Documentação Fletcher-3.0-Petrobras

Sequência de execução e funcionalidades:

Iniciamos a análise pelo arquivo main.c que é o responsável por iniciar o programa. A main recebe alguns parâmetros de entrada sendo eles:

- **Form prob** (**enum *Form* {ISO, VTI, TTI}):** O programa de simulação sísmica permite a escolha de diferentes formulações de problemas, e essas formulações são representadas pelo tipo enum Form. O Form pode assumir os valores ISO, VTI e TTI, que podem ser diferentes maneiras de simular a propagação das ondas sísmicas. No contexto da simulação sísmica, ISO, VTI e TTI podem se referir a diferentes tipos de anisotropia (uma propriedade que varia de acordo com a direção) leia: https://sbgf.org.br/mysbgf/eventos/expanded_abstracts/12th_CISBGf/SBGF_3099.pdf :

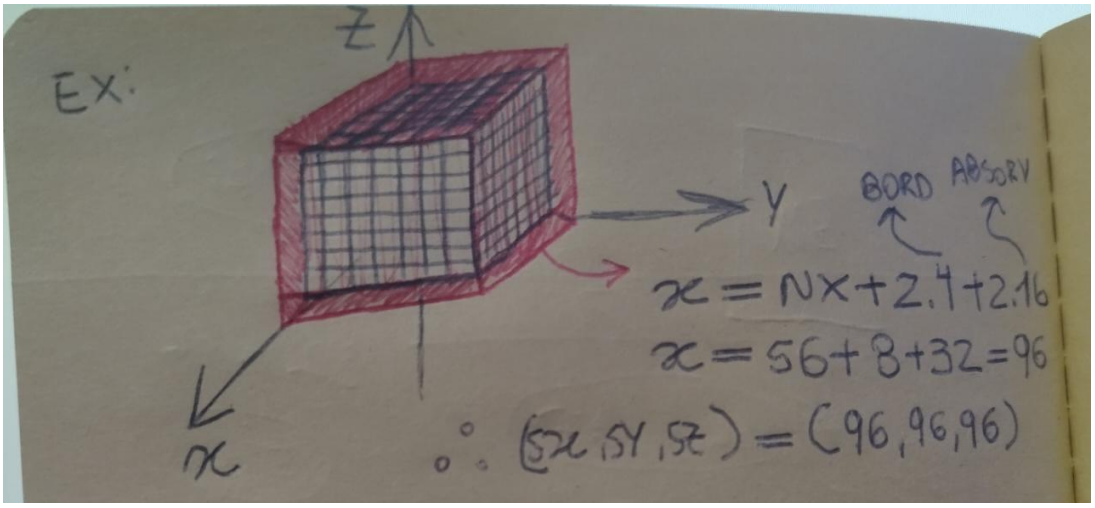
    - **ISO**: Isotrópico - Isso pode representar um meio no qual as propriedades da onda não mudam com a direção. Neste meio, as ondas sísmicas se propagam com a mesma velocidade em todas as direções.
    - **VTI**: Anisotropia de Transversal Verticalmente Isotrópica - Isso pode representar um meio em que as propriedades das ondas sísmicas variam com a direção, mas são isotrópicas no plano vertical. Isto é, as propriedades da onda são as mesmas ao longo de qualquer direção vertical.
    - **TTI**: Anisotropia de Transversal Isotrópica Inclinada - Isso pode representar um meio onde as propriedades das ondas sísmicas variam com a direção e são isotrópicas ao longo de uma direção que é inclinada (não vertical).
    - Percebe-se que atualmente as execuções testadas passam o parâmetro TTI, ou seja, selecionamos a anisotropia em meios com isotropia transversalmente inclinada.

- **int nx, ny, nz:** Número de pontos na grade ao longo dos eixos x, y e z, respectivamente.
- **int bord:** Tamanho da borda usada para aplicar o stencil nas extremidades do grid. O stencil é uma técnica usada para calcular novos valores de uma função em um grid. Essa técnica de **STENCIL** é frequentemente usada em simulações numéricas que envolvem a resolução de equações diferenciais parciais (EDPs). Em uma simulação sísmica, por exemplo, você pode estar resolvendo a equação da onda, que é uma EDP. A equação da onda descreve como as ondas se propagam no tempo. Pode-se usar um stencil

para calcular o próximo valor de cada ponto na grade baseado em seus valores atuais e os valores de seus vizinhos. No entanto, existe um problema na borda do grid. Se você tem um ponto no limite do grid e tenta aplicar o stencil, você pode descobrir que alguns dos pontos vizinhos estão fora do grid, ou seja, você não tem valores para eles. Uma maneira de resolver este problema é estendendo a grade com uma "borda". Esta borda é preenchida com valores que permitem que você aplique o stencil mesmo na borda do grid.

