# Capítulo 2

# Otimização de Programas Paralelos com uso do OpenACC

Evaldo B. Costa – *IC/UFRJ – ebcosta@ic.ufrj.br*
Gabriel P. Silva – *IC/UFRJ – gabriel@ic.ufrj.br*

***Resumo***

*Este minicurso tem por objetivo apresentar técnicas de otimização de programas paralelos que façam uso de diretivas do OpenACC. Para isso, serão utilizadas ferramentas que realizam uma análise completa de desempenho do código para identificação de regiões paralelizáveis e quais métodos podem ser aplicados. O OpenACC é um modelo de programação para computação paralela que pode ser executado em diversos tipos de arquiteturas: multicore, manycore e aceleradores. Assim, neste minicurso são avaliados os efeitos dos componentes de hardware sobre o desempenho de programas paralelos. Ressaltam-se as modificações que devem ser feitas no código para explorar com vantagem as características dos recursos computacionais, avaliando os seus respectivos impactos no desempenho de um programa.*

## 2.1. Arquitetura dos Aceleradores Gráficos

As arquiteturas dos aceleradores gráficos (GPUs) são bem diferencidas das arquiteturas dos processadores convencionais. O paralelismo nos aceleradores gráficos é explorado através de um conjunto maçico de multiprocessadores de fluxo (*streaming multiprocessors (SM)*), executando em paralelo e de forma sincronizada trechos computacionalmente intensivos, chamados de *kernels*, das diversas aplicações.

Para o melhor entedimento dos aceleradores gráficos (GPUs) vamos estudar, sem perda de generalidade, a arquitetura de um tipo de acelerador gráfico desenvolvido pela NVIDIA, a arquitetura Kepler, observada na Figura 2.1.

Na Figura 2.1 verificamos que o acelerador gráfico possui uma arquitetura distinta, com diversos níveis de hierarquia de memória, algumas delas compartilhadas, outras exclusivas de cada processador de fluxo (SM). Analisamos esses e outros detalhes a seguir.


**Figura 2.1:** Arquitetura NVIDIA Kepler [NVIDIA 2014]

**Principais características da arquitetura Kepler** Cada unidade de multiprocessador de fluxo (Figura 2.2) possui 192 núcleos CUDA de precisão simples e 64 de precisão dupla, onde cada núcleo tem unidades lógicas de aritmética inteira de ponto flututante capazes de operar em modo "pipeline", incluindo operações do tipo *fused multiply-add* (FMA). As 32 unidades de função especial (SFU) dentro de cada SM são utilizadas para aproximar operações transcendentais como raiz quadrada, seno, cosseno e recíproco (1/x). O projeto dessa arquitetura está focado no desempenho/consumo energético, fundamental na computação de alto desempenho moderna.

O escalonador do multiprocessador de fluxo (SM) dispara as *threads* em grupos de 32 *threads* chamadas de *warps*. Cada SM possui quatro escalonadores de *warp*, permitindo um máximo de quatro *warps* disparadas e executadas concorrentemente. O número de registradores pode chegar até 255 registradores utilizados simultaneamente por cada *thread*.

Para melhorar ainda mais o desempenho, a arquitetura Kepler apresenta uma instrução de *shuffle* que permite às *threads* dentro de uma mesma *warp* compartilhar dados. Anteriormente, o compartilhamento de dados entre *threads* demandava o acesso à memória compartilhada, com operações de *load* e *store*, impactando em muito o desempenho de aplicações como a transformada de Fourier (FFT).

Outro tipo de instrução disponível são operações atômicas em memória, permitindo que as *threads* realizem adequadamente operações de *read-modify-write*, como soma, máximo, mínimo e compare-e-troque em estruturas de dados compartilhadas. Operações atômicas são amplamente utilizadas para ordenação em paralelo e para o acesso em paralelo a estruturas de dados compartilhadas sem a necessidade de travas que serializam a execução do código.


**Figura 2.2:** Multiprocessador de Fluxo (SM) [NVIDIA 2014]

A arquitetura de memória do acelerador está organizada em diversos níveis, possuindo uma cache L1 para cada SM, além de uma cache apenas de leitura, como visto na Figura 2.3.

A quantidade de memória de cada SM é configurável, por exemplo, a memória local (64 ou 128 KB) pode ser dividida nas seguintes proporções: 75% x 25%, 25% x 75% ou 50% x 50% entre uma memória compartilhada e uma cache L1.

Além da cache L1, a arquitetura Kepler introduz uma cache apenas de leitura de 48KB. O gerenciamento dessa cache pode ser feito automaticamente pelo compilador ou explicitamente pelo programador. O acesso a uma variável ou estrutura de dados que o programador identifica como apenas de leitura, pode ser declarada com a palavra chave *const __restrict*, permitindo ao compilador carregá-la na cache apenas de leitura.

Essa arquitetura possui também um cache de nivel 2 (L2) com 1,5 MB de capacidade. A cache L2 é o ponto primário de unificação de dados entre os diversos SMs, servindo operações de *load, store* e de textura, provendo um compartilhamento de dados eficiente e de alta velocidade.

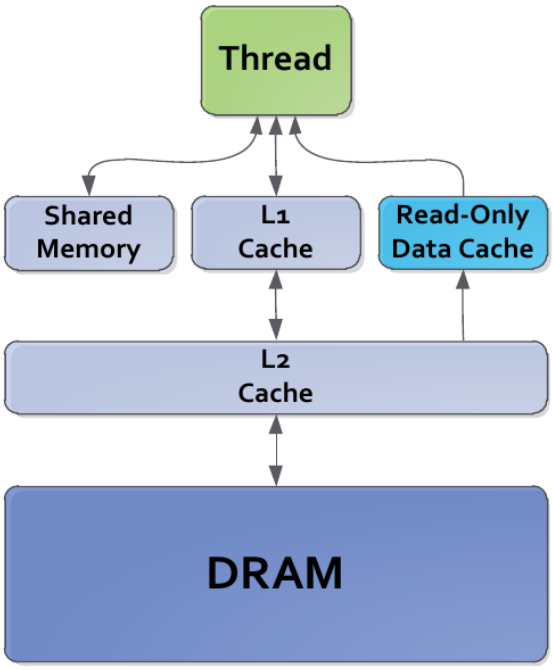



**Figura 2.3:** Hierarquia de Memória CUDA [NVIDIA 2014]

Algoritmos onde o endereço dos dados é conhecido previamente, tais como solucionadores de física, *ray tracing* e multiplicação esparsa de matrizes, se beneficiam especialmente das hierarquia de cache. Os *kernels* de filtro e convolução onde é necessário que diversos SMs leiam os mesmos dados, também se beneficiam dessa hierarquia.

A arquitetura possui uma série de outras facilidades como código de correção de erro, paralelismo dinâmico, gerenciamento de filas de trabalho e unidade de gerencimento de *grids*, que servem para melhorar o desempenho e a confiabilidade do acelerador. Maiores detalhes podem ser vistos na referência [NVIDIA 2014].

A arquitetura dos aceleradores está em constante evolução, na Tabela 2.1 apresentamos as características de alguns aceleradores NVIDIA lançados mais recentemente.

Para finalizar esta seção, gostaríamos de mostrar como se dá a execução das *threads* no acelerador, apresentando o modelo de programação paralela CUDA. O CUDA é uma combinação de plataforma de *hardware/software* que permite aos aceleradores gáficos executar programas escrito em C, C ++, Fortran e outras linguagens.

Um programa CUDA invoca funções paralelas chamadas *kernels*, que são executados em várias *threads* paralelas. O programador ou compilador organiza essas *threads* em blocos de *thread* e grades de blocos de *thread*, conforme mostrado na Figura 2.4.

Cada *thread* dentro de um *bloco de thread* executa uma instância do *kernel*. Cada *thread* também possui IDs de *thread* e de bloco dentro de seu *bloco de thread* e grade, um contador de programa, registradores, memória privada por *thread*, entradas e resultados de saída.

| Características da GPU | NVIDIA Tesla P100 | NVIDIA Tesla V100 | NVIDIA A100 | Kepler |
|---|---|---|---|---|
| GPU Codename | GP100 | GV100 | GA100 | GK110 |
| GPU Architecture | NVIDIA Pascal | NVIDIA Volta | NVIDIA Ampere | NVIDIA Kepler |
| Compute Capability | 6.0 | 7.0 | 8.0 | 3.5 |
| Threads / Warp | 32 | 32 | 32 | 32 |
| Max Warps / SM | 64 | 64 | 64 | 64 |
| Max Threads / SM | 2048 | 2048 | 2048 | 2048 |
| Max Thread Blocks / SM | 32 | 32 | 32 | 16 |
| Max 32-bit Registers / SM | 65536 | 65536 | 65536 | 65536 |
| Max Registers / Block | 65536 | 65536 | 65536 | 65536 |
| Max Registers / Thread | 255 | 255 | 255 | 255 |
| Max Thread Block Size | 1024 | 1024 | 1024 | 1024 |
| FP32 Cores / SM | 64 | 64 | 64 | 192 |
| Ratio of SM Registers to FP32 Cores | 1024 | 1024 | 1024 | 341 |
| Shared Memory Size / SM | 64 KB | up to 96 KB | up to 48 KB | up to 48 KB |

**Tabela 2.1:** Comparação entre Arquiteturas NVIDIA

Um *bloco de thread* é um conjunto de *threads* em execução simultânea cooperando entre si por meio de sincronização de barreira e memória compartilhada. Um *bloco de thread* possui um ID de bloco em sua grade. Uma grade é uma matriz de *blocos de thread* que executam o mesmo *kernel*, leem as entradas da memória global, gravam os resultados para a memória global e fazem a sincronização entre chamadas de *kernel* dependentes.

No modelo de programação paralela CUDA, cada *thread* tem um espaço de memória privado por *thread* usado para salvamento de registradores, chamadas de função e variáveis

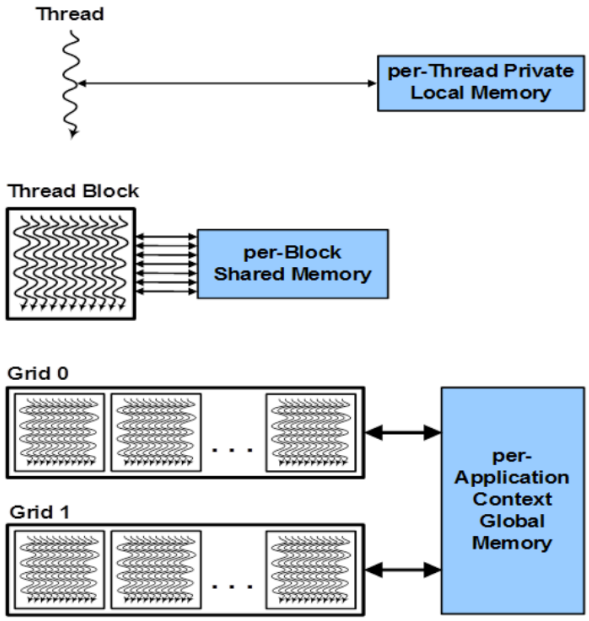



**Figura 2.4:** Modelo de programação CUDA [NVIDIA 2014]

de arranjo automáticas em "C". Cada *bloco de thread* tem um espaço de memória compartilhado por bloco usado para comunicação inter-thread, compartilhamento de dados e compartilhamento de resultados em algoritmos paralelos. Grades de *blocos de thread* compartilham resultados em um espaço de memória global após a sincronização global em todo o *kernel*.

A hierarquia de *threads* do CUDA é mapeada para uma hierarquia de processadores na GPU; uma GPU executa uma ou mais grades de *kernel*; um multiprocessador de streaming (SM) executa um ou mais *blocos de threads*; e os núcleos CUDA e outras unidades de execução de instruções no SM executam as *threads*.

