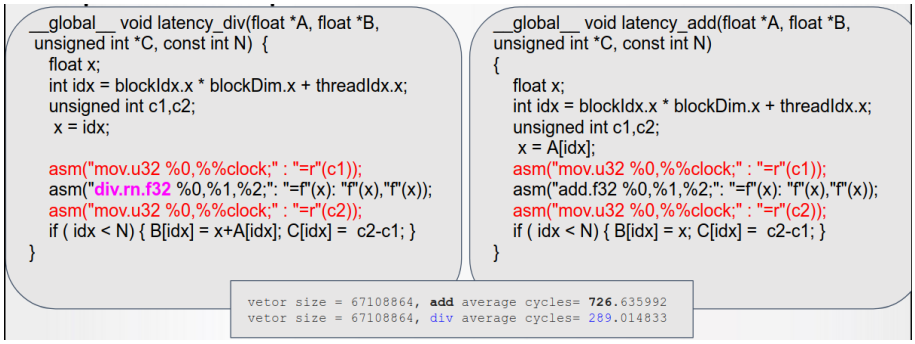

Figura 1.20. A esquerda um código para medir a latência da instrução DIV e a direita um código para medir a latência da instrução ADD.

Primeiro, cada *thread* irá calcular seu identificador único usando seu número de bloco e seu número dentro do bloco e armazenar em Idx. Em seguida, surge a primeira diferença entre os dois códigos. Enquanto o kernel da divisão inicializa X com o identificador Idx da *thread*, o kernel da adição inicializa X com o valor do elemento do vetor A na posição Idx.

Para medir o número de ciclos, ambos os kernels usam a mesma estratégia. A primeira instrução em assembler

```
asm("mov.u32 %0,%%clock;" : "=r"(c1));
```

irá ler o número atual de ciclos de relógio (clock) e armazenar na variável inteira c1. Depois, cada kernel executa sua instrução div ou add. Em seguida, a variável c2 registra o número de ciclos após a execução do div ou add. Finalmente, a diferença de c2 e c1 irá medir a latência em número de ciclos do div e do add, respectivamente. Este valor é registrado no vetor C e retornado. Portanto, o programa principal pode calcular qual é a latência média tendo acesso detalhado a latência medida de cada *thread*. Em baixo na figura 1.20, temos que foram executados 2^{26} threads que são 67108864 threads. Podemos observar que em média adição gastou 726,6 ciclos e a divisão 289,0 ciclos.

A primeira questão surge então. Por que a instrução de adição foi mais lenta se o processo de medida do ciclos foi o mesmo ? A resposta está na instrução anterior ao add que disparou uma leitura em memória $X = A[idx]$. Esta instrução não trava a instrução *mov* que faz a leitura inicial dos ciclos e armazena em c1. Porém, a instrução *add* precisa do valor do X e irá aguardá-lo. Portanto, c1 irá contabilizar também a latência da leitura de $A[idx]$ da memória. Por isso, o valor médio de 726,6 ciclos da adição inclui a leitura na memória. Se modificamos este exemplo para $X = idx$, semelhante ao código da divisão, a latência média da adição será reduzida para 37,4 ciclos.

Este pequeno exemplo mostra que é possível fazer análises detalhadas da latência da GPU para implementações com otimizações avançadas. Sugerimos ao leitor o minicurso do WSCAD de 2019 [Ferreira et al. 2019] e trabalhos que fazem uma análise das latências da GPU [Arafa et al. 2019, Jia et al. 2019].

1.5. Galeria de Exemplos

Esta seção apresenta uma galeria de exemplos de códigos para ilustrar técnicas de programação em GPU como o uso de memória compartilhada, das operações atômicas dentre outras. Serão quatro exemplos: a multiplicação de matrizes, redução, convolução e histogramas. Além disso, vários *links* para outros exemplos estarão disponíveis para atividades extras. Nosso objetivo é mostrar o impacto das otimizações em vários exemplos. É possível explorar o impacto também nas diferentes GPUs disponíveis no Colab.