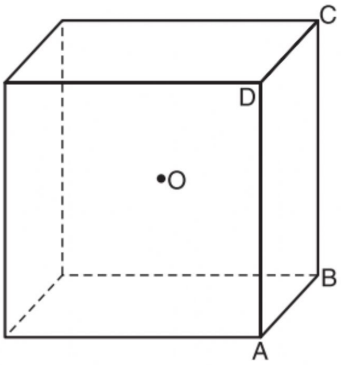
- **int absorb:** Tamanho da zona de absorção. Esta é uma região onde as ondas sísmicas são artificialmente amortecidas para evitar reflexões na borda da grade. Sem tal zona de absorção, as ondas que alcançam a borda da grade seriam refletidas de volta para a grade, o que é indesejado porque na realidade as ondas sísmicas continuariam se propagando além da borda do modelo. O valor **int absorb** no código é o número de pontos de grade na zona de absorção. No caso o **absorb** está setado para 16, isso significa que há 16 pontos de grade de largura na zona de absorção em todas as direções da borda do modelo.
- **float dx, dy, dz:** Passo do grid em x, y e z, que é a distância entre os pontos adjacentes da grade nessas direções. Por exemplo, se dx é 12.5, isso significa que cada ponto na grade ao longo do eixo x está a uma distância de 12.5 unidades do seu vizinho mais próximo. Da mesma forma, dy e dz são a distância entre pontos de grade adjacentes ao longo dos eixos y e z. O tamanho do passo da grade determina a resolução espacial da simulação. Uma resolução espacial maior (ou seja, um passo de grade menor) permitirá que você modele detalhes menores, mas também exigirá mais pontos de grade e, portanto, aumentará o custo computacional da simulação.
- **float dt**: Avanço no tempo em cada passo de tempo da simulação. O valor dt representa o tamanho do passo de tempo na simulação. Ele define a quantidade de tempo que a simulação avança a cada passo. No caso de dt ser definido como 0.001, isso significa que, a cada passo da simulação, o tempo avança 0.001 unidades.
- **float tmax:** Tempo final desejado para a simulação. No caso de tmax = 0.02, isso sugere que a simulação é projetada para estudar a propagação de ondas sísmicas durante um curto período de tempo após a origem da fonte.
- **int sx, sy, sz:** Dimensões totais do grid em x, y e z, que incluem os pontos do grid, as bordas e as zonas de absorção.
- **int ixSource, iySource, izSource:** Índices x, y e z da fonte de onda sísmica no grid.
- **int iSource:** Índice da fonte mapeado em um array 1D.

Após isso é calculado o valor de sx, sy e sz que seriam as dimensões totais do grid em x, y e z, que incluem os pontos do grid, as bordas e as zonas de absorção:



Logo após é calculado o número de iterações (usando a função “ceil” para arredondar para cima). O valor é calculado dividindo o tempo total de execução (tmax) pelo tempo de cada iteração (dt): **st = ceil(tmax/dt)**

Após isso é calculado a localização da fonte na simulação. Em particular, a fonte está sendo colocada no centro do grid.



Posteriormente usa-se a uma função “ind” que converte índices 3D em um índice 1D. iSource é então o índice 1D da fonte no grid. Em muitas simulações, o grid 3D é representado como um array 1D por motivos de eficiência, então essa função ind seria usada para mapear os índices 3D para um índice 1D.

// mapeando o array 3D [sz][sy][sx] em 1D [sx*sy*sz] na ordem da linha principal
// o uso requer implicitamente a definição das variáveis sx e sy no local da chamada
#define ind(ix,iy,iz) (((iz)*sy+(iy))*sx+(ix))
Posteriormente é verificado se as diretivas OpenMP e OpenACC estão habilitadas na compilação do programa e, se estiverem, imprimem algumas informações sobre a execução paralela. Após isso, vários arrays são alocados dinamicamente para armazenar informações sobre a anisotropia em cada ponto do grid da simulação. A anisotropia se refere à variação nas

propriedades físicas de um meio em diferentes direções. Em simulações sísmicas, a anisotropia é uma consideração importante, pois as velocidades de ondas sísmicas podem variar em diferentes direções dependendo das propriedades do material.