O SM executa threads em grupos de 32 *threads* chamados *warps*. Embora os programadores possam ignorar os *warps* para uma execução funcionalmente correta, o desempenho das aplicações pode ser muito melhorado, mantendo-se as *threads* de um mesmo *warp* executando o mesmo código e realizando acessos de memória com endereços próximos.

## 2.2. Memória Unificada (Unified Memory)

As memórias da CPU e GPU são normalmente separadas, ou seja, as informações armazenadas em seus sistemas de memórias não são compartilhadas. A movimentação de dados entre os dois sistemas de memórias é realizada sempre que existe a necessidade de processamento de alguma informação.

Uma forma de solucionar esse problema é o uso de memória unificada. A memória unificada é um espaço de endereçamento único acessível tanto pela CPU como pela GPU. Com o uso dessa tecnologia de *hardware/software*, as aplicações alocam dados que podem ser lidos e escritos tanto pelas CPUs como pelas GPUS (Figura 2.5).

Quando o código em execução em uma CPU ou GPU acessa dados alocados dessa maneira (*CUDA managed memory*), o sistema de *software* CUDA e/ou *hardware* se encarrega de migrar as páginas para a memória do processador que estiver acessando esses dados.

Destacamos que a arquitetura de GPU Pascal é a primeira com suporte de *hardware* para falha e migração de página de memória virtual, por meio de seu mecanismo de migração de página. As GPUs mais antigas baseadas nas arquiteturas Kepler e Maxwell também suportam uma forma mais limitada de memória unificada.

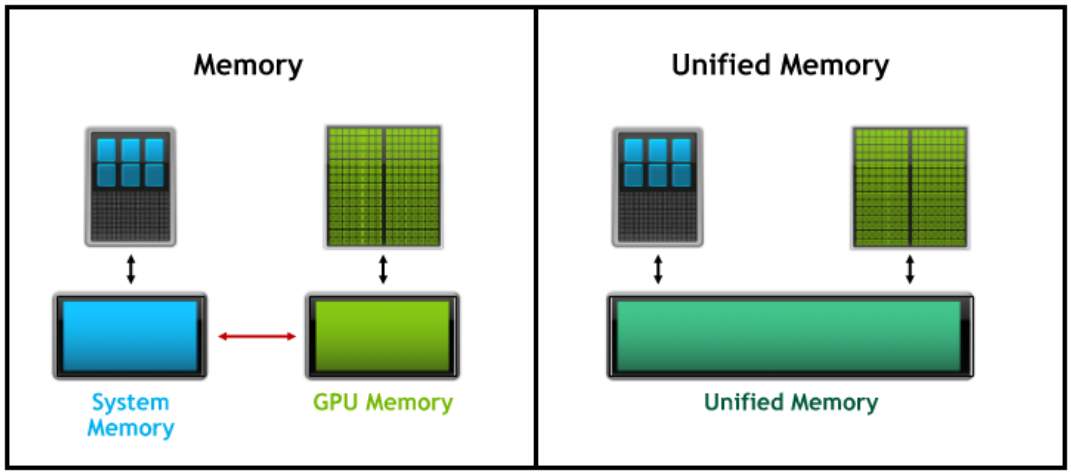



**Figura 2.5:** Modelo de memória tradicional e memória unificada [Harris 2017]

A alocação de memória unificada pode ser feita explicitamente em CUDA, substituindo-se as chamadas para a função **malloc()** por chamadas à função **cudaMallocManaged()**, uma função de alocação de memória que retorna um ponteiro acessível por qualquer processador.

Já no OpenACC essa substituição das chamadas é feita automaticamente pelo compilador. Para o uso da Memória Unificada no OpenACC com o compilador PGI, basta utilizar usar a opção *-ta=tesla:managed* na compilação do código do programa. Portanto, as cláusulas e diretivas de dados OpenACC não são necessárias para dados "gerenciados". Elas são essencialmente ignoradas e, de fato, podem ser omitidas.

O uso de memória gerenciada se aplica apenas aos dados alocados dinamicamente. Dados estáticos (variáveis externas e estáticas em C, módulos em Fortran, blocos comuns e variáveis salvas) e dados locais das funções são ainda manipulados pelo ambiente de execução do OpenACC.

Embora o uso da memória unificada ofereça uma grande simplificação da programação, o desempenho final é variável em função de cada aplicação e deve ser avaliado com cuidado antes de sua utilização em produção [Harris 2017].

## 2.3. Conceitos de gang, worker e vector

Como o OpenACC serve como uma linguagem para aceleradores genéricos existem três níveis de paralelismo disponíveis no OpenACC. Eles especificam o nível de paralelismo

contidos na rotina, são chamados de *gangue, trabalhador e vetor*. Uma *gangue* é composta por um ou vários *trabalhadores* e corresponde a um *bloco de threads* em CUDA. Todos os *trabalhadores* de uma *gangue* podem compartilhar os mesmos recursos, como memória cache ou processador.

Em OpenACC, um *trabalhador* é um grupo de *vetores*. A sua dimensão vertical é igual ao número de *trabalhadores* e a dimensão horizontal é o tamanho do *vetor*. A dimensão do *trabalhador* se estende até a altura da gangue (*bloco de threads*). Cada *vetor* OpenACC (um elemento do arranjo iterado) é uma *thread* CUDA. A dimensão do *vetor* se dá ao longo da largura do *bloco de threads*.

Cada *trabalhador* corresponde a um número de *threads* igual ao tamanho do vetor. Então um *trabalhador* corresponde a uma *warp* em CUDA apenas se o *vetor* tiver um comprimento igual a 32; já que um *trabalhador* não corresponde necessariamente a uma *warp*. Por exemplo, um *trabalhador* pode corresponder a duas *warps* se o *vetor* tiver tamanho 64. A característica principal de uma *warp* é que todas as suas *threads* executam concorrentemente. Uma *grade* CUDA é composta de vários *blocos de threads* ou gangues do OpenACC, os quais podem ser organizados em uma ou duas dimensões.

Os aceleradores podem ter limitações quanto aos valores que podem ser atribuídos a esses particionamentos. Por exemplo, para GPUS da NVIDIA, as seguintes limitações existem:

- O comprimento de um *vetor* deve ser um múltiplo de 32 (até 1024)
- O tamanho de uma *gangue* é dado pelo número de *trabalhadores* vezes o tamanho de um *vetor*, não podendo ser maior que 1024.

As diretivas do OpenACC que especificam nível de paralelismo são **gang, worker e vector**, respectivamente para os níveis *gangue, trabalhador e vetor*. Essas diretivas também podem ser combinadas em um laço específico. Por exemplo, um laço **gang vector** pode ser particionado entre *gangues*, cada uma delas com 1 *trabalhador* implicitamente, e depois vetorizado.

A especificação OpenACC reforça que o laço mais externo deve ser um laço de uma **gang**, o laço paralelo mais interno deve ser um laço **vector** e um laço **worker** pode aparecer no meio. Um laço seqüencial (**seq**) pode aparecer em qualquer nível.

O uso dos níveis de paralelismo são aplicados na diretiva **parallel loop** para gerar maior ganho na execução do laço. Também podem ser usadas da diretiva **kernels**.

```
#pragma acc parallel loop gang
for(i = 0 ; i < size; i++)
  #pragma acc loop worker
  for(j = 0 ; j < size; j++)
    #pragma acc loop vector
    for(k = 0 ; k < size; k++)
       c[i][j]+=a[i][k]*b[k][j];
```

**Exemplo 2.1:** Cláusulas da diretiva parallel loop

Adicionalmente, o programador pode definir esses parâmetros dentro de uma região **parallel** ou **kernels** com o uso das cláusulas **num_gangs(N), num_workers(M), vector_length(Q)**. Esses níveis serão empregados para todos os *kernels* disparados dentro da região.

**#pragma acc parallel num_gangs(expr-inteira)**

**#pragma acc parallel num_workers(expr-inteira)**

**#pragma acc parallel vector_length(expr-inteira)**

## 2.4. Movimentação de dados

Um grande fator de impacto de desempenho no processamento paralelo é a movimentação de dados, principalmente quando se faz o processamento dos dados em lugares diferentes. Quando se usa processamento em aceleradores nem sempre é possível carregar todos os dados para o acelerador, isso ocorre em geral porque a memória da CPU é maior a dos aceleradores, embora os aceleradores tenham maior largura de banda (Figura 2.6).

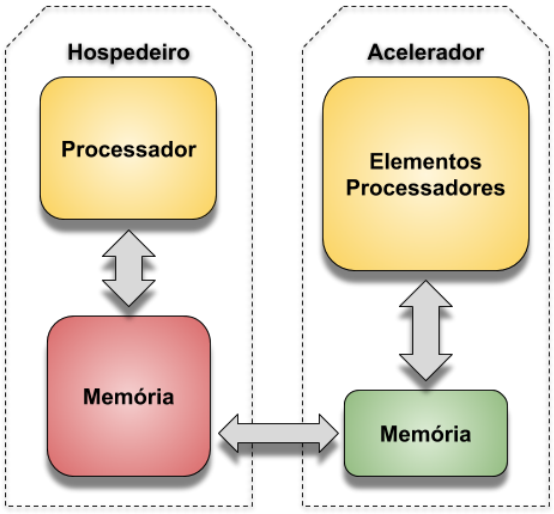



**Figura 2.6:** Modelo básico de movimentação de dados entre hospedeiro e o acelerador [Chen 2017]

A movimentação de dados entre o hospedeiro e o acelerador é feita através do barramento, que é lento em comparação com largura de banda de memória. Por sua vez o acelerador não pode executar o processamento dos dados até que eles estejam na sua memória local.

Para realizar a movimentação de dados entre o hospedeiro e o acelerador durante a execução do programa é necessário o uso das cláusulas de dados. As cláusulas de movimentação de dados podem ser usadas nas diretivas **data**, **kernels** ou **parallel**.

| Cláusula | Descrição |
|---|---|
| **copy** | Cria espaço para as variáveis listadas no dispositivo, inicia as variáveis copiando dados para o dispositivo no início da região, copia os resultados de volta para o hospedeiro no final da região e finalmente libera o espaço no dispositivo quando terminar. |
| **copyin** | Cria espaço para as variáveis listadas no dispositivo, inicia a variável copiando os dados para o dispositivo no início da região e libera o espaço no dispositivo quando terminar, sem copiar os dados de volta para o hospedeiro. |
| **copyout** | Cria espaço para as variáveis listadas no dispositivo, mas não as inicia. No final da região, copia os resultados de volta para o hospedeiro e libera o espaço no dispositivo. |
| **create** | cria espaço para as variáveis listadas e as libera no final da região, mas não copia nenhum dos dados de/para o dispositivo. |
| **present** | As variáveis listadas já estão presentes no dispositivo, portanto, nenhuma outra ação precisa ser executada. Isso é usado com mais frequência quando existe uma região de dados em uma rotina de maior nível. |
| **deviceptr** | As variáveis listadas usam a memória do dispositivo que foi gerenciada fora do OpenACC, portanto as variáveis devem ser usadas no dispositivo sem qualquer conversão de endereço. Esta cláusula é geralmente usada quando o OpenACC é misturado com outro modelo de programação. |

**Tabela 2.2:** Cláusulas da Diretiva Data

**#pragma acc data [cláusula]**

## 2.5. Dicas para a paralelização de laços

A paralelização de estruturas iterativas pode disparar avisos de compilação e, às vezes, é necessário reexpressar o código do laço. Por exemplo, se o programador usa uma diretiva como **kernels ou parallel**, e se o compilador vê alguma dependência entre os laços, o compilador não paralelizará esse trecho do código. Expressando a mesma iteração de uma maneira diferente, pode ser possível evitar os avisos de compilação e fazer com que o compilador execute os laços em paralelo. Os exemplos de código abaixo ilustram laços que geram mensagens de erro do compilador.

```
#pragma acc kernels
{
   while (i < N && found == -1) {
      if (A[i] >= 102.0f) {
         found = i;
      }
      ++i;
   }
}
```

Compilando o código acima os seguintes avisos são gerados pelo compilador:

```
Accelerator restriction: loop has multiple exits
Accelerator region ignored
```

O problema aqui é que "i" poderia assumir valores diferentes quando o *loop while* é encerrado, dependendo se uma *thread* em execução encontra um valor de A [i] maior ou igual a 102,0. O valor de "i" vai variar de execução para execução e não produzirá o resultado que o programador pretendia.

