COMPUTAÇÃO PARALELA

AULA 6

Prof. Leonardo Gomes

CONVERSA INICIAL

Nesta aula, discutiremos o processamento em GPU, em especial com o API CUDA, uma importante tecnologia utilizada para paralelismo e computação de alto desempenho. Vamos debater o uso eficiente da memória e de modelos de comunicação que se aplicam em todo o tipo de processamento paralelo, mas que é especialmente importante de ser observado quando se programa para GPU.

Ao final desta aula, esperamos atingir os seguintes objetivos:

1. Entendimento da arquitetura básica das GPU, memória e organização dos núcleos;
2. Entendimento do modelo de programação paralela utilizando a API CUDA, principais comandos e sincronismo;
3. Capacidade de avaliar e entender o uso adequado da memória e identificação de diferentes modelos de comunicação.

TEMA 1 – INTRODUÇÃO AO CUDA

CUDA é uma API voltada para o processamento de propósito geral dentro das placas gráficas. Ao longo desta aula, discutiremos como iniciar nessa tecnologia e a que finalidades ela atende.

As placas gráficas GPU são projetadas pensando em processamento gráfico e renderização dinâmica de imagens em uma velocidade muito maior do que uma CPU padrão, pensada com propósito de uso geral. A estratégia básica das GPUs consiste em paralelizar a atividade de processamento vetorial em um número muito grande de núcleos semelhante ao sistema SIMD da taxonomia de Flynn.

A tecnologia CUDA foi desenvolvida pela Nvidia em 2006 e oferece um ganho em desempenho muito significativo em algoritmos altamente paralelizáveis. Infelizmente até o momento essa API é suportada apenas para placas de vídeo da própria Nvidia.

1.1 Sobre as GPUs

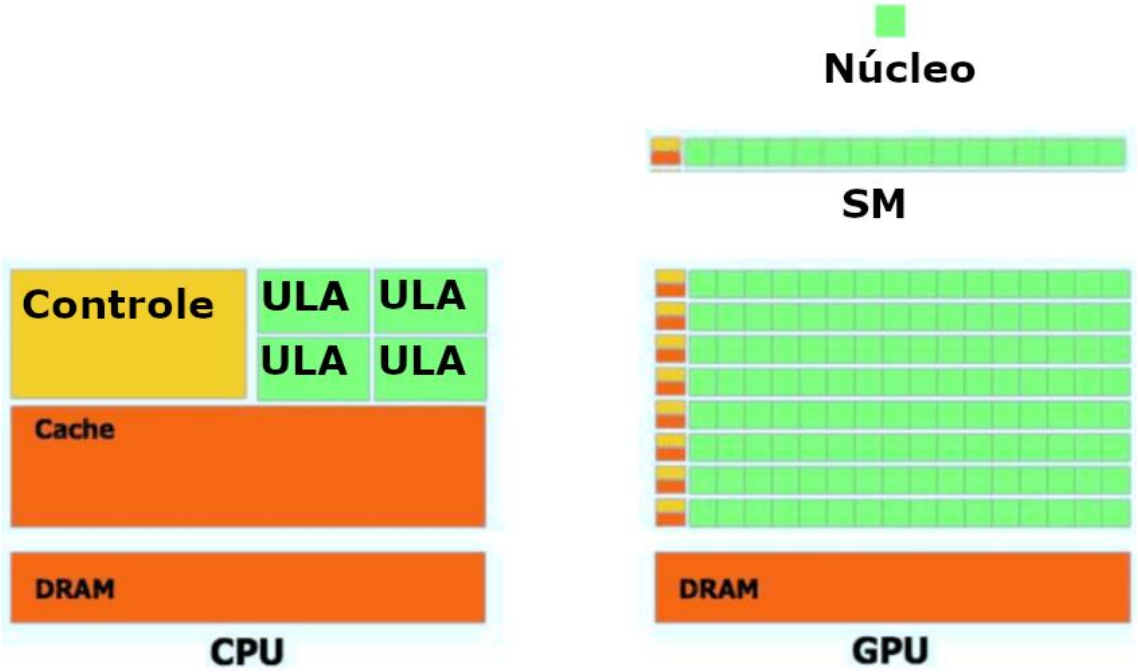
As placas de vídeo, desde seu projeto, são pensadas em paralelo e por isso suas arquiteturas são muito diferentes das CPUs tradicionais. Algumas das GPUs são amplamente comercializadas para público doméstico e chegam a ter

aproximadamente 4 mil núcleos, memória RAM própria e ampla largura de banda ao barramento, o que permite comunicação muito rápida e com pouco acesso ao barramento.

As *threads* geradas para GPU CUDA são diferentes das *threads* comuns geradas pelo sistema operacional. A gerência e troca de contexto é muito mais rápida, visto que os projetos em CUDA são pensados para trabalhar com milhares de *threads*, o que difere de quando trabalhamos com CPU, cujo processamento paralelo comumente leva em conta não mais do que poucas dezenas de *threads* por CPU.