Esta seção está organizada da seguinte forma. A seção 1.5.1 apresenta duas versões de multiplicação de matrizes, onde uma versão é ingênua e outra faz uso de ladrilhos e memória compartilhada. A seção 1.5.2 apresenta três versões de uma redução de soma, ilustrando também o uso da memória compartilhada. A seção 1.5.3 ilustra dois códigos de convolução com uma dimensão, que também faz uso da memória compartilhada. Finalmente, a seção 1.5.4 apresenta um exemplo de histograma com operações atômicas em memória global e compartilhada. Os exemplos para as seções 1.5.1 e 1.5.2 estão no laboratório 5.1 enquanto os exemplos para as seções seguintes estão no laboratório 5.2.

1.5.1. Multiplicação de Matrizes

Esta seção apresenta duas implementações de multiplicação de matrizes. A primeira versão é um código ingênuo ou *naive* que percorre a matriz A por linha e a matriz B por coluna fazendo a multiplicação. Supondo duas matrizes A e B de tamanho $N \times N$, para cada elemento da matriz resultante teremos N multiplicações e $N-1$ adições. Cada *thread* irá calcular um ponto da matriz resultante. Portanto, cada *thread* fará $2N-1$ operações, N leituras de memória na mesma linha e N leituras de memória na mesma coluna de B e apenas uma escrita na memória. As GPUs do colab possuem cache L1 e L2, partes dos dados ficarão temporariamente nas caches, melhorando o desempenho. Porém, o padrão de acesso ao percorrer uma coluna na matriz B é ineficiente. O resultado é um desempenho bem abaixo do valor de pico de uma GPU. A segunda versão faz uso da memória compartilhada para armazenar uma sub-matriz ou um *ladrilho* da matriz. Além disso, otimiza o acesso para evitar conflitos de bancos na leitura da matriz B onde uma coluna deve ser percorrida. Resumindo, a versão otimizada maximiza o reuso dos dados, assim para um total de $O(N^3)$ operações, apenas $O(N^2)$ leituras/escritas são realizadas na memória,

resultando em um reuso de $O(N)$ operações de cálculo para cada operação de memória.

A tabela 1.3 apresenta os tempos de execução para as 4 GPU disponíveis no Colab considerando as duas implementações. As matrizes A e B tem 1024×1024 elementos, ocupando 4 Mega Bytes, pois cada elemento ocupa 4 bytes. Os blocos são bi-dimensionais e tem 32x32 threads, o grid também é bi-dimensional com 32x32 blocos, totalizando assim $32 \times 32 \times 32 \times 32 = 1$ Mega *thread*s, que equivale à 1048576 threads.

Tabela 1.3. Tempo de execução em ms nas quatro GPU disponíveis no Google Colab para as duas implementações da multiplicação de matrizes. A versão ingênua e a com ladrilhos foram extraídas do repositório *GPGPU Programming with CUDA* [Nick 2020].

Implementação	K80 (ms)	P4 (ms)	P100 (ms)	T4 (ms)
Ingênua	36.51	15.08	6.77	15.07
Ladrilhos e Memória Compartilhada	13.38	7.25	1.94	7.25

Podemos observar uma diferença de 3 vezes no tempo de execução. A versão com ladrilhos ainda pode ser otimizada. Recomendamos aos leitores interessados, as otimizações propostas por Volkov [Volkov 2010]. Outra sugestão de atividade complementar é explorar as outras três implementações disponíveis repositório *GPGPU Programming with CUDA* [Nick 2020].

A GPU T4 tem os operadores tensores. Um outro experimento é executar o exemplo *cudaTensorCoreGemm* da Nvidia [Nvidia 2020] que utiliza os tensores e tem um desempenho de 9 TFlop/s em comparação com o código padrão dos exemplos da Nvidia, o *matmul*, que inclui ladrilhos e tem um desempenho de 1,64 TFlop/s em um GPU P100. Na versão ingênua da tabela 1.3, o desempenho foi de 317 GFlop/s e na versão com ladrilhos foi de 1,1 TFlop/s.

1.5.2. Redução

Uma operação importante em computação paralela é a redução. Nesta seção iremos mostrar três implementações da redução de soma.