Re-expressando o código com o laço "for" a seguir, com uma lógica de desvio, o compilador agora reconhece o primeiro laço como sendo paralelizável.

```
#pragma acc kernels
{
   for (i=0; i<N; ++i) {
      if (A[i] >= 102.0f) {
         found[i] = i;
      }
      else {
         found[i] = -1;
      }
   }
}
i=0;
while (i < N && found[i] < 0) {
   ++i;
}
```

Embora esse código seja um pouco maior, com dois laços, acelerar o primeiro laço compensa a separação de um laço em dois. Normalmente, separar um laço em dois é ruim para o desempenho, mas nesse caso ao expressaros laços paralelos temos um ganho de desempenho [Murphy 2016].

Uma coisa importante a ser observada sobre a construção **kernels** é que o compilador analisará o código e apenas paralelizará quando estiver certo de que é seguro fazê-lo. Em alguns casos, o compilador pode não ter informações suficientes em tempo de compilação para determinar se um laço é seguro para ser paralelizado; nesse caso, isso não será feito, mesmo que o programador possa ver claramente que o laço pode ser paralelizado com segurança.

Por exemplo, no caso do código C/C ++, em que as matrizes são passadas para as funções como ponteiros, o compilador nem sempre pode ser capaz de determinar que duas matrizes não compartilham a mesma área de memória, também conhecido como *aliasing* de ponteiros. Se o compilador não puder saber que os dois ponteiros não possuem *alias*, não será capaz de paralelizar um laço que acessa essas matrizes.

Uma prática recomendada para os programadores em C é usar a palavra-chave *restrict* (ou o decorador *__restrict* em C ++) sempre que possível, para informar ao compilador que os ponteiros não têm *alias*, o que frequentemente fornecerá ao compilador informações

suficientes para paralelizar laços que não o seriam de outra forma. Além da palavra-chave *restrict*, declarar variáveis constantes usando a palavra-chave *const* pode permitir que o compilador use memória apenas de leitura para essa variável, se essa memória existir no acelerador.

O uso de *const* e *restrict* é uma boa prática de programação em geral, pois fornece ao compilador informações adicionais que podem ser usadas na otimização do código. Um benefício adicional que a construção **kernels** fornece é que, se os dados forem movidos para o dispositivo para uso em laços contidos na região, esses dados permanecerão no dispositivo por toda a extensão da região ou até que sejam necessários novamente no hospedeiro dessa região. Isso significa que, se vários laços acessarem os mesmos dados, eles apenas serão copiados uma vez para o acelerador. Quando o laço paralelo é usado em dois laços subsequentes que acessam os mesmos dados, o compilador pode ou não copiar os dados entre o hospedeiro e o dispositivo entre os dois laços, o que pode resultar em menor movimentação de dados por padrão.

## 2.6. Diretivas e Cláusulas Avançadas

### 2.6.1. Cláusula collapse

A execução de um laço em OpenACC está associada ao laço imediatamente a seguir. Uma diretiva é necessária para cada laço. Isso tende a ser complicado, especialmente se vários laços devem ser tratados da mesma maneira. A cláusula **collapse** é útil nesse caso. O argumento para a cláusula **collapse** é um número inteiro positivo constante, que especifica quantos laços fortemente aninhados serão associados para a criar um novo laço.

Quais as vantagens em usar a cláusula **collapse**?

- colapsar os laços externos para permitir a criação de mais *gangs*.
- colapsar os laços internos para permitir comprimentos de vetor mais longos.
- colapsar todos os laços, quando for possível, para fazer as duas coisas: ter mais *gangs* criadas e vetores maiores.

Esta cláusula é especialmente útil quando alguns laços não tem um número total de iterações suficientemente grande para fazer uso efetivo do acelerador. A sua sintaxe é vista a seguir:

**#pragma acc loop collapse(n)**

O exemplo a seguir apresenta um trecho de código com um laço com o uso desta cláusula, seguido de um laço que exemplifica o efeito do seu uso.

```
#pragma acc parallel loop collapse(2)
    for (int i = 0; i < N; i++)
         for (int j = 0; j < M; j++)

#pragma acc parallel loop
    for (int ij = 0; ij < N*M; ij++)...
```

### 2.6.2. Diretiva Routine

As chamadas de função ou sub-rotina em laços paralelos podem ser problemáticas para os compiladores, pois nem sempre é possível para o compilador ver todos os laços de uma só vez. Os compiladores OpenACC 1.0 eram forçados a fazer *inline* de todas as rotinas chamadas em regiões paralelas ou a não paralelizar laços contendo chamadas de rotina.

O OpenACC 2.0 introduziu a diretiva **routine**, que instrui o compilador a criar uma versão de dispositivo da função ou sub-rotina para que possa ser chamada de uma região de dispositivo. Para leitores já familiarizados com a programação CUDA, essa funcionalidade é semelhante ao especificador da função *__device__*.

Para orientar a otimização, você pode usar cláusulas para informar ao compilador se a rotina deve ser criada para paralelismo de nível de **gang, work, vector ou seq** (sequencial). Você pode especificar várias cláusulas para rotinas que podem ser chamadas com vários níveis de paralelismo.

Fazer isso corretamente exige que você coloque uma cláusula **routine** apropriada antes da definição da rotina para chamar a rotina com o nível certo de paralelismo.

```
#pragma acc routine vector
void foo(float* v, int i, int n) {
  #pragma acc loop vector
  for ( int j=0; j<n; ++j) {
    v[i*n+j] = 1.0f/(i*j);
  }
}

#pragma acc parallel loop
for ( int i=0; i<n; ++i) {
  foo(v,i);
  //chamada no dispositivo
}
```

**Exemplo 2.2:** Diretiva Routine

Quando a rotina *foo* é chamada a partir do código do hospedeiro, ele será executado no hospedeiro, incrementando os valores do hospedeiro. Quando chamado de dentro de uma construção paralela do OpenACC, ela incrementará os valores do dispositivo.

Teoricamente esta diretiva permitira o uso de funções recursivas, contudo há alguns fatores que que limitam a profundidade da recursão. Por exemplo, os dispositivos NVIDIA estão limitados a 16 níveis de recursão, assim como dispositivos AMD possuem outros

limites.

Nota: a partir da versão 14.9 do compilador PGI, uma diretiva **routine** sem nenhuma cláusula de nível de paralelismo (**gang, worker ou vector**) será tratada como se uma cláusula **seq** estivesse presente.

### 2.6.3. Operações Atômicas

Quando uma ou mais iterações de um laço precisam acessar um elemento na memória ao mesmo tempo, condições de corrida podem ocorrer. Por exemplo, se uma iteração do laço está modificando o valor contido em uma variável e outra está tentando ler a mesma variável em paralelo, diferentes resultados podem ocorrer dependendo de qual iteração ocorra primeiro.

Em programas seriais, os laços sequenciais garantem que a variável será modificada e lida em uma ordem previsível, mas os programas paralelos não garantem que uma iteração específica de um laço irá ocorrer antes da outra. Em casos simples, como encontrar uma soma, valor máximo ou mínimo, uma operação de redução irá garantir que o programa será executado corretamente.

Para operações mais complexas, a diretiva **atomic** garantirá que não haverá duas *threads* executando a operação nela contida simultaneamente. O uso da diretiva **atomic** às vezes é uma parte necessária do processo de paralelização para garantir a correção do código.

A diretiva **atomic** aceita uma das quatro cláusulas seguintes para declarar o tipo de operação contida na região:

- A operação **read** assegura que duas iterações de um laço não farão leituras da região ao mesmo tempo;
- A operação **write** garantirá que não haja duas iterações realizando escrita na região ao mesmo tempo;
- Uma operação **update** é uma operação de leitura e de escrita combinadas;
- Finalmente, uma operação **capture** executa uma atualização, mas salva o valor calculado nessa região para ser utilizada no código seguinte à região.

Se nenhuma cláusula for definida, uma operação **update** é assumida.

Um histograma é uma técnica comum para contar quantas vezes os valores ocorrem em um conjunto de entrada de acordo com o seu valor. O código do exemplo abaixo percorre uma série de números inteiros de um intervalo conhecido e conta o total de ocorrências de cada número nesse intervalo. Como cada número no intervalo pode ocorrer várias vezes, precisamos garantir que cada elemento no vetor de histograma seja atualizado atomicamente. O código abaixo demonstra usando a diretiva **atomic** para gerar um histograma.

```
#pragma acc parallel loop
    for(int i=0; i < HN; i++)
        h[i]=0;
#pragma acc parallel loop
    for(int i=0; i < N; i++) {
#pragma acc atomic update
        h[a[i]]+=1; }
```

**Exemplo 2.3:** Diretiva atomic

Observe que as atualizações no vetor do histograma *h* são executadas atomicamente. Como estamos incrementando o valor do elemento de um vetor, uma operação **update** é usada para ler o valor, modificá-lo e gravá-lo novamente.

### 2.6.4. Cláusula tile

É a adição da cláusula **tile** à diretiva **acc loop**. Com a cláusula **tile** é possível otimizar o laço através da operação de blocos menores para explorar o acesso aos dados. Considere o seguinte exemplo de transposição de matriz.

```
#pragma acc parallel loop private(i,j) tile(8,8)
for(i=0; i<rows; i++)
    {
     for(j=0; j<cols; j++)
      {
        out[i*rows + j] = in[j*cols + i];
      }
    }
```

**Exemplo 2.4:** Cláusula tile

Ao adicionar a cláusula tile (8,8) ao laço paralelo, serão criados automaticamente pelo compilador dois laços adicionais que funcionam em um *chunk* 8x8 (tile) da matriz antes de passar para o próximo *chunk*. Com isso o compilador faz a otimização dentro do bloco, com o objetivo de obter melhor desempenho. Embora uma transposição de matriz não tenha muita reutilização de dados, outros algoritmos podem ter uma melhora significativa no desempenho, explorando a localidade e a reutilização de dados nos laços disponíveis.

## 2.7. Cláusulas device_type e vector_length

O OpenACC permite que os programadores consigam otimizar suas diretrizes para aceleradores específicos com o uso da cláusula **device_type**, com isso é possível obter melhores desempenhos. Com o OpenACC 1.0, diretivas de pré-processador eram necessárias para ajustar as diretivas para uso em aceleradores específicos. Além de dificultar a manutenção do código, devido à duplicação de diretivas, isso significa que é impossível oferecer suporte a vários tipos de dispositivos no mesmo executável. Já com a versão do OpenACC 2.0 ele permite que determinadas cláusulas sejam fornecidas especificamente para determinadas arquiteturas.

Apenas as cláusulas **async, wait, num_gangs, num_workers e vector_length** podem

aparecer em seguida a uma cláusula **device_type**.

**#pragma acc loop device_type(lista-tipo-dispositivo)**

Um exemplo de cláusula que pode ser utilizada em associação com a cláusula **device_type** é a **vector_length**, que é utilizada para especificar o tamanho do vetor que será usado em um laço paralelo (com a diretiva **parallel loop**).