Cada um dos núcleos da placa de vídeo é simples e não muito mais do que uma ULA das CPU's tradicionais. Por isso trechos dos programas que possuam uma grande interdependência de informações de forma que não apresentem possibilidade de paralelismo terão um desempenho inferior na GPU quando comparados a uma CPU tradicional. Enquanto as CPUs possuem um pequeno conjunto de núcleos poderosos, as GPUs consistem de diversos *streaming multiprocessors* (em tradução livre, *multiprocessadores de fluxo*). Ao longo desta aula, nós os chamaremos apenas de SM, que são unidades de processamento com uma única controladora e memória própria e diversos núcleos de processamento. A quantidade de SMs pode chegar até algumas dezenas, já os núcleos por SM podem chegar até algumas centenas. Ao todo, é possível chegar até perto de 4 mil núcleos CUDA. Na Figura 1, vemos a arquitetura básica de uma GPU.

Figura 1 – Representação lado a lado de uma CPU e uma GPU



Fonte: Weiss, 2010, p. 15.

1.2 Instalação do CUDA

O CUDA foi originalmente desenvolvido para ser compatível com C, C++ e Fortran, no entanto já temos ferramentas que promovem soluções para Python, Java e F# da Microsoft.

Saiba mais

Para instalar o CUDA, é necessário primeiro possuir uma placa de vídeo Nvidia compatível e fazer o *download* das bibliotecas necessárias. As instruções detalhadas para cada sistema operacional e para verificação de compatibilidade da sua placa podem ser encontradas no *link* a seguir da própria Nvidia.

GPU accelerated computing with C and C++. **Nvidia**, S.d. Disponível em: <https://developer.nvidia.com/how-to-cuda-c-cpp>. Acesso em: 30 out. 2019.

Em linhas gerais, é necessário seguir três passos principais:

1. Ter os *drivers* da placa de vídeo atualizados;
2. Instalar o chamado *CUDA Toolkit* que contém todas as bibliotecas necessárias;
3. Configurar as variáveis de ambiente para apontarem os diretórios onde as bibliotecas e demais arquivos estão instalados.

O Visual Studio conta inclusive com o *plug-in* chamado Nsight, que facilita a instalação e a criação de novos projetos dentro da API.

Os arquivos dos códigos fonte, embora sejam escritos em linguagem C/C++ ou Fortran padrão, possuem extensão **.cu** e utilizam o compilador **nvcc** que internamente divide o código em trechos executados pela GPU e CPU. A parte da CPU é compilada por um compilador padrão, já a parte GPU é compilada pela solução própria da Nvidia.

1.3 Conceitos gerais sobre CUDA

Podemos classificar os computadores com placa de vídeo como arquitetura heterogênea com dois processadores, o *host* (do inglês anfitrião) que é representado pela CPU, e o *device* (do inglês dispositivo), representado pela GPU. Os programas feitos com a API CUDA tornam possível que um mesmo programa tenha acesso aos dois processadores, permitindo que trechos com grande interdependência de informação que não podem ser paralelizados sejam rodados na CPU, que possui grande latência e trechos sem a mesma interdependência para ser rodado em paralelo na GPU, que possui grande vazão.

Em um programa simples, costumeiramente temos o seguinte fluxo.

- Primeiro CPU aloca memória;
- Dados são copiados para memória da GPU;
- Solicita a execução de uma função, chamada aqui de *kernel* em múltiplas *threads* paralelas na GPU;
- GPU executa o *kernel,* sobre os dados, geralmente o código é escrito para rodar em paralelo, mas sem assumir um número fixo de *threads*;
- CPU novamente acessa memória da GPU para obter resultado.

TEMA 2 – PRIMEIRO CÓDIGO

Neste tema, vamos analisar nosso primeiro código para GPU. Os primeiros comandos e a lógica por trás deles.

Neste código, iremos pegar todos os elementos de um *array* e elevá-los ao quadrado.

```
#include <stdio.h>
// código do kernel
__global__ void quadrado(float * d_saida, float * d_entrada){
    int idx = threadIdx.x;
    float f = d_entrada[idx];
    d_saida[idx] = f * f;
}

int main(int argc, char ** argv) {
      const int TAMANHO = 64;
      const int ARRAY_BYTES = TAMANHO * sizeof(float);

      // h_entrada e h_saida são as memória do host
      float h_entrada[TAMANHO];
      for (int i = 0; i < TAMANHO; i++) {
            h_entrada[i] = float(i);
      }
      float h_saida[TAMANHO];

      // ponteiro para a memória do device
      float * d_entrada;
      float * d_saida;

      // alocando memória
      cudaMalloc((void**) &d_entrada, ARRAY_BYTES);
      cudaMalloc((void**) &d_saida, ARRAY_BYTES);

      // transferindo para a GPU
      cudaMemcpy(d_entrada, h_entrada, ARRAY_BYTES,
cudaMemcpyHostToDevice);

      // rodando o kernel
      quadrado<<<1, TAMANHO>>>(d_saida, d_entrada);
```

```
	// copia a resposta novamente para a CPU
	cudaMemcpy(h_saida, d_saida, ARRAY_BYTES,
cudaMemcpyDeviceToHost);

	// imprime a resposta array
	for (int i =0; i < TAMANHO; i++) {
		printf("%f", h_saida[i]);
		printf(((i % 4) != 3) ? "\t" : "\n");
	}

	cudaFree(d_entrada);
	cudaFree(d_saida);

	return 0;
}
```