Cada array é inicialmente declarado como um ponteiro para float e então um bloco de memória é alocado para cada um usando a função malloc. O tamanho de cada bloco de memória alocado é o número total de pontos no grid (sx*sy*sz), multiplicado pelo tamanho de um valor float. Isso significa que cada array tem espaço suficiente para armazenar um valor float para cada ponto do grid.

Descrição do que cada um dos arrays de anisotropia estão armazenando:

- **vpz**: armazena a velocidade da onda P normal ao plano de simetria.
- **vsv**: armazena a velocidade da onda SV normal ao plano de simetria.
- **epsilon**: armazena o parâmetro isotrópico de Thomsen, epsilon, que descreve o grau de anisotropia.
- **delta**: armazena o outro parâmetro isotrópico de Thomsen, delta, que também descreve o grau de anisotropia.
- **phi**: armazena o ângulo de azimute de simetria isotrópica.
- **theta**: armazena o ângulo de profundidade de simetria isotrópica.

Após há um switch que analisa a entrada prob (TTI, ISO, VTI), como usamos TTI só será analisado o caso em que é um meio TTI. É nesse switch que está sendo inicializados os arrays de anisotropia que foram alocados dinamicamente anteriormente. Para cada ponto do grid, ele inicializa os valores nos arrays de anisotropia da seguinte maneira:

- Define a velocidade da onda P normal ao plano de simetria (vpz) como 3000.
- Define o parâmetro isotrópico de Thomsen epsilon (epsilon) como 0.24.
- Define o outro parâmetro isotrópico de Thomsen delta (delta) como 0.1.
- Define o ângulo de azimute de simetria isotrópica (phi) como 1.0. O comentário sugere que isso é para evitar coeficientes nulos.
- Define o ângulo de profundidade de simetria isotrópica (theta) como o arco tangente de 1.0.

Após isso também é garantida a estabilidade numérica da simulação através da definição de algumas variáveis. Em métodos numéricos para resolver equações diferenciais parciais, como as usadas em simulações sísmicas, a

relação entre o passo de tempo e o passo de espaço é crucial para garantir a estabilidade do método.

Seguindo adiante é realizada a primeira chamada de função externa sendo ela a RandomVelocityBoundary que é responsável por ajustar as velocidades P (vpz) e S (vsv) nas bordas e zonas de absorção do grid da simulação. A função tem como parâmetros as dimensões totais do grid (sx, sy, sz), as dimensões internas do grid (nx, ny, nz), o tamanho da borda (bord), a largura da zona de absorção (absorb) e os arrays vpz e vsv que representam as velocidades P e S em cada ponto do grid.

De forma breve ela realiza as seguintes tarefas:

- Inicialmente, a função calcula as velocidades P e S máximas dentro do grid interno da simulação (não incluindo a borda e a zona de absorção). Essas velocidades máximas são armazenadas nas variáveis maxP e maxS.
- A função, então, percorre todo o grid (incluindo bordas e zonas de absorção). Para cada ponto do grid, a função faz uma de três coisas, dependendo de onde o ponto se encontra:
    - a. Se o ponto está dentro do grid interno, nada é feito. A velocidade desse ponto já foi definida anteriormente.
      b. Se o ponto está dentro da zona de absorção, a velocidade é definida como uma combinação da velocidade no ponto correspondente do grid interno e uma velocidade aleatória. A proporção da velocidade interna e da velocidade aleatória depende da distância do ponto à borda do grid interno. Quanto mais longe o ponto estiver do grid interno, maior será a proporção da velocidade aleatória. A velocidade aleatória é calculada como maxP ou maxS (dependendo se estamos ajustando vpz ou vsv) multiplicado por um número aleatório entre 0 e 1.
      c. Se o ponto está na borda do grid (não na zona de absorção), as velocidades P e S são definidas como zero. Isso efetivamente impede qualquer propagação de ondas através desses pontos, agindo como uma "parede" perfeitamente refletora.

Após isso é realizada a alocação de memória para os campos de pressão nos passos de tempo anterior, atual e futuro. Os campos de pressão são representados por quatro arrays de float: pp, pc, qp e qc. Cada um desses arrays tem um tamanho de sx*sy*sz, que é o número total de pontos no grid.

- pp representa a pressão no passo de tempo anterior
- pc representa a pressão no passo de tempo atual
- qp representa a pressão no próximo passo de tempo
- qc representa a pressão no passo de tempo atual para outro campo de pressão (pode estar representando outro modo de vibração ou uma dimensão diferente)

Esses arrays são usados para resolver as equações diferenciais que regem a propagação das ondas sísmicas. O método numérico utilizado (normalmente uma variação do método de diferenças finitas) requer o conhecimento dos valores de pressão em diferentes passos de tempo para atualizar os valores de pressão no próximo passo de tempo.