No Exemplo 2.5 é especificado um comprimento difernte de vetor, dependendo do tipo de acelerador que será usado. Nesse exemplo, se o laço for utilizar um acelerador **NVIDIA**, o compilador utilizará um comprimento vetorial de 256; se for utilizar um acelerador **Radeon**, o compilador usa um comprimento de vetor de 512; e para qualquer outro acelerador que não seja especificado será usado comprimento vetorial de 64. Ambas as cláusulas podem ser utilizadas em conjunto com as demais cláusulas do OpenACC.

```
#pragma acc parallel loop \
        device_type(nvidia) vector_length(256) \
        device_type(radeon) vector_length(512) \
                            vector_length(64)
    for (int i = 0; i < n; ++i) {
         c[i] = a[i] + b[i];
    }
```

**Exemplo 2.5:** Cláusula tile

## 2.8. PGI Profiler

O PGI Profiler é uma ferramenta utilizada para analisar o desempenho de programas paralelos escritos com OpenMP, MPI, OpenACC ou CUDA. O PGI Profiler permite a visualização e otimização do desempenho de uma aplicação através da análise da linha do tempo de execução da aplicação. Com isso é possível identificar regiões de gargalos que podem ser otimizadas, eliminando ou reduzindo esses gargalos para alcançar um melhor desempenho.

Para iniciar a análise da execução de uma aplicação é utilizamos primeiro o comando **pgprof**. Esse comando gera um arquivo de saída com as informações de uso dos recursos computacionais em todos os trechos da aplicação. Não é necessário nenhum tipo de alteração no código para criar o arquivo de saída, entretanto, existem alguns parâmetros do compilador PGI que podem ser usados para coletar mais informações de uso dos recursos. Veja mais detalhes na referência [PGI 2019].

Após a execução do comando **pgprof**, a análise da aplicação pode ser realizada com o uso de comandos em modo terminal ou gráfico. Para executar no modo terminal, use o comando:

**# pgprof [parametro] [aplicação]**

Em modo gráfico executar somente o comando:

**# pgprof**

## 2.9. Exemplos

Após a introdução dos conceitos e das principais diretivas que serão usadas neste minicurso, veremos alguns exemplos de aplicações dessas diretivas, como se comportam e quais as melhores opções do seu uso para melhorar o desempenho.

### 2.9.1. Cálculo de Pi

Iniciaremos com o exemplo o cálculo do número Pi. Em computação existem diversos algoritmos que podem ser utilizados para o cálculo aproximado do Pi. No Exemplo 2.6 é apresentada uma implementação do cálculo de Pi utilizando uma integral cujo resultado é aproximado com uso do método do trapézio.

```
#include <stdio.h>
#define N 1000000000
int main(int argc, char *argv[]) { /* calcpi_seq.c  */
double pi = 0.0f;
long i;
    for (i = 0; i < N; i++) {
         double t = (double) ((i+0.5)/N);
         pi += 4.0/(1.0+t*t);
    }
    printf("pi = %f\n",pi/N);
    return(0);
}
```

**Exemplo 2.6:** Cálculo de Pi sequencial

Para execução do código em paralelo no acelerador podem ser utilizadas as diretivas **kernels ou parallel** do OpenACC. Para o uso da diretiva **kernels**, adicionaremos a linha:

**#pragma acc kernels**.

```
    #pragma acc kernels
    for (long i = 0; i < N; i++)
    {
       double t=(double) ((i+0.5)/N);
       pi += 4.0/(1.0+t*t);
    }
```

Para o uso da diretiva parallel adicionaremos, sempre antes do laço, a linha:

**#pragma acc parallel loop reduction(+: pi)**

```
    #pragma acc parallel loop reduction(+: pi)
    for (long i = 0; i < N; i++)
    {
       double t=(double) ((i+0.5)/N);
       pi += 4.0/(1.0+t*t);
    }
```

Durante a compilação do código é possível observar o resultado do uso dessas diretivas (Figura 2.7).

```
$  pgcc -acc -ta=nvidia -Minfo=all piacc_kernels.c -o piacc_kernels

main:
     11, Loop is parallelizable
         Generating Tesla code
         11, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
        14, Generating implicit reduction(+:pi)
$
$  pgcc -acc -ta=nvidia -Minfo=all piacc_parallel.c -o piacc_parallel

main:
     10, Generating Tesla code
         11, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(+:pi)
     10, Generating implicit copy(pi) [if not already present]
$
```

**Figura 2.7:** Saída do compilador PGI

```
$  pgcc -acc -ta=nvidia -Minfo=all piacc_kernels.c -o piacc_kernels
main:
     10, Generating copy(pi) [if not already present]
     13, Loop is parallelizable
         Generating Tesla code
         13, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         16, Generating implicit reduction(+:pi)
$
$  pgcc -acc -ta=nvidia -Minfo=all piacc_parallel.c -o piacc_parallel

main:
     10, Generating copy(pi) [if not already present]
     12, Generating Tesla code
         13, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(+:pi)
$
```

**Figura 2.8:** Saída do compilador PGI usando movimentação de dados

Na diretiva **kernels** a redução sempre é feita de forma implícita pelo compilador, mas a movimentação dos dados para o acelerador não ocorre. Por sua vez, com uso da diretiva **parallel**, a redução é realizada de forma explícita e a movimentação dos dados é feita de forma implícita pelo compilador.

É possível fazer a movimentação de dados explicitamente para o acelerador usando a diretiva **data**. Deve-se então adicionar a linha **#pragma acc data copy(pi)** antes do laço, de modo que seja feita a cópia da variável *pi* para a memória do acelerador. Desse modo, todo o cálculo da variável *pi* é realizado no acelerador (Figura 2.8).

Em alguns casos, o uso de movimentação de dados do hospedeiro para o acelerador pode ser mais lento por conta custo computacional. Neste exemplo, o tempo gasto em movimentar os dados para o acelerador não compensa devido ao baixo custo computacional do

*kernel*, sendo mais eficiente realizar esses cálculos no hospedeiro. Na Figura 2.9 e Figura 2.10 os tempos de execução usando a movimentação de dados são maiores tanto com a diretiva **kernels** como com a diretiva **parallel**.

```
$ ./piacc_kernels
O valor de pi é: 3.141593
O tempo de execução é 0.905411 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/piacc_kernels.c
  main  NVIDIA  devicenum=0
    time(us): 48
    10: compute region reached 1 time
        10: data copyin transfers: 1
             device time(us): total=25 max=25 min=25 avg=25
        11: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=644,773 max=644,773 min=644,773 avg=644,773
        11: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=121 max=121 min=121 avg=121
        11: data copyout transfers: 1
             device time(us): total=23 max=23 min=23 avg=23
$
$ ./piacc_kernelsdm
O valor de pi é: 3.141593
O tempo de execução é 0.945265 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/piacc_kernelsdm.c
  main  NVIDIA  devicenum=0
    time(us): 71
    10: data region reached 2 times
        10: data copyin transfers: 1
             device time(us): total=16 max=16 min=16 avg=16
        19: data copyout transfers: 1
             device time(us): total=55 max=55 min=55 avg=55
    12: compute region reached 1 time
        13: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=646,115 max=646,115 min=646,115 avg=646,115
        13: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=122 max=122 min=122 avg=122
$
```

**Figura 2.9:** Tempo de execução usando movimentação de dados com a diretiva kernels

```
$ ./piacc_parallel
O valor de pi é: 3.141593
O tempo de execução é 0.943007 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/piacc_parallel.c
  main  NVIDIA  devicenum=0
    time(us): 70
    10: compute region reached 1 time
        10: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=644,827 max=644,827 min=644,827 avg=644,827
        10: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=124 max=124 min=124 avg=124
    10: data region reached 2 times
        10: data copyin transfers: 1
             device time(us): total=16 max=16 min=16 avg=16
        16: data copyout transfers: 1
             device time(us): total=54 max=54 min=54 avg=54
$
$ ./piacc_paralleldm
O valor de pi é: 3.141593
O tempo de execução é 0.943742 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/piacc_paralleldm.c
  main  NVIDIA  devicenum=0
    time(us): 66
    10: data region reached 2 times
        10: data copyin transfers: 1
             device time(us): total=16 max=16 min=16 avg=16
        19: data copyout transfers: 1
             device time(us): total=50 max=50 min=50 avg=50
    12: compute region reached 1 time
        12: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=644,876 max=644,876 min=644,876 avg=644,876
        12: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=122 max=122 min=122 avg=122
$
```

**Figura 2.10:** Tempo de execução usando movimentação de dados com a diretiva parallel

Para medir os tempos de execução dos diferentes códigos, variou-se o valor de $N$. Os valores de $N$ usados para as medições foram entre $1,0 \times 10^9$ a $1,5 \times 10^{10}$. Na Tabela 2.3, são apresentados os resultados obtidos para o tempo de processamento.

| Tamanho de N | kernels | kernels+data | parallel | parallel+data |
|---|---|---|---|---|
| $1,0 \times 10^9$ | 0,305284 | 0,344658 | 0,343664 | 0,343403 |
| $2,0 \times 10^9$ | 0,346696 | 0,384993 | 0,384135 | 0,385037 |
| $1,0 \times 10^{10}$ | 0,688010 | 0,723660 | 0,723909 | 0,728856 |
| $1,5 \times 10^{10}$ | 0,905921 | 0,943896 | 0,944225 | 0,944746 |

**Tabela 2.3:** Uso de recursos do acelerador

Nas Figura 2.11 e Figura 2.12, são apresentadas as análises do comportamento do cálculo

de Pi com uso do **pgprof** com o uso de ambas diretivas, com movimentação de dados (diretiva **data**). Como visto anteriormente, o comportamento é o mesmo independente da diretiva utilizada.