Conforme comentado anteriormente, o *kernel* é a função que é executada em GPU por diversas *threads* simultaneamente. Ela, em que devemos descrever o código da perspectiva de uma única *thread*, é marcada com a diretiva __**global**__. Em nosso exemplo, cada *thread* receberá então um ponteiro para a posição na memória da GPU onde está localizado o *array* com os dados de d_entrada e d_saída. Com a instrução **threadIdx.x** recuperamos a identificação da *thread* em questão. É possível organizar conjuntos de *threads* em blocos. Nesse código específico, estamos operando com um único bloco de 64 *threads*, e o comando em questão retorna um valor inteiro entre 0 e 63, distinguindo a *thread*, a qual então irá calcular um único valor quadrado da posição do *array*, que coincide com o seu índice e irá salvar a resposta na mesma posição do *array* de saída.

Nas primeiras linhas da função *main***,** temos a criação de um *array* com os valores inicializados com de 0 até 63 iguais seus índices. Na linha de código **cudaMalloc((void**) &d_entrada, ARRAY_BYTES);** a memória é alocada na placa de vídeo, já na linha **cudaMemcpy(d_entrada, h_entrada, ARRAY_BYTES, cudaMemcpyHostToDevice);** o conteúdo da memória do *array* h_entrada (*array* declarado no *host*) é copiado para entrada (*array* declarado na memória do *device*). O último parâmetro

cudaMemcpyHostToDevice indica a origem e o destino da memória, no caso, saindo do *host* para o *device,* já **cudaMemcpyDeviceToHost** indica o fluxo contrário. O comando **cudaFree(d_entrada);** por sua vez desaloca a memória do ponteiro indicado na memória do *device.* Os comandos em funcionalidade são equivalentes ao **malloc**, **memcpy** e **free** tradicionais do C**.**

O código **quadrado<<<1, TAMANHO>>>(d_saida, d_entrada);** por sua vez é a chamada do *kernel* quadrado que deverá ser executado por um bloco de 64 *threads* (constante TAMANHO armazena o valor 64 neste exemplo), os parâmetros **d_entrada** e **d_saida** são passados ao *kernel.*

2.1 Blocos e *threads*

Dentro do comando que invoca o *kernel* – no nosso exemplo, especificamente **quadrado<<<1, TAMANHO>>>(d_saida, d_entrada);** –, temos o operador <<<int, int >>>. O primeiro valor marca a quantidade de blocos, e o segundo, a quantidade *threads* por bloco. As GPUs hoje suportam até 512, 1024 ou 2048, dependendo da placa específica. Uma grande quantidade de processamento pode ser dividida em um número qualquer de blocos, porém não pode ultrapassar o valor limite de *threads.*

Dependendo da natureza do problema (por exemplo, no processamento de imagens), pode ser interessante considerar as *threads* em mais de uma dimensão. Por isso tanto a quantidade de blocos quanto a quantidade de *threads* podem ser informadas como 3 dimensionalidades distintas dentro da *struct* **dim3(x,y,z).** Ficaria aproximadamente assim:

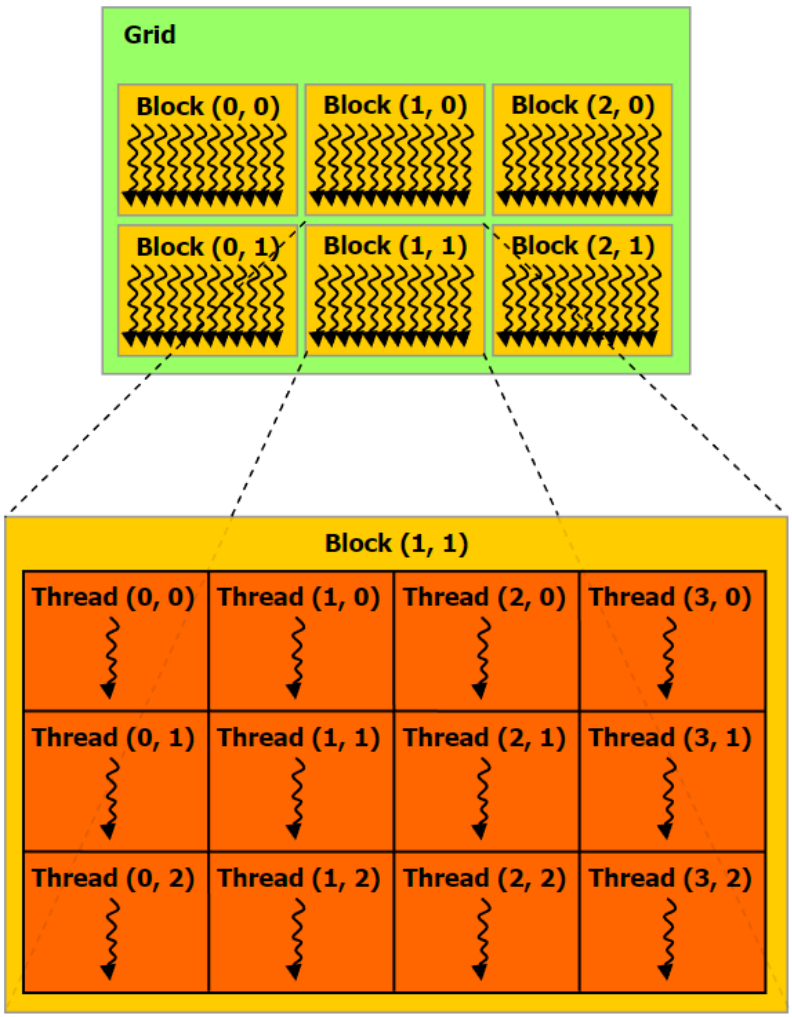
kernel<<<dim3(x1,y1,z1), dim3(x2,y2,z2)>>>(...);