Então é realizada a inicialização das variáveis que definem os limites das coordenadas x, y e z na grade tridimensional.

- ixStart, iyStart e izStart são as coordenadas iniciais em cada eixo (x, y, z), respectivamente. Estão definidas como 0, o que geralmente representa a borda inicial da grade em cada dimensão.
- ixEnd, iyEnd e izEnd são as coordenadas finais em cada eixo. Estão definidas como sx-1, sy-1 e sz-1, respectivamente. sx, sy e sz representam o número total de pontos na grade ao longo de cada eixo, então sx-1, sy-1 e sz-1 seriam os índices do último ponto na grade em cada dimensão (array começa em 0 logo “sx / sy / sz - 1” são os ultimos pontos na grade)

Após isso é realizada a chamada da segunda função denominada OpenSliceFile que está abrindo um arquivo no formato RFS que será continuamente adicionado durante a execução do programa.

1. A função começa definindo um ponteiro para uma estrutura chamada Slice, que é alocada dinamicamente usando malloc. Essa estrutura contém informações sobre o arquivo de fatia que está sendo aberto.
2. Em seguida, ela verifica em qual direção a "fatia" está ocorrendo. A direção da fatia é determinada por qual dos índices de início e fim são iguais (ixStart e ixEnd, iyStart e iyEnd, ou izStart e izEnd). Se os índices de início e fim de x forem iguais, a direção é definida como XSLICE. Se os índices de início e fim de y forem iguais, a direção é definida como YSLICE. Se os índices de início e fim de z forem iguais, a direção é definida como ZSLICE. Se nenhum deles for igual, a direção é definida como FULL.

3. Depois disso, ela cria os nomes dos arquivos de cabeçalho e binário, e abre esses arquivos para gravação ("w+").
4. Então, ela salva todas as informações relevantes na estrutura Slice que foi alocada no início. Isso inclui os índices de início e fim para cada direção, a contagem de iterações, os incrementos para cada direção, e o incremento de tempo.
5. Finalmente, ela retorna o ponteiro para a estrutura Slice.

Sendo essa a estrutura Slice:

```
enum sliceDirection {XSLICE, YSLICE, ZSLICE, FULL};


You, 3 weeks ago | 1 author (You)
typedef struct tsection {
  enum sliceDirection direction;
  int ixStart;
  int ixEnd;
  int iyStart;
  int iyEnd;
  int izStart;
  int izEnd;
  int itCnt;
  float dx;
  float dy;
  float dz;
  float dt;
  FILE *fpHead;
  FILE *fpBinary;
  char fNameHeader[128];
  char fNameBinary[128];
} Slice, *SlicePtr;
```

De maneira geral, essa função está preparando um arquivo e a estrutura de dados correspondente para que o programa possa gravar dados de uma "fatia" da grade tridimensional durante cada passo de tempo da simulação. Uma "fatia" aqui se refere a um subconjunto 2D da grade 3D.

Após isso é chamada a função DumpSliceFile que está escrevendo dados de uma "fatia" de um array 3D em um arquivo que foi previamente aberto. Ela faz isso em etapas:

- Para cada ponto ao longo das direções z e y dentro da fatia especificada pela estrutura Slice (delimitada por izStart, izEnd, iyStart, iyEnd, e ixStart), ela escreve os dados correspondentes do array 3D arrP no arquivo binário associado à estrutura Slice. Isso é feito usando a função fwrite, que escreve dados brutos em um arquivo. O número de elementos a serem escritos é determinado pela diferença entre ixEnd e

ixStart, acrescida de 1 (representando o intervalo inclusivo entre ixStart e ixEnd).
- Em seguida, ela incrementa a contagem de iterações na estrutura Slice (itCnt). Isso é usado para acompanhar quantas vezes a função foi chamada, ou seja, quantas vezes os dados foram anexados ao arquivo.

Portanto, a função DumpSliceFile está anexando uma fatia de dados do array 3D arrP em um arquivo no formato RFS, cujo nome e caminho são armazenados na estrutura SlicePtr p.

Finalmente, a função Model é chamada, ela é a que faz a maior parte do trabalho tendo a complexidade mais elevada que as demais e possuindo diversas chamadas a outras funções durante sua execução. A função Model() é responsável por conduzir uma simulação com base em vários parâmetros de entrada. Ela faz várias coisas, incluindo inicialização, execução de um loop de tempo, chamada para outras funções de processamento e gravação dos resultados.