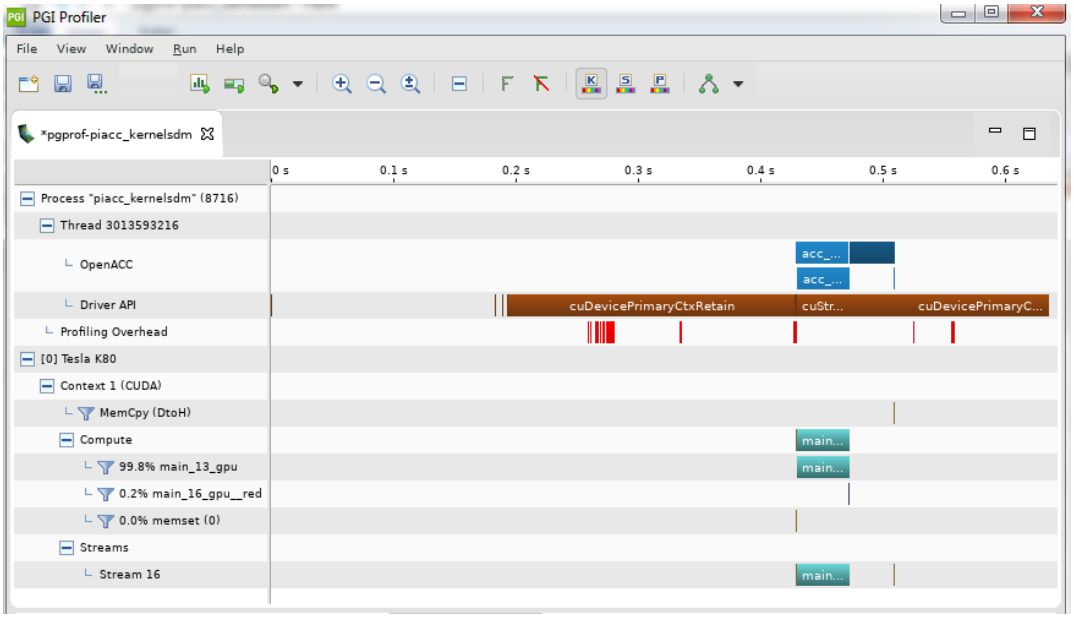



**Figura 2.11:** Análise de execução do código usando a diretiva kernels com movimentação de dados

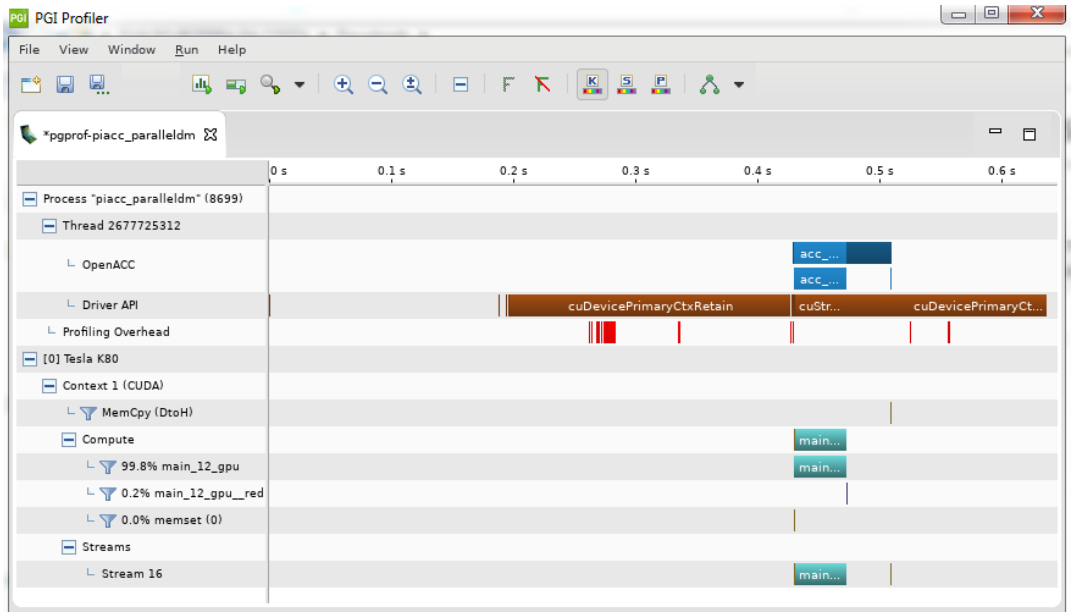
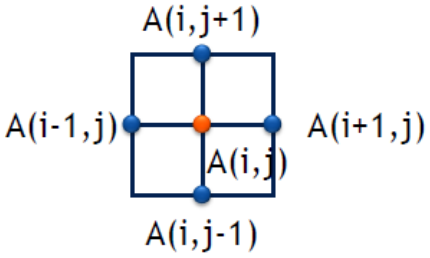



**Figura 2.12:** Análise de execução do código usando a diretiva parallel com movimentação de dados

### 2.9.2. Método Jacobi

O Método de Jacobi é um procedimento iterativo para a resolução de sistemas lineares. Converge iterativamente para o valor correto, calculando novos valores em cada ponto a partir da média dos pontos vizinhos. Neste exemplo faremos o cálculo da temperatura na placa usando a equação de Laplace: $\nabla^2 f(x,y) = 0$.




$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define COLUMNS    1000
#define ROWS       1000
#define MAX_TEMP_ERROR 0.01
double Anew[ROWS+2][COLUMNS+2], A[ROWS+2][COLUMNS+2];
void initialize();
int main(int argc, char *argv[]) {
    int i, j, iteration=1, max_iterations=1000;
    double dt=100;
    initialize();
    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {
        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                Anew[i][j] = 0.25 * (A[i+1][j] +
                A[i-1][j] + A[i][j+1] + A[i][j-1]);
            }
        }
        dt = 0.0;
        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
              dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
              A[i][j] = Anew[i][j];
            }
        }
        iteration++;
    }
    printf("\n Erro maximo na iteracao %d era %f\n",
           iteration-1, dt);
}
```

**Exemplo 2.7:** Método de Jacobi Sequencial

Como pode ser visto no código sequencial (Exemplo 2.7), o primeiro laço dentro do *while* de convergência calcula o novo valor para cada elemento com base nos valores atuais de seus vizinhos, cujo resultado é armazenado em uma matriz temporária **Anew**. Isso garante que todos os valores sejam calculados usando o estado atual de **A** antes que o conteúdo de **A** seja novamente atualizado. Como resultado, cada iteração do laço é completamente independente da outra.

Esse laço também calcula um máximo valor de erro. O valor do erro é a máxima diferença de temperatura entre o novo valor e o antigo. Se o erro entre duas iterações estiver dentro de alguma tolerância, o problema será considerado como convergido e o laço externo será encerrado. O segundo laço simplesmente atualiza o valor de **A** com os valores calculados em **Anew**.

Para que o cálculo seja executado pelo acelerador usaremos a diretiva **parallel** do OpenACC. Adicionar a linha antes do primeiro laço:

**#pragma acc parallel loop**

E a linha:

**#pragma acc parallel loop reduction(max:dt)**

Antes do segundo laço, como visto a seguir.

```
        #pragma acc parallel loop
        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                Anew[i][j] = 0.25 * (A[i+1][j] + A[i-1][j] +
                                     A[i][j+1] + A[i][j-1]);
            }
        }
        dt = 0.0;
        #pragma acc parallel loop reduction(max:dt)
        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
              dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
              A[i][j] = Anew[i][j];
            }
        }
        iteration++;
    }
    printf("\n Erro maximo na iteracao %d era %f\n",
           iteration-1, dt);
}
```

Na Figura 2.13 vemos a saída do compilador PGI. A movimentação dos dados para o acelerador é realizada de forma automática pelo compilador. Desse modo a matriz de cálculo não está armazenada no acelerador. Toda vez que o acelerador executa uma operação, as informações são gravadas na matriz que está na memória do hospedeiro. Esta operação tem um alto custo computacional, tornando a execução do código lenta (Figura 2.14).

```
$ pgcc -acc -ta=tesla -Minfo=all jacobiacc.c -o jacobiacc

main:
     29, Generating Tesla code
         30, #pragma acc loop gang /* blockIdx.x */
         31, #pragma acc loop vector(128) /* threadIdx.x */
     29, Generating implicit copyin(A[:][:]) [if not already present]
         Generating implicit copyout(Anew[1:1000][1:1000]) [if not already present]
     31, Loop is parallelizable
     39, Generating Tesla code
         40, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:dt)
         41, #pragma acc loop vector(128) /* threadIdx.x */
     39, Generating implicit copy(A[1:1000][1:1000],dt) [if not already present]
         Generating implicit copyin(Anew[1:1000][1:1000]) [if not already present]
     41, Loop is parallelizable
initialize:
     58, Memory zero idiom, loop replaced by call to __c_mzero8
$
```

**Figura 2.13:** Saída do compilador PGI sem uso de movimentação de dados

```
$ ./jacobiacc
Max error at iteration 1000 was 0.034767
O tempo de execução é 11.718717 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/jacobiacc.c
  main  NVIDIA  devicenum=0
    time(us): 3,339,276
    29: compute region reached 1000 times
        29: kernel launched 1000 times
            grid: [1000]  block: [128]
            elapsed time(us): total=152,516 max=181 min=147 avg=152
    29: data region reached 2000 times
        29: data copyin transfers: 1000
             device time(us): total=671,517 max=690 min=668 avg=671
        37: data copyout transfers: 1000
             device time(us): total=660,866 max=674 min=659 avg=660
    39: compute region reached 1000 times
        39: kernel launched 1000 times
            grid: [1000]  block: [128]
            elapsed time(us): total=275,134 max=421 min=268 avg=275
        39: reduction kernel launched 1000 times
            grid: [1]  block: [256]
            elapsed time(us): total=22,640 max=67 min=20 avg=22
    39: data region reached 2000 times
        39: data copyin transfers: 3000
             device time(us): total=1,336,393 max=683 min=5 avg=445
        47: data copyout transfers: 2000
             device time(us): total=670,500 max=678 min=7 avg=335
$
```

**Figura 2.14:** Tempo de execução sem uso de movimentação de dados

Para resolver esse problema é necessário informar ao compilador que uma cópia da matriz **A** deve ser feita para o acelerador no início da região paralela. Desse modo, não será

mais necessário gravar as informações no hospedeiro toda vez que for realizada uma operação de escrita na matriz pelo acelerador. Adicionalmente, a matriz **Anew** será criada exclusivamente no acelerador. Usaremos a diretiva **data** do OpenACC para realizar essas operações. Deve-se então adicionar a linha antes dos dois laços:

**#pragma acc data copy(A) create (Anew)**

```
#pragma acc data copy(A) create(Anew)
{
    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {
        #pragma acc parallel loop
        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                Anew[i][j] = 0.25 * (A[i+1][j] + A[i-1][j] +
                                     A[i][j+1] + A[i][j-1]);
            }
        }
        dt = 0.0;
        #pragma acc parallel loop reduction(max:dt)
        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
              dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
              A[i][j] = Anew[i][j];
            }
        }
        iteration++;
    }
}
```

Agora todas os acessos às matrizes **A** e **Anew** serão realizados exclusivamente no acelerador, com isto, não haverá custo adicional de acesso aos dados na memória do hospedeiro para o cálculo da matriz (Figura 2.15).

```
$ pgcc -acc -ta=tesla -Minfo=all jacobiaccdm.c -o jacobiaccdm

main:
    27, Generating create(Anew[:][:]) [if not already present]
        Generating copy(A[:][:]) [if not already present]
    30, Generating Tesla code
        31, #pragma acc loop gang /* blockIdx.x */
        32, #pragma acc loop vector(128) /* threadIdx.x */
    32, Loop is parallelizable
    40, Generating Tesla code
        41, #pragma acc loop gang /* blockIdx.x */
            Generating reduction(max:dt)
        42, #pragma acc loop vector(128) /* threadIdx.x */
    40, Generating implicit copy(dt) [if not already present]
    42, Loop is parallelizable
initialize:
    59, Memory zero idiom, loop replaced by call to __c_mzero8
$
```

**Figura 2.15:** Saída com do compilador PGI usando movimentação de dados

O tempo total gasto para a execução sem a movimentação de dados foi de 11,71 segundos.

Usando a movimentação de dados o tempo total passou para 0,80 segundos, ou seja, 14,62 vezes mais rápido. Este é um exemplo onde o uso de movimentação de dados propicia ganhos consideráveis (Figura 2.16).

```
$ ./jacobiaccdm
Max error at iteration 1000 was 0.034767
O tempo de execução é 0.809362 segundos.