A primeira versão é mais simples, apenas percorre um vetor e vai somando e reduzindo. Primeiro, cada bloco fica responsável ao equivalente a um bloco de dados do vetor. Apenas as *threads* pares irão trabalhar no primeiro passo. Ou seja, ocorre uma subutilização das threads. Dentro do bloco, eles começam somando dois elementos adjacentes. Por exemplo, a *thread* 0 irá somar os elementos das posições 0 e 1, enquanto a *thread* 2 irá somar os elementos das posições 2 e 3. No próximo passo, cada *thread* i múltiplo de 4 irá acumular a soma das *threads* i e i+2. Por exemplo, a *thread* 0 irá acumular sua soma que já tem os valores das posições 0 e 1 e somar ao acumulado pela thread 2 que somou as posições 3 e 4. A figura 1.21(a) mostra a ideia da implementação da redução. A primeira versão implementada é simples e não grava em memória compartilhada. Como grava na memória global, gera muitas operações de alto custo.

A segunda versão organiza as *threads* de forma diferente. Além disso, cada *thread* faz oito leituras na memória global. Para um vetor de tamanho N serão disparados

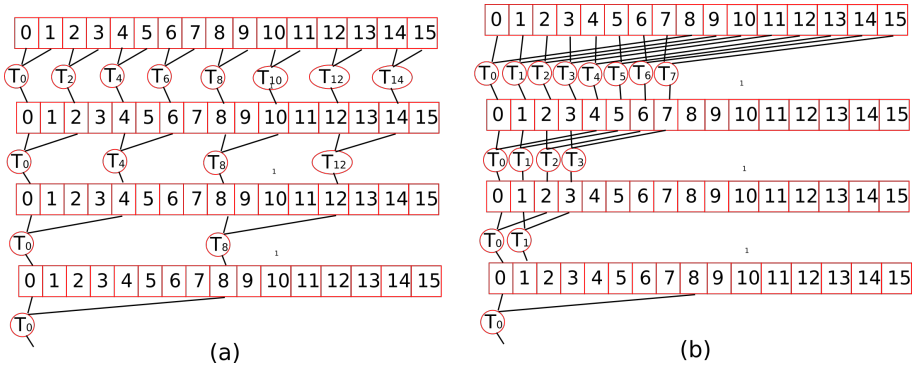

Figura 1.21. (a) Redução com threads subutilizados nos warps; (b) Redução com threads agrupados no mesmo warp.

N/(8*B) blocos, pois cada bloco de threads irá processar 8 blocos de dados do vetor. Como são partes independentes, o bloco dispara leituras independentes melhorando a vazão no acesso aos dados iniciais. Estes dados são gravados temporariamente em registros da *thread* e somados localmente. Depois são gravados na memória global e todas as *threads* do bloco são sincronizados. Para evitar threads ociosos dentro do warp, a estratégia de redução segue o padrão da figura 1.21(b).

Para simplificar a explicação, suponha um bloco de 16 elementos, a *thread* 0 irá somar o elemento 0 e elemento da metade mais 1, ou seja, o elemento 8. Depois a *thread* 0 irá somar o acumulado mais a soma acumulada da posição 4. Esta implementação foi extraída do livro [Cheng et al. 2014].

A terceira versão usa a memória compartilhada para realizar a redução dos elementos dentro do bloco. Primeiro, cada *thread* do bloco copia um elemento da global para memoria compartilhada do bloco. A redução é realizada dentro do bloco, evitando divergências dentro do warp e finalmente apenas a *thread* 0 de cada bloco escreve o resultado na memória global para finalizar a redução com o restantes dos blocos.

A tabela 1.4 apresenta os tempos de execução para as 4 GPUs disponíveis no Colab considerando as três implementações. O vetor avaliado tem 2^{24} elementos, ocupando 67108864 Bytes e foram disparados 16777216 threads, 512 thread por blocos.

Tabela 1.4. Tempo de execução nas quatro GPUs disponíveis no Google Colab para as três implementações da redução de soma. A versão simples e a com *unroll* fator 8 foram extraídas do livro [Cheng et al. 2014]. A versão em memória compartilhada foi extraída do repositório *GPGPU Programming with CUDA* [Nick 2020].

Implementação	K80 (ms)	P4 (ms)	P100 (ms)	T4 (ms)
Simples	7.86	3.36	2.02	3.35
Unroll Fator 8	0.87	0.32	0.22	0.31
Memória Compartilhada	3.27	1.91	0.81	1.92

A tabela 1.4 mostra que existe uma diferença significativa no tempo e mais opções ainda podem ser exploradas. A segunda versão usa a estratégia de mais trabalho para cada

thread [Volkov 2010] que resulta em um melhor desempenho.