- **Inicialização:** Ela começa inicializando variáveis e definindo alguns valores. Além disso, verifica se a funcionalidade PAPI (Performance Application Programming Interface) está ativada, que é usada para coletar informações sobre a performance da simulação. Se o PAPI estiver habilitado, ele inicializa os contadores correspondentes.
- **Inicialização do Driver:** Chama a função DRIVER_Initialize(), que inicializa o ambiente de computação ou recursos necessários para a simulação.
- **Loop de tempo:** Inicia um loop que irá durar o número especificado de passos de tempo (st). Em cada iteração:
    - Calcula o valor da fonte para o timestep atual e insere-o nos arrays de pressão e densidade com a função DRIVER_InsertSource().
    - Se o PAPI estiver ativado, inicia a coleta de dados de desempenho.
    - Chama a função DRIVER_Propagate(), que executa uma etapa da simulação.
    - Troca os ponteiros dos arrays que contêm a pressão e densidade do passo de tempo atual e do passo de tempo anterior.
    - Se o PAPI estiver ativado, termina a coleta de dados de desempenho e atualiza os contadores.

    - Verifica se é o momento de registrar um snapshot dos dados de simulação. Se for, chama DumpSliceFile() para gravar a pressão atual em um arquivo e atualiza o tempo para o próximo snapshot.
- **Finalização:** Após o término do loop de tempo:
    - Coleta e imprime dados sobre o desempenho da simulação, incluindo tempo total de execução, taxa de amostras por segundo e memória usada.
    - Grava esses dados de desempenho em um arquivo CSV.
    - Se o PAPI estiver ativado, grava os dados coletados pelo PAPI no arquivo CSV.
    - Chama a função DRIVER_Finalize(), que provavelmente faz a limpeza dos recursos alocados para a simulação.

Análise Detalhada do Funcionamento da Função Model

Inicialização das Váriáveis: Inicialmente é realizada a inicialização de algumas váriaveis posteriormente usadas no processo de simulação:

- **tSim=0.0**: Esta linha define tSim como zero. Parece que tSim será usado para rastrear o tempo de simulação atual.
- **int nOut=1**: Esta linha define nOut como um. nOut provavelmente é usado para rastrear o número atual de instantâneos ou saídas da simulação.
- **float tOut=nOut*dtOutput**: Aqui, tOut é calculado como o produto de nOut e dtOutput. tOut provavelmente representa o tempo no qual o próximo snapshot ou saída será produzido.
- **const long samplesPropagate=(long)(sx-2*bord)* (long)(sy-2*bord)*(long)(sz-2*bord)**: Esta linha calcula o número de amostras que serão propagadas em cada passo de tempo da simulação. É o produto das dimensões do espaço de simulação, menos duas vezes o tamanho da borda em cada dimensão.
- **const long totalSamples=samplesPropagate* (long)st**: Finalmente, esta linha calcula o número total de amostras que serão processadas durante toda a simulação. É o número de amostras propagadas em cada passo de tempo multiplicado pelo número total de passos de tempo.

Além disso, inicializa o sistema de monitoramento de desempenho chamado PAPI (Performance API), que é uma biblioteca que permite coletar informações sobre o desempenho do código durante a execução. Ele é ativado apenas se o código foi compilado com o suporte ao PAPI ativado. A função Model prepara o ambiente para coletar dados de desempenho usando o PAPI, se ele estiver disponível.

Após isso, O arquivo precomp.h está realizando cálculos preliminares e alocação de memória para vários arrays necessários para a computação no modelo de propagação de ondas sísmicas. A inclusão deste arquivo é condicionada pela definição de MODEL_INITIALIZE, o que significa que esses cálculos são realizados apenas durante a fase de inicialização do modelo. O precomp.h detalhadamente:

- O código aloca memória para vários arrays de coeficientes de derivadas **(ch1dxx, ch1dyy, ch1dzz, ch1dxy, ch1dyz, ch1dxz),** cada um do tamanho

da malha do problema (sx*sy*sz). Esses coeficientes são usados para calcular as derivadas espaciais nas equações diferenciais parciais que descrevem a propagação da onda.