Se algum dos valores de y1, z1, y2 ou z2 não forem definidos, ele assume uma dimensionalidade de 1. De dentro do *kernel* é possível recuperar a informação sobre o índice e a quantidade tanto de blocos quanto de *threads*.

- **threadIdx.x, threadIdx.y** e **threadIdx.z:** são os 3 índices das *threads* dentro do bloco;
- **blockDim:** O tamanho do bloco, quantidade de *threads*;
- **blockIdx.x, blockIdx.y** e **blockIdx.z:** são os 3 índices dos blocos em si;
- **gridDim:** tamanho da grade, quantidade de blocos.

Em linhas gerais, é interessante utilizar múltiplas dimensões quando o problema que se está abordando também está em múltiplas dimensões.

Figura 2 – Representação visual dos blocos divididos em duas dimensões dentro da grade e das *threads* também divididas em duas dimensões



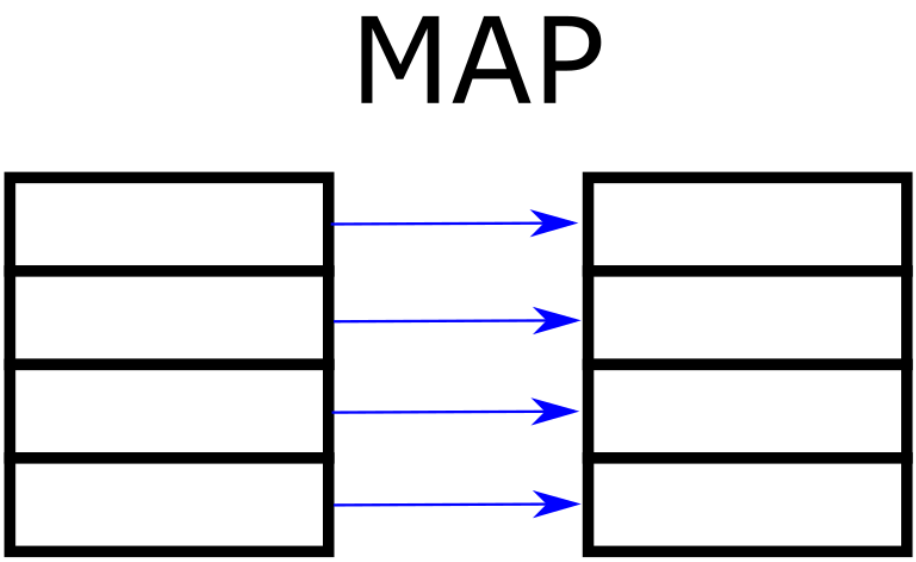
Fonte: Cuda Programming, S.d.

TEMA 3 – PADRÕES DE COMUNICAÇÃO

Existem diversos padrões de comunicação entre as *threads.* Isso resume como será a relação entre memória e as atividades desenvolvidas pelas *threads*.

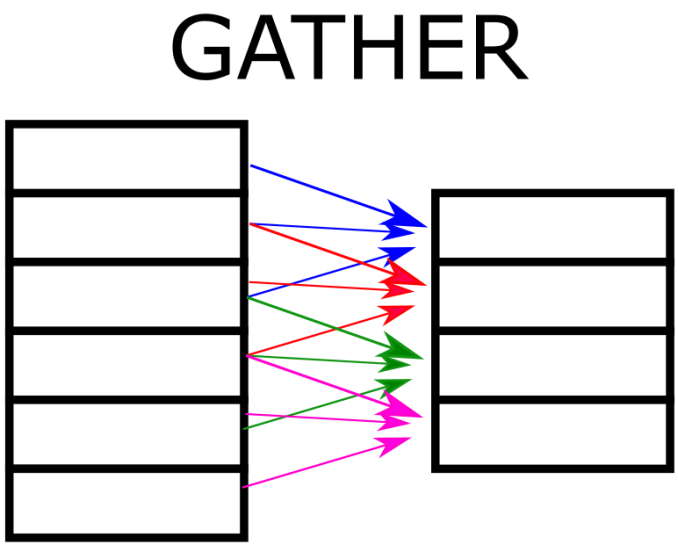
O padrão de comunicação mais simples é o chamado *Map*, de acordo com o qual cada *thread* lê um único dado e realiza processamento sobre o este e por fim escreve em uma única posição da memória. O exemplo do quadrado reflete esse padrão de comunicação. A correspondência aqui é de uma posição para leitura e uma posição para escrita. Esse modelo é muito eficiente em GPU, no entanto acaba contando com pouca flexibilidade.