Como atividades complementares, nossa sugestão é explorar as outras cinco implementações do livro [Cheng et al. 2014] disponíveis no repositório [John Cheng 2016] e as outras cinco implementações de redução disponíveis repositório *GPGPU Programming with CUDA* [Nick 2020].

1.5.3. Convolução

Como mencionado anteriormente, os exemplos dessa seção e da próxima seção se encontram no laboratório 5.2. Esta seção apresenta dois códigos para a operação de convolução em um vetor. O primeiro código é uma versão ingênua onde todas as operações são realizadas na memória. Como existe reuso dos dados, as caches ajudam no desempenho de forma transparente ao programador. A segunda versão armazena a máscara de convolução na memória de constantes e faz uma cópia dos trechos do vetor para memória compartilhada. Os códigos foram extraídos do repositório *GPGPU Programming with CUDA* [Nick 2020].

Este exemplo ilustra o uso da memória compartilhada com reuso dos dados que são lidos uma única vez da memória global. As *threads* irão percorrer a máscara de forma sincronizada. Quando maior a máscara, maior será o reuso e o desempenho por elemento da GPU.

A figura 1.22 apresenta um exemplo de convolução com uma máscara *a* de três elementos aplicada a um vetor *m*, gerando os resultados do somatório dos três elementos no vetor n. O exemplo mostra apenas o cálculo do quarto elemento do vetor com o valor inicial igual a 7. Na convolução ou *stencil-1D*, a operação será aplicada a todos os elementos do vetor.

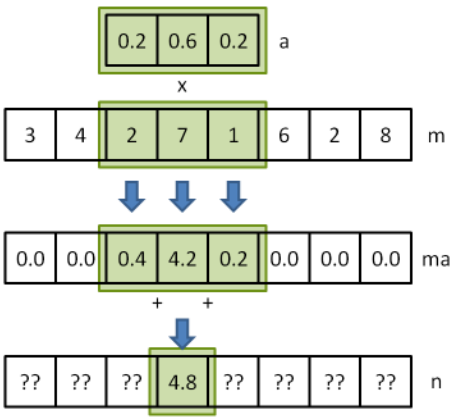

Figura 1.22. Exemplo de convolução 1D com uma máscara de 3 elementos. Figura extraída [Cole et al. 2011]

A tabela 1.5 apresenta os tempos de execução para as 4 GPUs disponíveis no Colab considerando as duas implementações. O vetor avaliado tem 2^{24} elementos, ocupando 67108864 Bytes, a máscara tem 7 elementos e foram disparados 16777216 threads, 256 threads por bloco.

Como atividades complementares, nossa sugestão é explorar as outras três imple-

Tabela 1.5. Tempo de execução nas quatro GPUs disponíveis no Google Colab para duas implementações da Convolução 1D extraídas do repositório *GPGPU Programming with CUDA* [Nick 2020].

Implementação	K80 (ms)	P4 (ms)	P100 (ms)	T4 (ms)
Ingênua	36.51	15.08	6.77	0.94
Memória Compartilhada	13.38	7.25	1.94	0.94

mentações de convolução, incluindo uma versão bi-dimensional disponíveis no repositório *GPGPU Programming with CUDA* [Nick 2020]. Outro exercício é variar o número de *threads*, tamanho do vetor e tamanho da máscara.

1.5.4. Histograma

Nesta seção iremos calcular um histograma sobre um longo vetor. Duas implementações serão comparadas. A primeira utiliza a operação atômica em memória global para acrescentar um elemento na sua célula do histograma. A segunda versão calcula localmente o histograma na memória compartilhada e também faz uso da operação atômica só que na memória compartilhada.

Para explorar o paralelismo no cálculo do histograma, cada *thread* irá classificar um ou mais elementos do vetor concorrentemente. O exemplo desta seção irá fazer um histograma das ocorrências das letras do alfabeto no vetor *a*. As letras serão classificadas em 7 categorias, as ocorrências das letras a,b,c,d,e,f e g serão contabilizadas nas classes 0,1,...,6, respectivamente. As ocorrências de h serão contabilizadas juntas com a letra a, a letra i com a letra b e assim sucessivamente de 7 em 7 letras.