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/jacobiaccdm.c
  main  NVIDIA  devicenum=0
    time(us): 15,135
    27: data region reached 2 times
        27: data copyin transfers: 1
             device time(us): total=689 max=689 min=689 avg=689
        50: data copyout transfers: 1
             device time(us): total=654 max=654 min=654 avg=654
    30: compute region reached 1000 times
        30: kernel launched 1000 times
            grid: [1000]  block: [128]
            elapsed time(us): total=149,177 max=182 min=144 avg=149
    40: compute region reached 1000 times
        40: kernel launched 1000 times
            grid: [1000]  block: [128]
            elapsed time(us): total=267,672 max=278 min=261 avg=267
        40: reduction kernel launched 1000 times
            grid: [1]  block: [256]
            elapsed time(us): total=20,539 max=37 min=19 avg=20
    40: data region reached 2000 times
        40: data copyin transfers: 1000
             device time(us): total=5,599 max=14 min=4 avg=5
        48: data copyout transfers: 1000
             device time(us): total=8,193 max=23 min=7 avg=8
$
```

**Figura 2.16:** Tempo de execução usando movimentação de dados

As Figuras 2.17 e 2.18 apresentam os resultados gerados usando o **pgprof**. Na Figura 2.17 a análise do código foi realizada sem movimentação de dados e a Figura 2.18 com a movimentação de dados para o acelerador.

Nota-se que o tempo e o custo computacional são maiores quando não é feita a movimentação de dados, visto que toda vez que se faz uma operação na matriz, existe a necessidade do acelerador fazer o acesso aos dados no hospedeiro. Quando a matriz é movimentada de forma definitiva para o acelerador, esse custo computacional é muito menor.

### 2.9.3. Cálculo do fractal de Mandelbrot

Os fractais são figuras geométricas complexas que apresentam como característica principal a autossimilaridade. O fractal de Mandelbrot é um fractal definido como o conjunto de pontos C no plano complexo. O conjunto de Mandelbrot é obtido quando submetemos

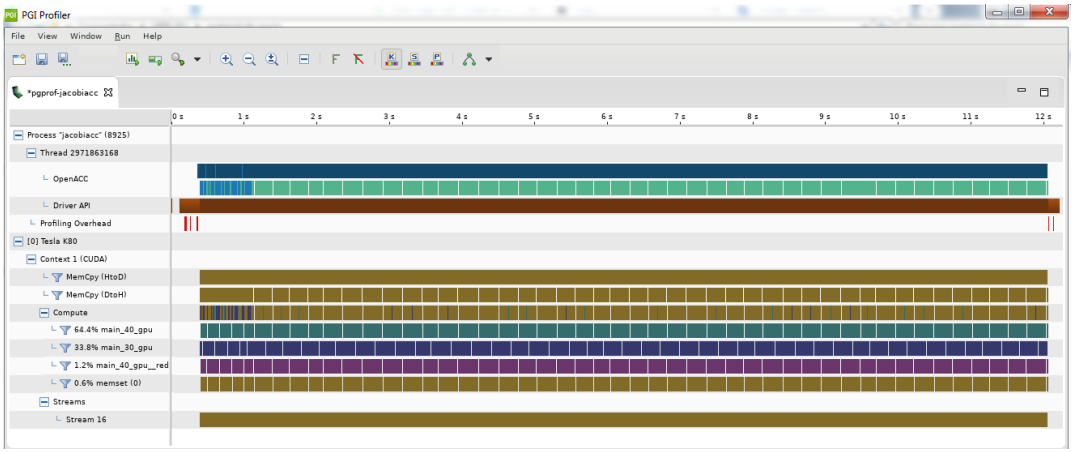



**Figura 2.17:** Análise de execução do código sem uso de movimentação de dados

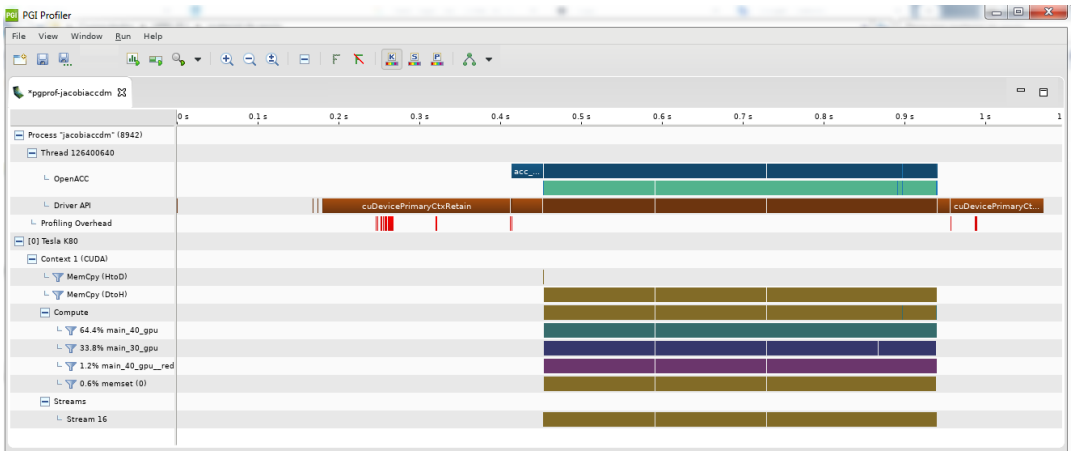



**Figura 2.18:** Análise de execução do código usando movimentação de dados

os números complexos a um processo iterativo e recursivo utilizando a fórmula:

$$Z_{n+1} = {Z_n}^2 + C \tag{1}$$

A execução do fractal de Mandelbrot dentro do laço na versão sequencial (Exemplo 2.8) será feito por uma *thread*, independente da quantidade de processadores que existam no sistema (Figura 2.19).

```c
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <omp.h>
#include <stdlib.h>
#define X_RESN 800 /* x resolution */
#define Y_RESN 800 /* y resolution */
typedef struct complextype
{
  float real, imag;
} Compl;
void main(int argc, char *argv[]) {
  unsigned int width, height,
      x, y,
      border_width,
      display_width, display_height,
      screen;
  char *window_name = "Mandelbrot", *display_name = NULL;
  unsigned long valuemask = 0;
  FILE *fp, *fopen();
  char str[100];
  int i, j, k;
  Compl z, c;
  float lengthsq, temp;
  width = X_RESN;
  height = Y_RESN;
  x = 0;
  y = 0;
  border_width = 4;
  double t_inicio = omp_get_wtime();
  int counter = 0;
  for (i = 0; i < X_RESN; i++)
    for (j = 0; j < Y_RESN; j++)
    {
      z.real = z.imag = 0.0;
      c.real = ((float)j - 400.0) / 200.0;
      c.imag = ((float)i - 400.0) / 200.0;
      k = 0;
      do
      {
        temp = z.real * z.real - z.imag * z.imag + c.real;
        z.imag = 2.0 * z.real * z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real * z.real + z.imag * z.imag;
        k++;
      } while (lengthsq < 4.0 && k < 100000);
      if (k == 100000){
          counter++;
      }
    }
  double t_fim = omp_get_wtime();
  printf("Imagem: %d e Y: %d \n", X_RESN, Y_RESN);
  printf("Tempo: \t %f\n", t_fim-t_inicio);
  printf("Counter %d\n", counter);
}
```

**Exemplo 2.8:** Mandelbrot Sequencial

```
$ pgcc -Minfo=all mandelbrotserial.c -o mandelbrotserial

main:
     84, FMA (fused multiply-add) instruction(s) generated
$
$ ./mandelbrotserial
Com tamanho de imagem X: 800 e Y: 800
Tempo para executar sequencial:          73.652713
Counter 60312
$
```

**Figura 2.19:** Compilação Mandelbrot sequencial e tempo de execução

O tempo total para a execução sequencial foi de 73,65 segundos. Para a versão em OpenACC (Figura 2.20), usaremos inicialmente a diretiva **parallel** adicionando a linha:

**#pragma acc parallel loop copy**

```
#pragma acc parallel loop copy(counter)
for (i = 0; i < X_RESN; i++)
  for (j = 0; j < Y_RESN; j++)
```

```
$ pgcc -acc -ta=tesla -Minfo=all mandelbrotacc.c -o mandelbrotacc
main:
     52, Generating copy(counter) [if not already present]
         Generating Tesla code
         53, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         54, #pragma acc loop seq
         74, Generating implicit reduction(+:counter)
     54, Loop carried scalar dependence for counter at line 74
     69, Accelerator restriction: induction variable live-out from loop: k
     70, Accelerator restriction: induction variable live-out from loop: k
$
$ ./mandelbrotacc
Com tamanho de imagem X: 800 e Y: 800
Tempo para executar paralelizado:          5.361158
Counter 60314

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/mandelbrotacc.c
  main  NVIDIA  devicenum=0
    time(us): 69
    52: compute region reached 1 time
        52: kernel launched 1 time
            grid: [7]  block: [128]
            elapsed time(us): total=5,056,435 max=5,056,435 min=5,056,435 avg=5,056,435
        52: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=76 max=76 min=76 avg=76
    52: data region reached 2 times
        52: data copyin transfers: 1
             device time(us): total=15 max=15 min=15 avg=15
        78: data copyout transfers: 1
             device time(us): total=54 max=54 min=54 avg=54
$
```

**Figura 2.20:** Compilação do código usando OpenACC e o tempo de execução

Embora esta versão do código usando OpenACC tenha um tempo de execução de 5,36

segundos, sendo 13,74 vezes mais rápido que a versão sequencial, podemos melhorar o código usando outras diretivas. O tempo de execução pode ainda ser diminuído com o uso da diretiva **atomic**:

**#pragma acc atomic update**

```
      if (k == 100000){
        #pragma acc atomic update
          counter++;
      }
    }
```

Com o uso dessa diretiva o tempo de execução passou para 0,40 segundos. Nessa nova versão do código o tempo de execução foi 13,40 vezes mais rápido em comparação de primeira versão do OpenACC e 184,12 mais rápido vezes em comparação a versão sequencial (Figura 2.21).

```
$ pgcc -acc -ta=tesla -Minfo=all mandelbrotacc.c -o mandelbrotacc2

main:
     52, Generating copy(counter) [if not already present]
         Generating Tesla code
         53, #pragma acc loop gang /* blockIdx.x */
         54, #pragma acc loop vector(128) /* threadIdx.x */
     54, Loop is parallelizable
     69, Accelerator restriction: induction variable live-out from loop: k
     70, Accelerator restriction: induction variable live-out from loop: k
$
$ ./mandelbrotacc2
Com tamanho de imagem X: 800 e Y: 800
Tempo para executar paralelizado:        0.406515
Counter 60314