Figura 3 – Representação do padrão *map*



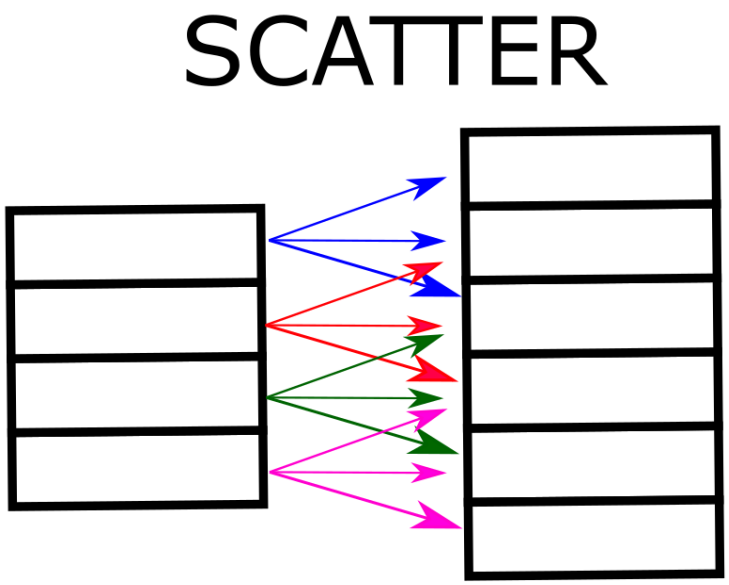
O padrão de comunicação *gather***,** no entanto, realiza a leitura de várias posições na memória e escreve em uma única posição ao final. Por exemplo, suponha que se deseja tirar a média dos valores vizinhos em conjuntos de três em três e salvar essa informação em uma posição diferente para cada conjunto conforme ilustrado na Figura 4. A correspondência aqui é de muitos na leitura para um na escrita.

Figura 4 – Representação do padrão *gather*.



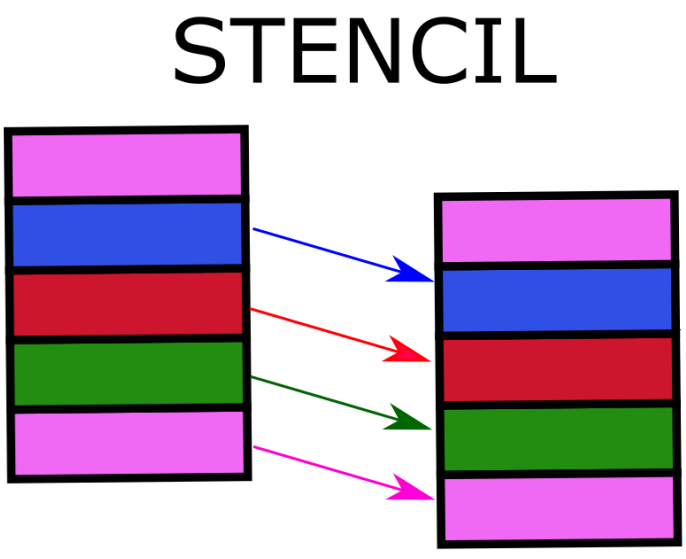
O padrão *scatter*, por outro lado, é o inverso do *gather*, pois o dado lido de uma única posição é processado e distribuído entre múltiplas posições na memória. A correspondência aqui é de um na leitura para muitos na escrita. Esse padrão pode introduzir um problema de sincronismo e condição de corrida para múltiplas *threads*, tentando escrever na mesma posição na saída.

Figura 5 – Representação do padrão *scatter*



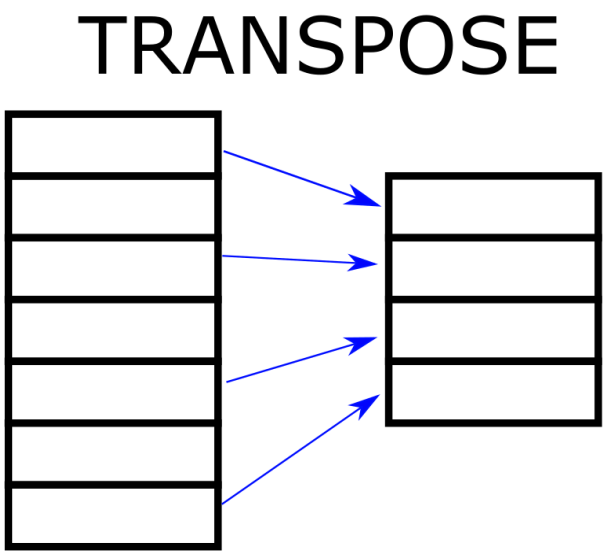
No padrão *stencil*, cada *thread* faz a leitura dos dados seguindo um determinado padrão predeterminado e faz a escrita em outro padrão pré-fixado da mesma forma. Pensando em aplicações de processamento de imagens, esse padrão de comunicação é bastante comum. Por exemplo, as posições próximas de um ponto da imagem em determinado padrão de vizinhança podem ser utilizadas para calcular a média dos valores dos pontos ou identificação de padrões. Nesse tipo de padrão, existe muito reúso de leitura de dados. Duas posições vizinhas acessam muitas posições em comum para leitura, o que permite certas otimizações quando uma mesma *thread* processa esses mesmos dados em sequência, otimizando o uso da *cache*.