- Em seguida, o código preenche esses arrays com valores calculados com base em ângulos de elevação e azimute (armazenados nos arrays theta e phi). Esses ângulos descrevem a direção da onda sísmica.
- Em seguida, o código aloca memória para vários arrays de velocidades **(v2px, v2pz, v2sz, v2pn)**, e preenche esses arrays com valores calculados com base nas velocidades das ondas P e S em diferentes direções (armazenados nos arrays vsv e vpz) e parâmetros de anisotropia (epsilon e delta). As velocidades das ondas P e S descrevem como as ondas sísmicas se propagam através do meio.
- Se a macro _DUMP está definida, o código imprime alguns dos valores calculados para diagnóstico ou depuração.

Posteriormente é realizada a chamada da função **DRIVER_Initialize** que é responsável por inicializar a simulação, passando uma série de parâmetros. Depois disso, ela chama **CUDA_Initialize,** que é a versão específica do CUDA dessa função de inicialização. Ou seja, **DRIVER_Initialize e CUDA_Initialize** juntos estão preparando a simulação para rodar na GPU, realizando cálculos preliminares, e movendo todos os dados necessários para a memória da GPU.

A função CUDA_Initialize faz várias coisas:

- Ela define um conjunto de ponteiros para a memória do dispositivo (dev_*), que serão usados para armazenar os campos de dados da simulação na GPU.
- Ela obtém o número de dispositivos CUDA disponíveis, seleciona o primeiro (dispositivo 0) e configura a GPU para uso.
- Ela verifica se os tamanhos sx e sy da grade são múltiplos de BSIZE_X e BSIZE_Y, respectivamente. Isso é necessário para garantir que a grade possa ser dividida de maneira uniforme entre os vários blocos de threads na GPU.
- Ela calcula o número total de células na grade (sxsy) e o tamanho total necessário para os arrays de dados (msize_vol e msize_vol_extra).
- Ela usa cudaMallocManaged e cudaMemPrefetchAsync para alocar memória na GPU para cada um dos arrays de dados e para mover esses dados para a GPU.
- Ela usa cudaMallocManaged, cudaMemset e cudaMemPrefetchAsync para alocar memória para os arrays de campo de onda pp, pc, qp e qc, e inicializá-los com zeros.
- Ela ajusta os ponteiros para pp, pc, qp e qc para apontar para o início do array, mais sxsy, ou seja, eles pulam a primeira "camada" da grade 3D.

- Ela usa cudaMallocManaged, cudaMemset e cudaMemPrefetchAsync para alocar e inicializar os arrays de campo de onda pDx, pDy, qDx e qDy.
- Ela verifica se houve algum erro ao chamar as funções CUDA e sincroniza o dispositivo.
- Imprime a quantidade de memória usada na GPU.

Após isso, o código entra em um loop que representa o passo do tempo de uma simulação física (muitas vezes em física computacional, especialmente na solução de equações diferenciais parciais, os cálculos são feitos iterativamente através de passos discretos de tempo). Cada passo do loop representa um passo no tempo (it), que vai de 1 até st. Dentro do loop, onde é realizado os seguintes passos:

1. A função Source(float dt, int it) está gerando uma onda sísmica, o pulso gaussiano modulado, para cada passo de tempo. O valor de retorno dessa função representa a amplitude da onda sísmica no tempo atual.
2. A função DRIVER_InsertSource(dt, it-1, iSource, pc, qc, src) é chamada para inserir o valor da fonte calculada na simulação. Esta função aceita o valor da fonte, a posição da fonte (iSource) e os arrays pc e qc, que representam os campos de ondas na simulação.
3. Dentro de DRIVER_InsertSource, a função CUDA_InsertSource(src, iSource, p, q) é chamada. Esta função usa CUDA para inserir o valor da fonte na simulação na GPU. A inserção é feita apenas se os ponteiros dev_pp e dev_qp (que são definidos externamente e, presumivelmente, apontam para os arrays de campo de onda na GPU) não são nulos.
4. Dentro de CUDA_InsertSource, um kernel CUDA chamado kernel_InsertSource é lançado para fazer a atualização real. O kernel CUDA kernel_InsertSource é um bloco de código que é executado na GPU. Este kernel em particular está sendo lançado com uma configuração de grade e bloco tal que um único thread seja executado. Isso é indicado pelo fato de que a condição if (ix==0) é verdadeira para apenas um thread. Uma vez que: const int ix=blockIdx.x * blockDim.x + threadIdx.x; if (ix==0). Nesse kernel, o valor da fonte (val), que foi calculado anteriormente pela função Source(float dt, int it), é adicionado às posições correspondentes (ou seja, iSource) nas matrizes qp e qc. Essas matrizes são versões da GPU das matrizes pc e qc usadas na função DRIVER_InsertSource. Portanto, o kernel CUDA é usado para inserir o valor da fonte calculado no campo de onda atual (ou seja, as matrizes qp e qc) na localização da fonte (iSource).
5. Após o lançamento do kernel, a função verifica se houve algum erro ao chamar as funções CUDA e sincroniza o dispositivo.

assim nesse trecho o código está avançando uma simulação física através de passos de tempo, calculando o valor de uma fonte em cada passo do tempo e inserindo esse valor na simulação usando a computação paralela na GPU.