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/mandelbrotacc.c
  main  NVIDIA  devicenum=0
    time(us): 72
    52: compute region reached 1 time
        52: kernel launched 1 time
            grid: [800]  block: [128]
            elapsed time(us): total=103,340 max=103,340 min=103,340 avg=103,340
    52: data region reached 2 times
        52: data copyin transfers: 1
             device time(us): total=14 max=14 min=14 avg=14
        78: data copyout transfers: 1
             device time(us): total=58 max=58 min=58 avg=58
$
```

**Figura 2.21:** Compilação do código usando a diretiva atomic e o tempo de execução

O uso da diretiva **parallel** e a movimentação de dados obteve um ganho significativo em relação a versão sequencial (Figura 2.22). Porém, quando foi feito o uso da diretiva **atomic**, esse ganho foi muito superior a versão sequencial (Figura 2.23).

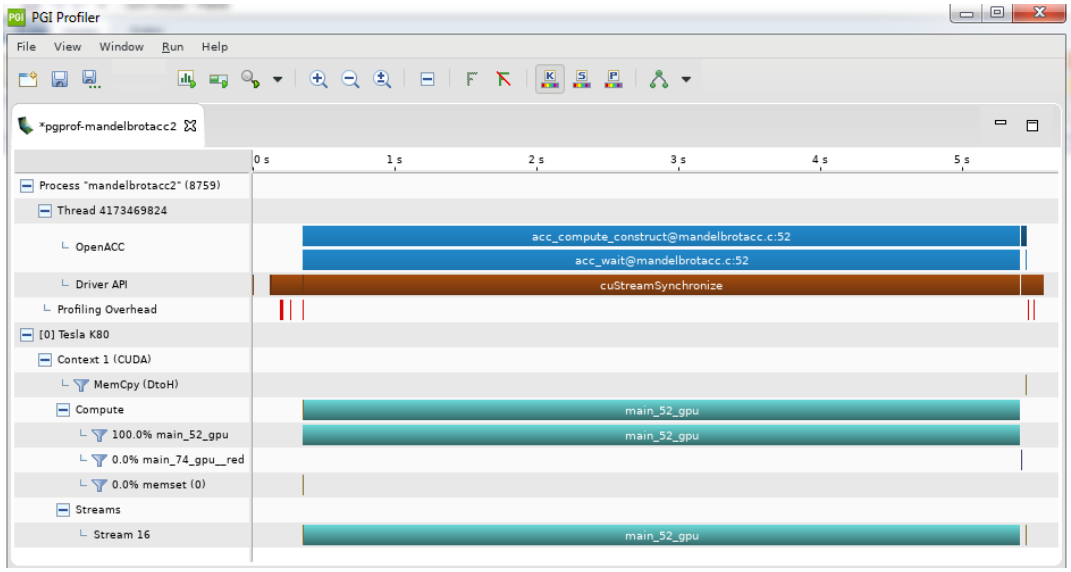



**Figura 2.22:** Análise de execução do código usando movimentação de dados

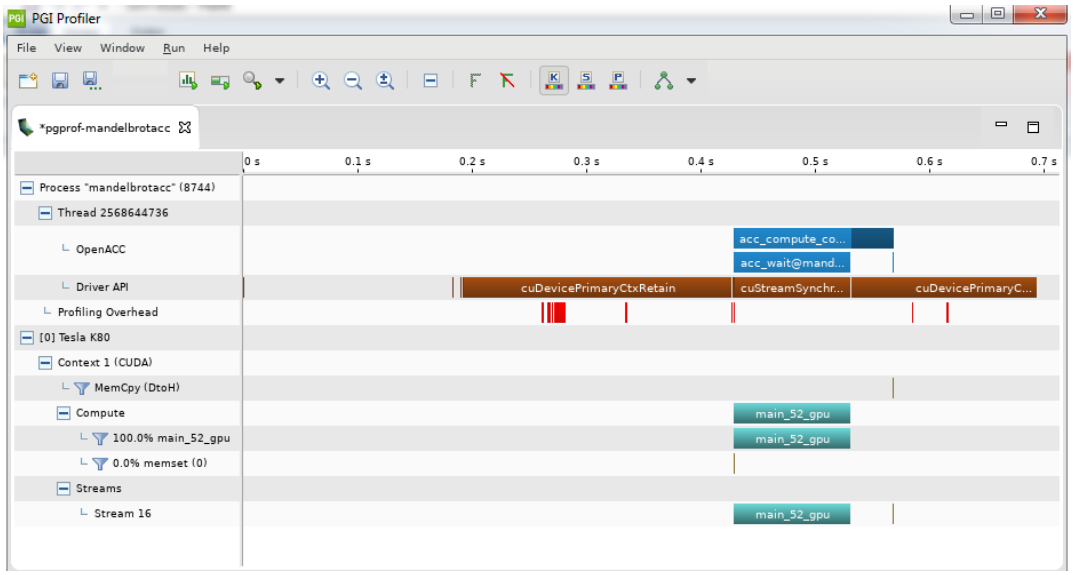



**Figura 2.23:** Análise de execução do código usando movimentação de dados e a diretiva atomic

### 2.9.4. Cálculo de números primos

O código do Exemplo 2.9 calcula a quantidade de números primos entre 0 e um determinado valor inteiro N. Este programa basicamente verifica se N é divisível por algum número ímpar entre 0 e a raiz quadrada de N, sendo que os números pares são descartados de imediato.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
int primo (long int n) {
        for (long int i = 3; i < (long int)(sqrt(n) + 1); i+=2)
             if (n%i == 0)
                 return 0;
        return 1;
}
int main(int argc, char *argv[]) {
    long long int i, n, quantidadePrimos = 0;
    if (argc < 2) {
        printf("Valor invalido! Entre com o valor do maior inteiro
   \n");
        return 0;
    }
    else {
        n = strtol(argv[1], (char **) NULL, 10);
    }
    double inicio = omp_get_wtime();
    for (i = 3; i <= n; i += 2)
         if(primo(i) == 1) quantidadePrimos++;
    quantidadePrimos += 1;
    double fim = omp_get_wtime();
    printf("Quantidade de numeros primos entre 1 e %ld e : %ld \n"
   , n, quantidadePrimos);
    printf("O tempo de execucao foi de : %f \n", fim-inicio);
    return(0);
}
```

**Exemplo 2.9:** Primos Sequencial

Na Figura 2.24 pode ser vista a compilação do código e o tempo total para a execução sequencial, que foi de 16,12 segundos. Para a versão em OpenACC, primeiramente usaremos as diretivas **routine** e **parallel** dentro da rotina **primo()**. Para o uso da diretiva **routine** adicionar a linha:

**#pragma acc routine**

Antes de iniciar a rotina e a linha:

**#pragma acc loop**

Antes do primeiro laço para calcular a raiz quadrada.

```
#pragma acc routine
int primo (long int n) {
    #pragma acc loop
        for (long int i = 3; i < (long int)(sqrt(n) + 1); i+=2)
             if (n%i == 0)
                 return 0;
        return 1;
}
```

Para o programa principal adicionaremos a linha:

**#pragma acc parallel loop reduction(+:quantidadePrimos)**

Antes do segundo laço para calcular a quantidade de números primos existentes no intervalo. O resultados podem ser vistos na Figura 2.25.

```
    #pragma acc parallel loop reduction(+:quantidadePrimos)
    for (i = 3; i <= n; i += 2)
         if(primo(i) == 1) quantidadePrimos++;
    quantidadePrimos += 1;
```

```
$ pgcc -Minfo=all primosserial.c -o primosserial
$
$ ./primosserial 10000000
Quantidade de números primos entre 1 e 10000000 é : 664579
O tempo de execução foi de : 16.123722
$
```

**Figura 2.24:** Compilação do código sequencial e tempo de execução

```
$ pgcc -acc -ta=tesla -Minfo=all primosacc.c -o primosacc

primo:
      7, Generating acc routine seq
         Generating Tesla code
main:
     33, Generating Tesla code
         34, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(+:quantidadePrimos)
     33, Generating implicit copy(quantidadePrimos) [if not already present]
$
$ ./primosacc 10000000
Quantidade de números primos entre 1 e 10000000 é : 664579
O tempo de execução foi de : 0.437250

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/primosacc.c
  main  NVIDIA  devicenum=0
    time(us): 68
    33: compute region reached 1 time
        33: kernel launched 1 time
            grid: [39063]  block: [128]
            elapsed time(us): total=137,373 max=137,373 min=137,373 avg=137,373
        33: reduction kernel launched 1 time
            grid: [1]  block: [256]
            elapsed time(us): total=89 max=89 min=89 avg=89
    33: data region reached 2 times
        33: data copyin transfers: 1
             device time(us): total=14 max=14 min=14 avg=14
        37: data copyout transfers: 1
             device time(us): total=54 max=54 min=54 avg=54
$
```

**Figura 2.25:** Compilação do código usando OpenACC e o tempo de execução

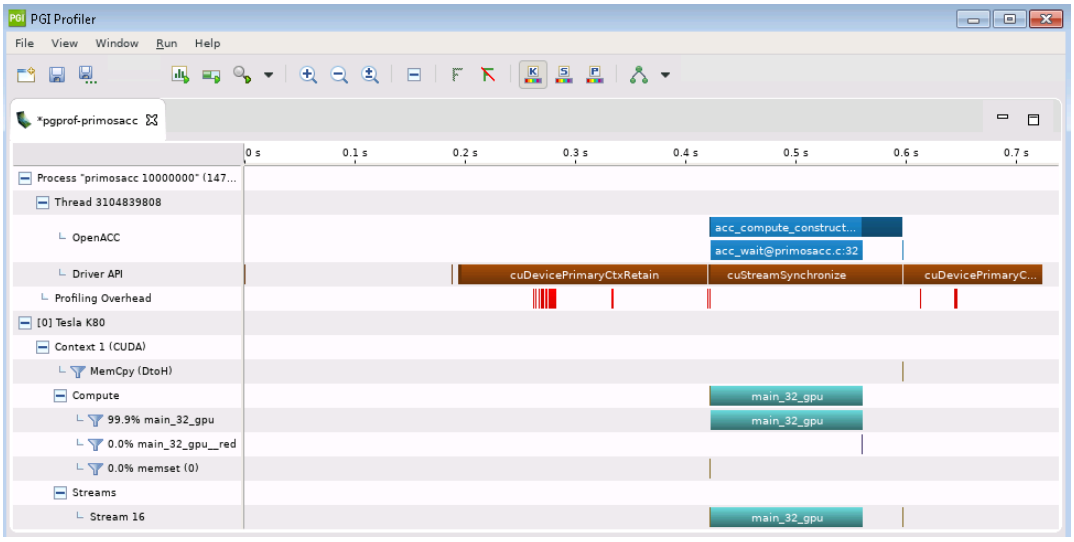



**Figura 2.26:** Análise de execução do código usando as diretiva Routine e parallel

A Figura 2.26 apresenta a análise da execução do código para o cálculo de números primos usando as diretivas **routine** e **parallel** utilizando o programa **pgprof**.

### 2.9.5. Cálculo de multplicação de matrizes

A multiplicação de matrizes corresponde ao produto entre duas matrizes. O produto de duas matrizes só é possível somente quando o número de colunas da primeira matriz é igual ao número de linhas da segunda matriz. O algoritmo que iremos apresentar, conhecido como "ingênuo", apresenta complexidade computacional $O(mnp)$, para a multiplicação de uma matriz $m$ x $n$ por outra $n$ x $p$, se todas as dimensões forem iguais a "n", diz-se que a complexidade é $O(n^3)$.

Utilizando-se álgebra linear, podem-se obter algoritmos que podem alcançar complexidades melhores, tais como o algoritmo do alemão Volker Strassen, que consegue uma complexidade de $O(n^{2,807})$ pela redução do número de multiplicações necessárias necessárias para cada sub-matriz 2x2 de 8 para 7.

Um outro algoritmo conhecido de multiplicação de matrizes é o Coppersmith-Winograd, com uma complexidade de $O(n^{2,3737})$. Contudo, a menos que as matrizes sejam enormes, esses algoritmos não resultam em reduções significativas no tempo de computação.

Assim sendo, na prática, o melhor método para acelerar a multiplicação de matrizes é o uso de algoritmos paralelos, como os que iremos apresentar a seguir. Contudo, vamos apresentar primeiramente o código sequencial "ingênuo", qque pode ser visto no Exemplo 2.10.

```
#include <stdio.h>
#include <omp.h>
#define SIZE 5000
float a[SIZE][SIZE];
float b[SIZE][SIZE];
float c[SIZE][SIZE];
int main() {
int i,j,k;
double tIni, tFinal;
    for (i = 0; i < SIZE; ++i) {
      for (j = 0; j < SIZE; ++j) {
          a[i][j] = (float)i + j;
          b[i][j] = (float)i - j;
          c[i][j] = 0.0f;
      }
    }
    tIni = omp_get_wtime();
    for (i = 0; i < SIZE; ++i) {
       for (j = 0; j < SIZE; ++j) {
          for(k=0; k < SIZE; ++k)
                c[i][j] = a[i][k] * b[k][j];
       }
    }
    tFinal = omp_get_wtime();
    printf("tempo: %3.6f\n", tFinal - tIni);
    return 0;
}
```

**Exemplo 2.10:** Multiplicação de matrizes sequencial

Esse código sequencial apresenta diversos problemas para uma implementação em OpenACC. Em primeiro lugar, deve ser feita a linearização das matrizes, para que o acesso aos dados no acelerador seja feito de uma forma mais otimizada, ou seja, em endereços sequenciais na memória.