Cada *thread* lê um elemento, calcula sua classe e incrementa o número de ocorrências da classe usando a operação atômica da GPU para memória global, evitando assim conflitos nos acessos paralelos. As GPUs oferecem recursos em hardware para otimizar esta operação.

A segunda versão usa a memória compartilhada. Supondo blocos com 512 threads. Primeiro, uma cópia local do histograma é criada. Suponha 7 classes. Apenas as sete primeiras *threads* irão inicializar a contagem com 0. Depois todas as *threads* irão cooperar para atualizar as contagens em paralelo usando a operação atômica na memória compartilhada para evitar condições de corrida. Finalmente, apenas os sete primeiras *threads* irão atualizar o resultado na memória global.

A tabela 1.6 apresenta os tempos de execução para as 4 GPUs disponíveis no Colab considerando as duas implementações. O vetor avaliado tem 2^{24} elementos, ocupando 16777216 Bytes, o histograma tem 7 classes e foram disparados 16777216 threads no total, 32 thread por elemento.

Podemos observar que a GPU K80 não teve melhoria de desempenho com o uso da memória compartilhada. Isto ocorre devido ao fato da implementação da operação atômica na memória compartilhada não ser eficiente na K80.

Tabela 1.6. Tempo de execução nas quatro GPUs disponíveis no Google Colab para duas implementações da Convolução 1D extraídas do repositório *GPGPU Programming with CUDA* [Nick 2020].

Implementação	K80 (ms)	P4 (ms)	P100 (ms)	T4 (ms)
Atômico em Memória Global	8.93	8.91	8.95	8.74
Atômico em Memória Compartilhada	7.05	1.87	2.31	0.74

1.5.5. Outros Exemplos

Como sugestão de mais exemplos, sugerimos o minicurso "Métricas e Números: Desmistificando a Programação de Alto Desempenho em GPU" [Ferreira et al. 2019] cujo os códigos para o Colab estão disponíveis no repositório [Ferreira and Canesche 2020]. Podemos destacar exemplos de transposição de matrizes, avaliação de polinômios, biblioteca Trust, algoritmo TEA de criptografia, além de sugestões de várias atividades. Indicamos também outros repositórios. O repositório GPU_Programming tem exemplos de propriedades da GPU, histogramas de equalização e convolução de imagens, redução, multiplicação de matrizes, soma de vetores e de prefixos. O repositório do livro Learn CUDA Programming [Jaegeun Han 2019] inclui exemplos introdutórios, acesso as memórias, modelo de programação com milhares de threads, além de exemplos de aprendizado profundo e *openacc*. Finalmente, o repositório micro-benchmarks contém alguns exemplos de como medir propriedades das arquiteturas como em qual multiprocessador um determinado bloco executou.

1.6. Experimentos com GPU

Esta seção apresenta experimentos com os códigos da galeria de exemplos da seção 1.5. Nos experimentos, os códigos são executados várias vezes com parâmetros diferentes e os resultados armazenados em um arquivo no formato *CSV*. Como ilustrado na seção 1.2.3, podemos usar as facilidades das bibliotecas do Python como a *Panda* e *Matplotlib* para gerar gráficos, facilitando a visualização da avaliação de desempenho. Os exemplos para as seções 1.6.1 e 1.6.2 estão no laboratório 6.1 enquanto os exemplos para as seções seguintes estão no laboratório 6.2.

1.6.1. Multiplicação de Matrizes

Nesta seção exploramos o exemplo de multiplicação de matrizes da seção 1.5.1. Modificamos o código principal para variar o tamanho das matrizes e armazenar os tempos de execução da versão ingênua (*naive*) e da versão com ladrilhos e memória compartilhada (*tiled*) em um arquivo .CSV. Depois, o laboratório ilustra com um código em Python como ler o arquivo e gerar automaticamente gráficos de barras com e sem escala logarítmica

A figura 1.23(a) mostra os gráficos resultantes que foram executados em uma GPU P4 usando blocos com 32 por 32 threads. Na figura 1.23(b) é apresentado o mesmo gráfico, contudo na escala logarítmica na base 10.