Figura 6 – Representação do padrão *stencil*.



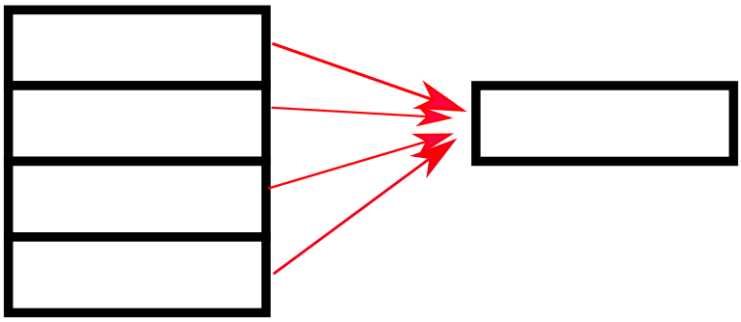
O padrão *transpose* é bastante utilizado para adaptar estruturas de dados. Geralmente quando temos um conjunto de dados de uma determinada entidade, nós o representamos em uma *struct*. Se desejamos representar várias dessas entidades, a forma mais natural seria utilizar um *array,* portanto um *array* de *struct*. No entanto, para otimizar o processamento, fazendo um melhor uso de memória *cache*, dentre outros recursos específicos dos processadores, o melhor é ter uma *struct* de *arrays*, permitindo que os mesmos tipos de dados estejam em sequência na memória.

Figura 7 – Representação do padrão *transpose*.



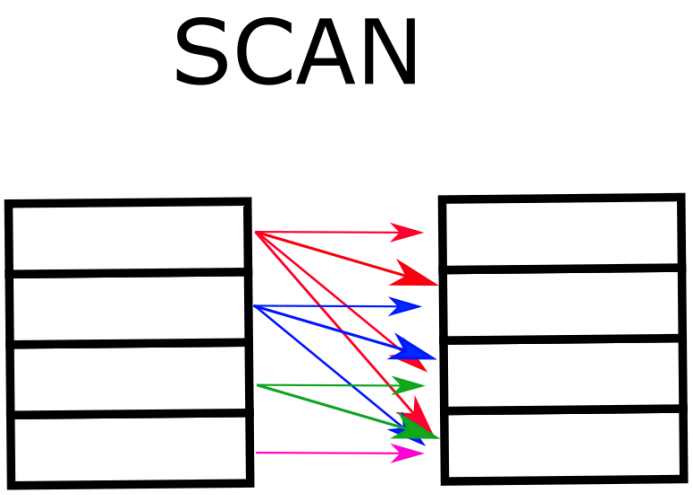
O padrão *reduce*, muito utilizado em aplicações de *machine learning*, entre outras aplicações estatísticas, consiste em transformar um grande conjunto de dados iterativamente em conjuntos menores a ponto de chegar a uma unidade. Alguns problemas computacionais que são simples, como a média de um *array*, podem parecer impossíveis de paralelizar em um primeiro momento, no entanto podemos repensá-lo como um *reduce*, que é paralelizável.

Figura 8 – Representação do padrão *reduce*



Outro padrão é o *scan*, que combina todos os dados de entrada com todos os de saída, sendo utilizado em aplicações como ordenação que demandam manipulação de todo o conjunto de dados.

Figura 9 – Representação do padrão *scan*



Em geral, o fluxo de um programa CUDA consiste no código principal no *host*, que invoca blocos de *threads* para executarem *kernels.* Cada bloco é alocado para diferentes SM de forma completamente transparente pela GPU. O programador não possui controle sobre a forma como a GPU distribui os blocos dentro das SMs. É importante observar que cada bloco roda em somente uma única SM, no entanto uma mesma SM pode rodar vários blocos simultâneos.

Numa primeira análise, essa falta de controle e de garantias do escalonamento de processos na GPU pode ser entendida como uma limitação, pois realmente não existe uma forma de comunicação direta entre blocos, no entanto ela possui várias vantagens:

- A GPU garante eficiência no uso dos recursos mantendo a GPU ocupada o máximo de tempo;
- Caso uma *thread* apresente lentidão não existe travamento dos demais núcleos, e novos blocos serão alocados nos núcleos conforme são liberados;
- Existe um ganho quanto à escalabilidade também. Não é necessário repensar o código para cada placa específica. Aumentando a quantidade de GPUs ou trocando a GPU por uma nova, o sistema automaticamente é adaptado.

As *threads* de um mesmo bloco rodam simultaneamente em uma SM. E todos os blocos dentro de um *kernel* terminam completamente sua execução antes que o *host* consiga iniciar a execução do próximo *kernel*.

TEMA 4 – MEMÓRIA EM CUDA

Um dos aspectos mais importantes ao se trabalhar com otimização de códigos e alta *performance* é o uso eficiente da memória. Neste tema, vamos discutir a estrutura da memória em CUDA e boas práticas de programação neste sentido.