Em segundo lugar, deve-se declarar uma variável temporária para armazenar o valor de **c[i][j]** que está sendo calculado no laço mais interno. Da forma que está colocada no código original, serão gerados acessos desnecessários à matriz **c**, independemente de ela estar armazenada na memória do hospedeiro ou do acelerador. Logo, no Exemplo 2.11 chegamos a uma versão sequencial mais adequada para trabalharmos a conversão para OpenACC.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5000
int main() {
float *a = (float*) malloc (sizeof(float)*SIZE*SIZE);
float *b = (float*) malloc (sizeof(float)*SIZE*SIZE);
float *c = (float*) malloc (sizeof(float)*SIZE*SIZE);
int   i, j, k;
float temp;
    for (i = 0; i < SIZE; ++i) {
      for (j = 0; j < SIZE; ++j) {
          a[i*SIZE+j] = (float)i + j;
          b[i*SIZE+j] = (float)i - j;
          c[i*SIZE+j] = 0.0f;
      }
    }
    for (i = 0; i < SIZE; ++i) {
       for (j = 0; j < SIZE; ++j) {
     temp  = 0.0;
          for (k = 0; k < SIZE; ++k) {
           temp += a[i*SIZE+k] + b[k*SIZE + j];
           }
           c[i*SIZE+j] = temp;
      }
    }
    return 0;
}
```

**Exemplo 2.11:** Multiplicação de matrizes otimizada

Para uma primeira versão em OpenACC utilizaremos a diretiva **kernels** e **data** e veremos como o compilador se comporta para realizar essa paralelização. Essa primeira versão é apresentada no Exemplo 2.12.

```
#pragma acc data copyin (a[0:SIZE*SIZE], b[0:SIZE*SIZE]), copy(c
   [0:SIZE*SIZE])
{
int   i, j, k;
float temp;
#pragma acc kernels loop
    for (i = 0; i < SIZE; ++i) {
      for (j = 0; j < SIZE; ++j) {
          a[i*SIZE+j] = (float)i + j;
          b[i*SIZE+j] = (float)i - j;
          c[i*SIZE+j] = 0.0f;
      }
    }
#pragma acc kernels loop
    for (i = 0; i < SIZE; ++i) {
       for (j = 0; j < SIZE; ++j) {
     temp  = 0.0;
          for (k = 0; k < SIZE; ++k) {
           temp += a[i*SIZE+k] + b[k*SIZE + j];
```

```
            }
            c[i*SIZE+j] = temp;
        }
    }
}
```

**Exemplo 2.12:** Multiplicação de matrizes - versão inicial OpenACC

Como podemos observar, o ganho de desempenho obtido não é tão relevante para essa versão, mesmo com os cuidados para a movimentação de dados para o acelerador e otimizações realizadas (Figura 2.27).

```
$ pgcc -acc -ta=tesla -Minfo=all matmulacc.c -o matmulacc

main:
     13, Generating copyin(a[:25000000]) [if not already present]
         Generating copy(c[:25000000]) [if not already present]
         Generating copyin(b[:25000000]) [if not already present]
     16, Complex loop carried dependence of a->,b-> prevents parallelization
         Loop carried dependence of a->,b-> prevents parallelization
         Loop carried backward dependence of b->,a-> prevents vectorization
         Complex loop carried dependence of c-> prevents parallelization
         Accelerator serial kernel generated
         Generating Tesla code
         16, #pragma acc loop seq
         17, #pragma acc loop seq
     17, Complex loop carried dependence of a->,b->,c-> prevents parallelization
     24, Complex loop carried dependence of c->,b-> prevents parallelization
         Loop carried dependence of c-> prevents parallelization
         Loop carried backward dependence of c-> prevents vectorization
         Complex loop carried dependence of a-> prevents parallelization
     25, Complex loop carried dependence of c->,b->,a-> prevents parallelization
         Generating Tesla code
         24, #pragma acc loop seq
         25, #pragma acc loop seq
         27, #pragma acc loop vector(128) /* threadIdx.x */
         28, Generating implicit reduction(+:temp)
     27, Loop is parallelizable
$
$ ./matmulacc
Tempo de execução: 485.34

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/matmulacc.c
  main  NVIDIA  devicenum=0
    time(us): 33,054
    13: data region reached 2 times
        13: data copyin transfers: 18
             device time(us): total=24,876 max=1,398 min=1,337 avg=1,382
        34: data copyout transfers: 6
             device time(us): total=8,178 max=1,376 min=1,333 avg=1,363
    15: compute region reached 1 time
        16: kernel launched 1 time
            grid: [1]  block: [1]
            elapsed time(us): total=1,667,599 max=1,667,599 min=1,667,599 avg=1,667,599
    23: compute region reached 1 time
        25: kernel launched 1 time
            grid: [1]  block: [128]
            elapsed time(us): total=483,247,625 max=483,247,625 min=483,247,625 avg=483,247,625
$
```

**Figura 2.27:** Compilação do código usando a diretiva kernels

No Exemplo 2.13 iremos fazer uso da diretiva **parallel** e das cláusulas **reduction**, **collapse** e **tile**. A diretiva que tem maior impacto é **tile** pelo fato de permitir o acesso otimizado

a sub-blocos das matrizes. Esse acesso é realizado nos níveis mais altos da hierarquia do acelerador, a partir cache de nível 2, resultando em tempos de computação muito mais otimizados. Note que essa otimização só é eficiente porque o laço aninhado possui três níveis e matriz foi linearizada (Figura 2.28).

```
#pragma acc data copyin (a[0:SIZE*SIZE], b[0:SIZE*SIZE]), copy(c
    [0:SIZE*SIZE])
{
    #pragma acc parallel loop gang vector collapse(2)
    for (i = 0; i < SIZE; ++i) {
      for (j = 0; j < SIZE; ++j) {
          a[i*SIZE+j] = (float)i + j;
          b[i*SIZE+j] = (float)i - j;
          c[i*SIZE+j] = 0.0f;
      }
    }
    #pragma acc parallel
    #pragma acc loop tile(256,256) independent
    for (i = 0; i < SIZE; ++i) {
      for (j = 0; j < SIZE; ++j) {
          temp  = 0.0;
                 #pragma acc loop reduction(+:temp)
                 for (k = 0; k < SIZE; ++k) {
                     temp += a[i*SIZE+k] + b[k*SIZE + j];
                 }
                 c[i*SIZE+j] = temp;
      }
    }
}
```

**Exemplo 2.13:** Multiplicação de matrizes - versão final OpenACC

```
$ pgcc -acc -ta=tesla -Minfo=all matmulacc2.c -o matmulacc2

main:
     13, Generating copyin(a[:25000000]) [if not already present]
         Generating copy(c[:25000000]) [if not already present]
         Generating copyin(b[:25000000]) [if not already present]
     15, Generating Tesla code
         16, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         17,   /* blockIdx.x threadIdx.x collapsed */
     23, Generating Tesla code
         25, #pragma acc loop gang, vector tile(256,256) /* blockIdx.x threadIdx.x */
         26,   /* blockIdx.x threadIdx.x tiled */
         29, #pragma acc loop seq
     29, Loop is parallelizable
$
$ ./matmulacc2
Tempo de execução: 0.47

Accelerator Kernel Timing data
/home/ebcosta/openacc/codes/erad/matmulacc2.c
  main  NVIDIA  devicenum=0
    time(us): 33,014
    13: data region reached 2 times
        13: data copyin transfers: 18
             device time(us): total=24,873 max=1,402 min=1,336 avg=1,381
        36: data copyout transfers: 6
             device time(us): total=8,141 max=1,365 min=1,319 avg=1,356
    15: compute region reached 1 time
        15: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=1,887 max=1,887 min=1,887 avg=1,887
    23: compute region reached 1 time
        23: kernel launched 1 time
            grid: [25600]  block: [1024]
            elapsed time(us): total=40,997 max=40,997 min=40,997 avg=40,997
$
```

**Figura 2.28:** Compilação do código usando a diretiva parallel

A Figura 2.29 apresenta a análise da execução do código de multiplicação de matrizes usando a diretivas **kernels**, como dito anteriormente mesmo com a movimentação de dados o ganho de desempenho obtido não é tão relevante.

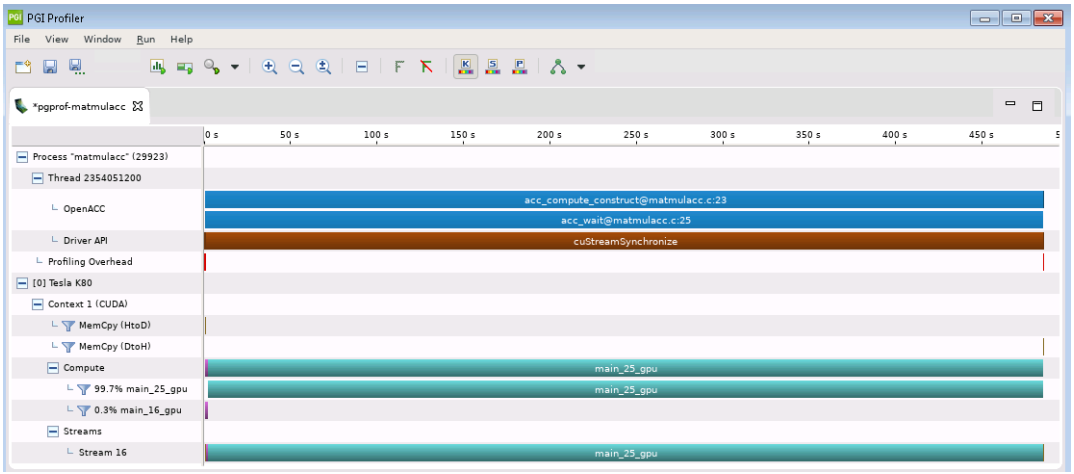



**Figura 2.29:** Análise de execução do código usando as diretiva data e parallel

A Figura 2.30 apresenta a análise da execução do código de multiplicação de matrizes usando a diretivas **parallel** com as cláusulas **collapse** e **tile**, observa-se que o desempenho foi muito superior a versão anterior utilizando a diretiva **kernels**.

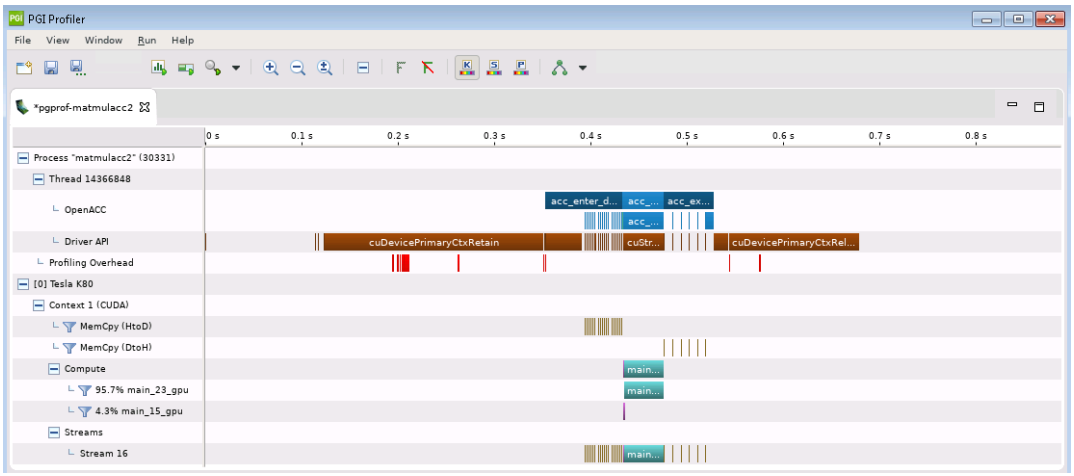



**Figura 2.30:** Análise de execução do código usando as diretiva data e parallel

## Referências

[Chen 2017] Chen, S. (2017). *Introduction to OpenACC*. Research Computing Services Information Services and Technology Boston University.

[Harris 2017] Harris, M. (2017). *Unified Memory for CUDA Beginners*. NVIDIA Corporation.

[Murphy 2016] Murphy, J. (2016). *More Tips on OpenACC Acceleration*. Microway Corporation.

[NVIDIA 2014] NVIDIA (2014). *NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110/210*. NVIDIA Corporation.

[PGI 2019] PGI (2019). *PROFILER USER'S GUIDE*. NVIDIA Corporation.