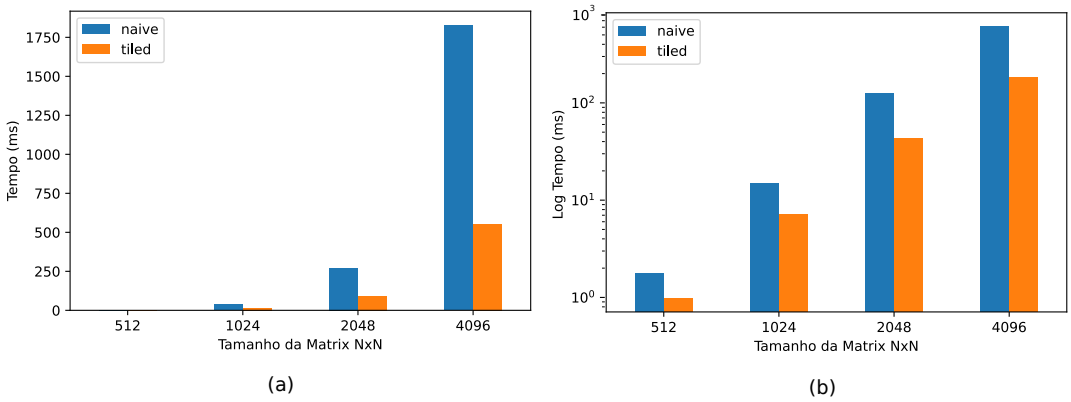

Figura 1.23. Tempo de execução para Multiplicação de Matrizes de vários tamanhos de dimensão NxN, começa com 512x512 até 4096x4906: (a) Tempo em escala linear; (b) Tempo em escala logarítmica.

1.6.2. Redução

Nesta seção iremos fazer um experimento para variar o tamanho do bloco, ou seja, o número de threads por bloco e avaliar o impacto no tempo de execução das três versões de redução apresentadas na seção 1.5.2. O código do programa principal foi alterado para receber uma lista de tamanho do bloco e automaticamente gerar o arquivo CSV com os tempos das três versões. Posteriormente, com um código Python, os resultados foram processados e apresentados na forma de gráfico com linhas, uma para cada versão.

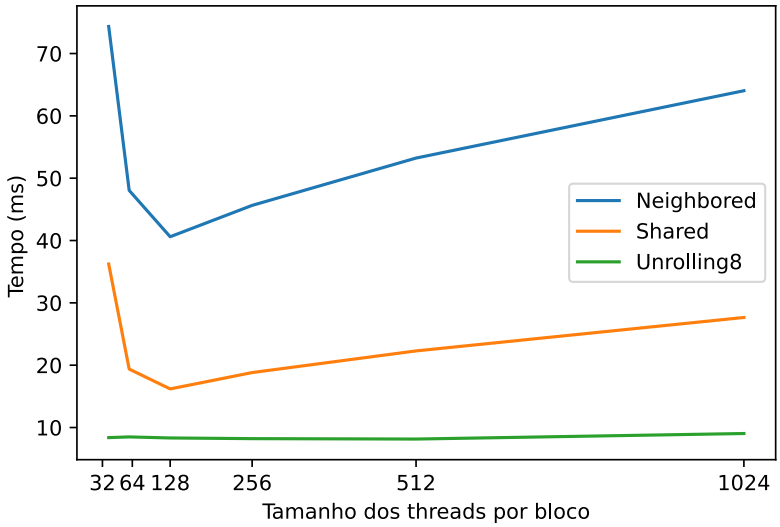

Figura 1.24. Tempo de execução em função do tamanho do bloco para três versões de redução apresentadas na seção 1.5.2. Os testes foram executados em uma GPU Nvidia P4

A figura 1.24 mostra os resultados com o tamanho do bloco variando de 32 a 1024 threads. Para a segunda versão que usa a técnica de 8 elementos por thread, armazena

resultados temporários em registradores e faz uso da otimização de *unroll* com o fator 8, não teve diferenças significativas. Porém para a primeira e a terceira versão, podemos ver claramente que o bloco com 128 threads produz o melhor resultado. Estes fatos podem ser explorados para buscar explicações e compreender o funcionamento das GPUs.