As *threads* CUDA possuem basicamente 4 espaços diferentes de memória com o qual é possível trabalhar:

- Cada *thread* possui uma *memória local*, privada para cada *thread*;
- As *threads* também tem acesso a uma *memória compartilhada***,** que pode ser acessada por todas *threads* do bloco e fisicamente fica dentro da SM;
- Temos também uma *memória global* que pode ser acessada por *threads* que estejam em blocos diferentes e inclusive em *kernels* diferentes que executem antes ou depois;
- E por fim temos o que chamamos a *memória do host* que é a memória tradicional da CPU. Geralmente os dados são copiados entre a memória do host para e a memória global da GPU antes e depois da execução dos *kernels*. Embora seja menos eficiente, na maioria dos contextos é possível que as *threads* da GPU acessem diretamente a memória do *host*.

Na intenção de otimizar o uso de memória, uma boa prática é identificar dados que são frequentemente acessados e migrá-los para memórias que sejam mais rápidas. Como regra geral, a memória local é mais rápida que a memória

compartilhada, que, por sua vez, é muito mais rápida do que a memória global, que, por sua vez, também é muito mais rápida do que a memória do *host*: local > compartilhada >> global >> *host*.

Para declarar uma posição de memória que seja local, basta declarar a variável dentro do *kernel*. O padrão é a geração de variáveis locais. Para fazer alocar um espaço compartilhado, utilizamos a palavra reservada __shared__ antes do tipo da variável. Agora, para declarar uma variável global, a ideia é que o espaço deve ser alocado pelo *host* e um ponteiro para o endereço alocado que seja passado por parâmetro na chamada do *kernel*. A seguir, o código CUDA ilustra esses diferentes acessos à memória:

```
#include <stdio.h>
  //O parâmetro array é um ponteiro para memória global
__global__ void memoria_compartilhada(float *array){
    // variáveis locais em cada thread.
    int i, indice = threadIdx.x;
    float media, soma = 0.0f;

    //variável compartilhada com outras threads do bloco.
    //esta variável deixa de existir com o fim do bloco.
    __shared__ float sh_arr[128];

    //copiando o dado da memória global para compartilhada
    //cada thread copiando um único elemento.
    sh_arr[indice] = array[indice];

    __syncthreads(); //garante que toda a copia foi feita.

    //Cada posição i do array receberá a média de todos os
elementos com índice menor do que i. Exemplo: na posição 5
receberá a média dos elementos de 0 até 4.
    for (i=0; i<indice; i++) { soma += sh_arr[i]; }
    media = soma / (indice + 1.0f);
```

```
    //Mudança variável global visível fora do kernel
    array[indice] = media;
    //Mudança na variável compartilhada, esta linha não tem
efeito pois a variável compartilhada acaba junto do bloco.
    sh_arr[indice] = 3.14;
}

int main(int argc, char **argv){

    //padronização h_ (memória host) e d_ (memória device)
    float h_arr[128];
    float *d_arr;
   //Memória alocada e copiada para o device
    cudaMalloc((void **) &d_arr, sizeof(float) * 128);
    cudaMemcpy((void *)d_arr, (void *)h_arr, sizeof(float) *
128, cudaMemcpyHostToDevice);
     //chamada do kernel 1 bloco e 128 threads
    memoria_compartilhada<<<1, 128>>>(d_arr);
    //Memória copiada de volta para o host depois de alterada
    cudaMemcpy((void *)h_arr, (void *)d_arr, sizeof(float) *
128, cudaMemcpyHostToDevice);
    return 0;
}
```

TEMA 5 – SINCRONISMO

Sincronismo é um dos problemas principais quando se trata de programação paralela, como evitar condições de corrida comprometendo ao mínimo o desempenho. Neste tema vamos debater algumas estratégias de sincronismo em CUDA.

5.1 Barreira

Uma das principais instruções de sincronismo dentro de CUDA é o conceito de **barreira**. Suponha um conjunto de *threads* executando um mesmo

código. Cada uma está em um ponto diferente da execução, mas não temos garantia de que todas executarão as instruções no mesmo ritmo e manterão o mesmo passo. É possível, por exemplo, que uma *thread* esteja finalizando determinada região de código enquanto outra mal começou. Por diversas razões, pode ser interessante garantir que, em um determinado ponto do código, todas as *threads* finalizem a execução de determinada região antes de prosseguir.

Por exemplo, suponha um código vise deslocar em uma posição todos os elementos de um *array*. O valor da posição (x+1) é copiado para a posição x. Em uma primeira tentativa, poderíamos pensar em fazer um *kernel* aproximadamente assim:

```
__global__ void desloca(int * array){
      int    idx    =    threadIdx.x;
   array[idx] = array[idx +1];
}
```

Neste *kernel* cada *thread* irá deslocar um único valor da posição igual ao seu índice +1 para a posição igual ao seu próprio índice. No entanto, sem uma garantia de sincronismo, diferentes execuções podem gerar diferentes resultados, dependendo da ordem que as *threads* executarem. A forma de solucionar esse problema seria dividir o comando **`array[idx] = array[idx +1];`** em dois e inserir o comando de barreira. Da seguinte forma:

```
__global__ void desloca(int * array){
      int    idx    =    threadIdx.x;
   int temp = array[idx +1];
   __syncthreads();
   array[idx] = temp;
}
```

Nesta segunda versão, garantimos que todos as *threads* farão a leitura do valor do *array* e antes de qualquer *thread* seguir com a execução do código, o comando **`__syncthreads();`** irá forçar uma interrupção na execução até que

todas as *threads* cheguem àquele ponto da execução. Só então as *threads* seguem para gravar o valor lido na posição adequada.