Posteriormente é iniciada a contagem de eventos de desempenho usando a função PAPI_start(eventset) da biblioteca PAPI (Performance Application Programming Interface) se a opção PAPI estiver habilitada. A função PAPI_start(int EventSet) é uma função da biblioteca PAPI que inicia a contagem dos eventos no conjunto especificado. EventSet é um identificador para um conjunto de eventos que você deseja contar

Após inicializado o PAPI começa a simulação da propagação dessa onda. Inicialmente, é chamada a função wtime() para obter o tempo atual. Essa função retorna o tempo em segundos desde algum ponto de referência t0 é então usado para marcar o início de um período de tempo que está sendo medido, anterior ao inicio da simulação da propagação. Logo após é realizada a chamada da função DRIVER_Propagate que vai realizar essa propagação. Dentro da função DRIVER_Propagate é chamada a função CUDA_Propagate.

Análise detalhada do funcionamento da função CUDA_Propagate

A função CUDA_Propagate é responsável por configurar e lançar um kernel CUDA, que realiza alguma operação de propagação em um conjunto de dados. A função kernel_Propagate é esse kernel CUDA. As declarações de variáveis externas no início da função CUDA_Propagate se referem a variáveis que já foram alocadas e inicializadas. Estas variáveis estão na memória do dispositivo (GPU) e serão utilizadas pelo kernel CUDA.

A configuração para lançar o kernel CUDA é realizada através da definição das dimensões dos blocos e da grid. As variáveis threadsPerBlock e numBlocks são configuradas para determinar a organização e a quantidade de threads que serão lançados para executar o kernel CUDA. [ANÁLISE DE PERFORMANCE SENDO REALIZADA PELA BRENDA].

A função kernel_Propagate é um kernel CUDA que realiza a propagação numérica de um sistema modelado por equação de Onda. O arquivo sample.h é incluído duas vezes no kernel. A primeira vez é antes de um loop ao longo da terceira dimensão (z) da grade de dados, e a segunda vez é dentro do loop. Cada inclusão do arquivo sample.h é enquadrada por uma macro de definição (SAMPLE_PRE_LOOP e SAMPLE_LOOP), que controla quais partes do sample.h são incluídas em cada ponto.

Na inclusão SAMPLE_PRE_LOOP, várias constantes são definidas que são usadas para calcular derivadas parciais na próxima inclusão de sample.h. As constantes strideX, strideY, e strideZ definem o espaçamento entre pontos na grade de dados para cada direção. Além disso, inversos de termos relacionados a dx, dy e dz também são calculados para uso posterior.

A inclusão SAMPLE_LOOP dentro do loop é onde a maior parte do trabalho computacional acontece. Aqui, derivadas parciais segundo e de cross de cada um dos campos p e q são calculadas usando as funções Der2 e DerCross. Estas derivadas são então combinadas de várias maneiras para calcular h1p, h2p, h1q, h2q, h1pmq, e h2pmq. Estas quantidades são então usadas para calcular o lado direito das equações de propagação para p e q (rhsp e rhsq). Finalmente, os novos valores de p e q são calculados usando uma fórmula que envolve os valores anteriores e atuais de p e q, bem como rhsp, rhsq e dt.

Análise quanto a dependência de Dados na execução do Kernel:

O modelo de execução em CUDA é baseado em threads independentes. Cada thread tem seu próprio espaço de endereçamento e não compartilha estado com outras threads. Nesse caso, as threads estão executando a mesma função do kernel e cada thread lê e escreve de diferentes locais na memória global. Especificamente, **cada thread calcula seu próprio índice ix e iy, que são usados para acessar a memória global de forma independente**. Portanto, **não há dependências de dados entre as threads.**

Cada thread calcula as derivadas parciais do campo p e q na posição do índice da thread. Estas derivadas são então usadas para calcular h1p, h2p, h1q, h2q, h1pmq e h2pmq. **Logo, cada thread realiza este cálculo independentemente, sem a necessidade de comunicar ou sincronizar com outras threads.**