1.6.3. Convolução

Esta seção irá variar o tamanho da máscara de convolução e medir o tempo de execução para as duas implementações de convolução apresentadas na seção 1.5.3. Semelhante as duas seções anteriores, o programa principal foi modificado para receber uma lista de tamanhos para as máscaras, o resultado é gravado em um arquivo CSV que é posteriormente processado por um código Python. Neste exemplo ilustramos um gráfico com barras no sentido horizontal.

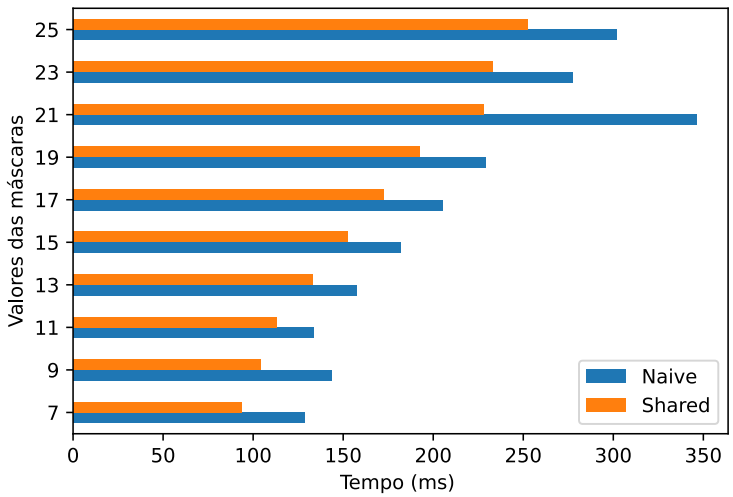

Figura 1.25. Tempo de execução em função do tamanho da máscara de convolução para as duas implementações da seção 1.5.3 executadas em uma GPU Nvidia K80

A figura 1.25 mostra o tempo de execução com o tamanho da máscara variando de 7 à 25 na implementação ingênua (*naive*) em memória global e a implementação em memória compartilhada (*shared*).

1.6.4. Histograma

Esta seção mostra dois exemplos de execução para as duas implementações de histogramas apresentadas na seção 1.5.4. O primeiro laboratório imprime o resultado de execução que é um histograma usando gráfico com barras. O histograma do exemplo tem 13 classes e ambas as implementações geram o mesmo resultado, como esperado. O segundo laboratório irá variar o número de classes do histograma e medir o tempo de execução para a implementação usando atômico na memória global e a versão com atômico na memória compartilhada (*shared*). A figura 1.26 apresenta o resultado dos dois experimentos.

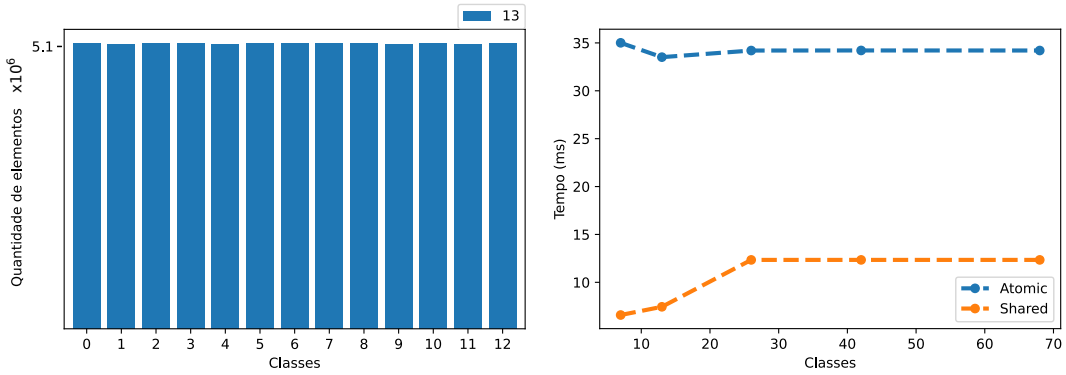

Figura 1.26. (a) Histograma com 13 classes; (b) Tempo de execução em funão do número de classes para as implementações de histograma da seção 1.5.4 executados em uma GPU Nvidia P100