5.2 Operações atômicas

Visando evitar condições de corrida, por vezes é interessante realizarmos operações atômicas, ou seja, mutuamente excludentes. Por exemplo, suponha uma variável **X = 10;**. Se duas *threads* incrementarem em 1 a mesma variável X, esperaremos um valor 12 como resposta. No entanto se a mesma variável for lida simultaneamente pelas *threads* antes que a outra tenha oportunidade de atualizar o valor, as duas irão incrementar para o valor 11. O interessante é que o comando de incrementar seja atômico, e a leitura e a atualização do valor ocorram de forma a bloquear outras *threads* que tentem acessar determinada posição de memória.

Os comandos atômicos são os seguintes: todos eles recebem 2 parâmetros, um valor e um endereço, e todas retornam o conteúdo original do endereço:

- atomicAdd() e atomicSub(), adição/subtração do valor no endereço;
- atomicExch(), salva o valor no endereço;
- atomicMin(), atomicMax(), descobre o menor/maior dentre endereço e valor e salva o salva no endereço;
- atomicInc(), atomicDec(), incrementa/decrementa em 1 se o valor for maior igual ao endereço;
- atomicCAS(), compara o valor com endereço. Se iguais, retorna o valor; caso contrário, retorna o antigo conteúdo do endereço;
- atomicAnd(), atomicOr(), atomicXor() realizam a operação lógica AND/OR/XOR e salvam o resultado no endereço.

A seguir, um pequeno exemplo que demonstra a forma inadequada de incrementar seguido da forma adequada utilizando atomicAdd.

```
__global__ void incrementa_errado(int * array){
   int idx = threadIdx.x;
   //incremento consecutivo, threads com índices maiores que o
tamanho do array tão a volta na posição igual ao tamanho.
   idx = idx % TAMANHO_ARRAY;
   array[idx] = array[idx] +1;
```

```
}
__global__ void incrementa_errado(int * array){
   int idx = threadIdx.x;
   //incremento consecutivo, threads com índices maiores que o
tamanho do array tão a volta na posição igual ao tamanho.
   idx = idx % TAMANHO_ARRAY;
   array[idx] = atomicAdd(&array[idx],1);
}
```

De forma geral, para aproveitar ao máximo a eficiência de uma GPU, temos algumas considerações sobre as regras gerais:

1. Métodos de sincronismo, quando possível, devem ser evitados, pois ele vem ao custo de desempenho. Sincronismo implica que certas *threads* ficarão paradas ocupando um núcleo enquanto aguardam o processamento de outras *threads*;
2. É conveniente transferir dados que são acessados com muita frequência para memórias mais rápidas. Local > compartilhada >> global;
3. Ao utilizar *arrays*, é conveniente trabalhar com dados de forma sequencial em vez de fazer acessos espalhados. Ao acessar uma posição na memória, um conjunto de dados vizinhos é levado para a memória *cache*. Se o acesso seguinte for em uma posição vizinha, certamente o dado já se encontrará na *cache* interna ao processador, minimizando o acesso à memória de verdade;
4. Evite divergência nos *kernels*. Por divergência entendem-se *threads* distintas, especialmente aquelas em um mesmo bloco, realizando atividades diferentes, um código com um *if* que fará certas *threads* realizarem códigos distintos, potencialmente um muito mais longo que outro. Ou um mesmo um *loop* que certas *threads* farão muito mais repetições que outras, tornando o processo ineficiente para algumas.

FINALIZANDO

Nesta aula, seguimos os nossos estudos comentando as arquiteturas paralelas das GPUs e como programá-las para aplicações de uso geral. Discutimos em maiores detalhes a API CUDA da Nvidia, sua instalação, principais propriedades e fizemos nossos primeiros códigos. Estudamos a forma

de estruturação de *threads* em blocos e como trabalham com os núcleos CUDA. Vimos os padrões de comunicação *gather, scatter, stencil, transpose, reduce* e *scan*, suas aplicações e exemplos de situações nas quais são geralmente adotados.

Além disso, também discutimos o uso adequado da memória, as situações e os diferentes tipos e estratégias para otimização do acesso a ela.

Por fim, falamos de questões de sincronismo e sobre como evitar condições de corrida. Com esses estudos, temos uma boa visão geral sobre os elementos que compõem a GPU e seu uso para programação via API CUDA, o que permitirá desenvolver soluções de alta *performance* utilizando equipamento comercializado para o público doméstico.

REFERÊNCIAS

CUDA PROGRAMMING. Disponível em: <http://cuda-programming.blogspot.com/>. Acesso em: 30 out. 2019.

WEISS, R. M. **GPU – Accelerated data mining with swarm intelligence**. Honors Thesis – Department of Computer Science Macalester College, Minnesota, EUA, maio 2010.