Após o cálculo de rhsp e rhsq, cada thread atualiza os valores de p e q na posição do índice da thread. Os novos valores são calculados com base nos valores antigos e atuais de p e q, bem como em rhsp, rhsq e dt. **Essa etapa de atualização também é realizada independentemente por cada thread.**

Após a conclusão do cálculo, todas as threads são sincronizadas antes de o controle ser devolvido ao CPU. Isso é feito através da chamada cudaDeviceSynchronize() na função CUDA_Propagate. Esta chamada bloqueia a execução do CPU até que todas as threads CUDA tenham concluído a execução, garantindo que todos os cálculos tenham sido concluídos e que todos os dados estejam consistentes antes que o programa continue.

A última operação na função CUDA_Propagate é um swap de ponteiros para os arrays pp e pc, e qp e qc. Sendo a função CUDA_SwapArrays uma implementação da técnica de "ping-pong", que é uma abordagem comum para lidar com dados sequencialmente dependentes em simulações onde a entrada atual é a saída do passo de tempo anterior. Em uma simulação típica de propagação temporal, o estado atual de um onda depende do estado anterior. Se estivermos computando o

próximo estado da onda, precisamos da informação do estado atual e do estado anterior.

Nesse contexto, na função CUDA_SwapArrays, pp e pc são usados para a propagação da onda p, enquanto qp e qc são usados para a propagação da onda q. De forma breve:

1. Inicialmente, **pp** e **qp** contêm o estado **anterior** das ondas **p** e **q**, respectivamente, enquanto pc e qc contêm o estado atual.
2. Quando o kernel é chamado, ele usa **pp**, **pc**, **qp**, e **qc** para calcular o **próximo estado** dos campos **p** e **q**, que ele armazena de volta em **pp** e **qp**.
3. Então a função CUDA_SwapArrays é chamada. Ela troca os ponteiros de pp e pc (e de qp e qc). Isso efetivamente move o estado atual para pp e qp (**para ser usado como estado anterior na próxima iteração**), e o recém-calculado próximo estado para pc e qc (**para ser usado como estado atual na próxima iteração**).

A vantagem dessa técnica é que ela evita a necessidade de alocar um novo buffer e copiar dados para ele a cada passo de tempo. Em vez disso, **os dois buffers são usados alternadamente como entrada e saída, economizando tempo e recursos de memória.**

Conclusão da análise de dependência de dados:

Em termos de dependência de dados, o kernel parece ser altamente paralelizável. O cálculo em cada ponto da grade de dados é independente dos cálculos em todos os outros pontos, o que significa que eles podem ser realizados em paralelo sem a necessidade de sincronização ou comunicação entre as threads. O código assume que cada thread CUDA tem acesso exclusivo a um único ponto na grade de dados, e que nenhuma thread tentará escrever no mesmo ponto simultaneamente.

Além disso, o código pode não funcionar corretamente se as dimensões da grade de dados não forem múltiplos do número de threads por bloco, já que algumas threads podem tentar acessar pontos fora da grade de dados.Quando se lança um kernel, especifica-se o número de blocos de threads na grade e o número de threads por bloco. No programa, os números de threads por bloco são definidos pelas constantes **`BSIZE_X` e `BSIZE_Y`**, e o número de blocos é calculado como **`(sx / threadsPerBlock.x)` e `(sy / threadsPerBlock.y)`.**

Se `sx` e `sy` não forem múltiplos de `BSIZE_X` e `BSIZE_Y`, respectivamente, haverá blocos no final da grade que não estarão completamente preenchidos com dados válidos. No entanto, todas as threads em um bloco ainda serão lançadas e tentarão acessar a memória. **As threads que correspondem a pontos fora da grade de**

dados tentarão acessar memória que está fora dos arrays de dados, o que resultará em um acesso à memória fora dos limites.

Por exemplo, suponha que `sx` é 17 e `BSIZE_X` é 8. Então, a grade terá 3 blocos na direção x. No último bloco, apenas a primeira thread terá um ponto correspondente na grade de dados, mas todas as 8 threads serão lançadas e tentarão acessar a memória. As threads 2 a 8 tentarão acessar memória além do final do array de dados, o que é um erro.

Finalmente, é importante notar que este é um kernel CUDA de uma única etapa. Ou seja, **cada chamada ao kernel avança a solução por um único passo de tempo**. **Para avançar a solução por vários passos de tempo, o kernel precisaria ser chamado várias vezes em sequência, possivelmente com uma sincronização do dispositivo entre as chamadas para garantir a consistência dos dados.**