1.7. Conclusão

Este minicurso apresentou uma sequência de atividades de laboratório no ambiente *Google Colaboratory* ou simplesmente *Colab* para o ensino de programação em GPU. Vários exemplos foram elaborados ou adaptados de repositórios de códigos com exemplos de implementações em GPU. Para explorar o potencial do uso do *colab* como ferramenta de ensino ilustramos vários recursos desde de a execução em várias linguagens, instalação e configuração do ambiente, integração e automatização de *scripts*. O tema central do minicurso é o ensino de GPU. Mostramos as opções e recursos das quatro GPUs atualmente disponíveis no *colab*, além de ilustrar como podemos medir o desempenho com base no tempo de execução e em outras métricas. Por fim, o material está disponível para a comunidade[3], pode ser usado de forma colaborativa e será expandido e atualizado.

Como trabalhos futuros sugerimos o desenvolvimento de mais exemplos e formas mais interativas com o uso de *widgets* (botões, caixas de texto, etc..) e a integração com exemplos de aprendizado de máquina. Como já mencionado no texto, sugerimos as referências a seguir para realizar vários experimentos em GPU: a referência [Cheng et al. 2014] cujo os códigos estão disponíveis [4], o minicurso da SBC Técnicas de Otimização de Código para Placasde Processamento Gráfico do Prof. Fernando M.Q.Pereira [Pereira 2011], o repositório CUDA by practice de Edgar Garcia Cano que contém vários exemplos de nível introdutório que podem ser executados no *Colab*, o tutorial disponível no Blog de Jonathan Hui e o curso online da Udacity.

Referências

[Arafa et al. 2019] Arafa, Y., Badawy, A.-H., Chennupati, G., Santhi, N., and Eidenbenz, S. (2019). Instructions' latencies characterization for nvidia gpgpus. *arXiv preprint arXiv:1905.08778*.

[Cheng et al. 2014] Cheng, J., Grossman, M., and McKercher, T. (2014). *Professional Cuda C Programming*. John Wiley & Sons.

[3] Clique aqui para fazer acesso aos laboratórios.

[4] Clique aqui para os códigos do livro CUDA C professional programming

[Cole et al. 2011] Cole, A., McEwan, A. A., and Singh, S. (2011). An analysis of programmer productivity versus performance for high level data parallel programming. In *Concurrent Systems Engineering Series 68*, pages 111–130.

[Ferreira and Canesche 2020] Ferreira, R. and Canesche, M. (2020). Github com colab do minicurso "métricas e números: Desmistificando a programação de alto desempenho em gpu", wscad 2019. https://github.com/cacauvicosa/wscad2019.

[Ferreira et al. 2019] Ferreira, R., Nacif, J., and Viana, S. (2019). Métricas e números: Desmistificando a programação de alto desempenho em gpu. In Menotti, R. and Galante, G., editors, *Minicursos do XXX Simpósio em Sistemas Computacionais de Alto Desempenho, WSCAD*, chapter 1, pages 5–34. SBC.

[Jaegeun Han 2019] Jaegeun Han, B. S. (2019). *Learn CUDA Programming*. Packt.

[Jia et al. 2019] Jia, Z., Maggioni, M., Smith, J., and Scarpazza, D. P. (2019). Dissecting the nvidia turing t4 gpu via microbenchmarking. *arXiv preprint arXiv:1903.07486*.

[John Cheng 2016] John Cheng, Max Grossman, T. M. (2016). Code samples. https://media.wiley.com/product_ancillary/29/11187393/DOWNLOAD/CodeSamples.zip.

[Mei and Chu 2016] Mei, X. and Chu, X. (2016). Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86.

[Nick 2020] Nick (2020). Gpgpu programming with cuda. https://github.com/CoffeeBeforeArch/cuda_programming.

[Nvidia 2020] Nvidia (2020). Cuda samples. https://github.com/NVIDIA/cuda-samples.

[Pereira 2011] Pereira, F. M. Q. (2011). Técnicas de otimização de código para placas de processamento gráfico. In *XXXI Congresso da SBC Jornada de Atualização da Informática*.

[Serpa et al. 2019] Serpa, M. S., Moreira, F. B., Navaux, P. O., Cruz, E. H., Diener, M., Griebler, D., and Fernandes, L. G. (2019). Memory performance and bottlenecks in multicore and gpu architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 233–236. IEEE.

[Volkov 2010] Volkov, V. (2010). Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA.