Capítulo
2

Introdução à Programação com OpenACC

Evaldo B. Costa, Gabriel P. Silva
Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brasil

Resumo

O OpenACC (programação para aceleradores) é um modelo de programação para computação paralela desenvolvido com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e portabilidade entre vários tipos de arquiteturas: multicore, manycore e GPUs. Este minicurso tem por objetivo apresentar este novo modelo de programação e suas facilidades de uso. A proposta deste minicurso é oferecer uma introdução à programação com OpenACC através de uma abordagem expositiva como: uma visão geral dos conceitos, diretivas e cláusulas; acrescidas da utilização de exemplos com os diversos tipos de arquiteturas alvo.

2.1. Introdução

As arquiteturas paralelas tem alcançado um alto grau de paralelismo com a utilização de um número cada vez maior de processadores, como podem ser encontrados nas arquiteturas *multicore, manycore* e GPUs (Figura 2.1). Esses tipos de arquitetura, com uso de processamento paralelo maciço, são usados extensivamente em aplicações nas áreas de geofísica, sequenciamento genético, simulações de modelos matemáticos e previsão do tempo, entre outras (SILVA, 2018) (COSTA; SILVA; TEIXEIRA, 2018).

Entre os tipos de arquiteturas existentes destacam-se as GPUs e *manycore*. Desenvolvida pela NVIDIA nos anos 90 a GPU (*Graphical Processing Unit*) se baseia em um grande número de núcleos para processamento paralelo maciço com foco na eficiência energética e para aplicações com demandas que melhorem o *throughput*. Inicialmente desenvolvidos com o objetivo de atender a área de jogos, rapidamente mostrou-se muito eficiente em outras áreas (A.; M., 2012).

Em 2010 a Intel iniciou os primeiros estudos para o desenvolvimento da arquitetura MIC (*Many Integrated Core*). A arquitetura do Intel MIC (*Many Integrated Core*) oferece paralelismo maciço e vetorização com foco em computação de alto desempenho

(HPC), que utiliza grandes demandas de dados para processamento. Atualmente essas arquiteturas estão presentes em todos os setores, como a engenharia, medicina, economia e finanças (RAHMAN, 2013).

Na programação dessas arquiteturas são utilizados modelos de programação como CUDA, OpenCL e OpenACC. Neste minicurso usaremos o modelo de programação OpenACC, que surgiu em 2011 utilizando diretivas de compilação de alto nível e foi desenvolvido para o uso em aceleradores como os produzidos pela NVIDIA e Intel (LARKIN, 2018).

O OpenACC foi desenvolvido pelos principais fabricantes de hardware e software como a Cray, CAPS, NVIDIA e PGI, com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e tornando possível a portabilidade do código independente de qual arquitetura foi desenvolvida inicialmente (OPENACC-STANDARD.ORG, 2015) (CHEN, 2017).

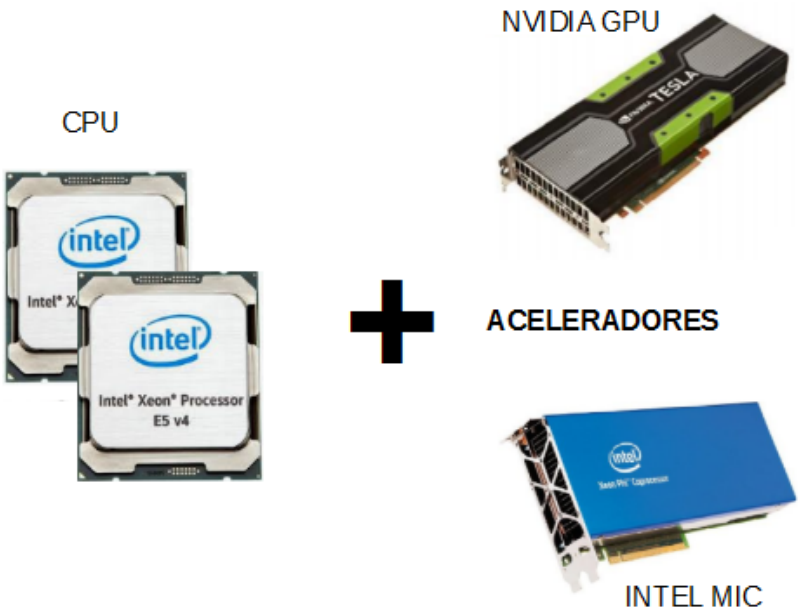


Figura 2.1: Tipos de arquiteturas: multicore, GPU e manycore

As *threads* são executados por processadores escalares. Os *thread blocks* são executados em multiprocessadores e não podem migrar. Diversos *thread blocks* podem residir em um multiprocessador, limitados pelo total de recursos disponíveis no multiprocessador, tais como memória compartilhada e banco de registradores. Um *kernel* é lançado como uma grade de *thread blocks*. Veja na Figura 2.2.

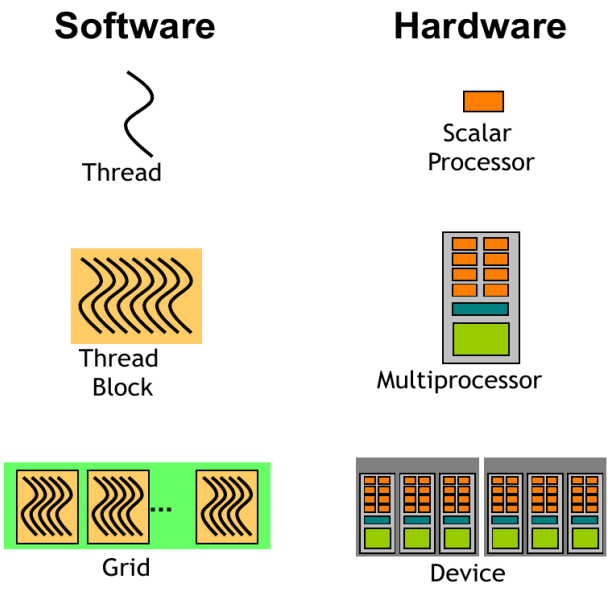


Figura 2.2: Modelo de Execução (ABBOTT, 2017)

Este minicurso introdutório à programação OpenACC consiste de três partes: na primeira parte serão abordados os conceitos iniciais, como definição do modelo, vantagens e desvantagens e onde melhor se aplica o seu uso; na parte seguinte estudaremos quais as principais diretivas usadas; finalmente na última parte veremos quais compiladores suportam o OpenACC e como compilar um programa, sendo que também examinaremos um exemplo com análise dos resultados.

2.2. Referencial teórico

2.2.1. Avaliação de desempenho

Antes de iniciarmos o estudo do OpenACC é importante destacar alguns conceitos sobre como avaliar o desempenho de um programa paralelo.

Uma das questões que surgem ao elaborar um programa paralelo é saber se o mesmo apresenta um desempenho adequado quando executado em um ambiente paralelo. Deste modo, para avaliar o desempenho de um sistema de processamento paralelo, as métricas mais importantes para serem consideradas são: *speedup* (também conhecido como ganho de desempenho ou aceleração), eficiência e escalabilidade.

São diversos fatores que influenciam essas métricas, tais como o custo de sincronização e comunicação, a distribuição das tarefas entre os processadores e o percentual do tempo de execução do programa que é passível de paralelização.

2.2.1.1. Speedup

O ***speedup***, ou aceleração, mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação em um único processador ($T(1)$) e o tempo gasto na execução com P

processadores ($T(P)$), como visto na Equação 1:

$$S(P) = \frac{T(1)}{T(P)} \tag{1}$$

Em condições ideais, quando o *speedup* é sempre igual a P, onde P é o número de processadores em uso, temos o chamado *speedup* linear. Mas, em geral, o *speedup* é menor do que P, devido principalmente à sobrecarga de comunicação entre os diferentes fluxos de execução do programa, perdas por sincronização e decomposição de tarefas mal feita. Quando isso acontece, chamamos o *speedup* de sublinear. Essa situação pode se deteriorar até o ponto em que a adição de mais processadores diminui, em vez de aumentar, o ganho obtido, caracterizando o que se chama de "retorno negativo".

Em algumas situações especiais, *speedups* superiores a P podem ser obtidos (denominados "*speedups* superlineares"). Exemplos destas situações são aplicações onde o volume de dados manipulados é grande o suficiente para exceder o tamanho da memória cache de um único processador. Nesse caso, ao dividir a aplicação entre P processadores, o volume de dados manipulado por cada processador é aproximadamente dividido por P, sendo este volume agora pequeno o suficiente para poder ser armazenado integralmente, ou com baixo nível de interferência destrutiva, na memória cache de cada processador. Dentro deste quadro, e respeitadas as condições mencionadas anteriormente, é razoável se esperar um *speedup* superior a P no processo de paralelização, já que o desempenho com um único processador fica muito prejudicado pela baixa taxa de acerto nos acessos à memória cache.

Uma situação análoga ocorre quando são feitas buscas em grandes bases de dados, tais como a busca de dados genômicos realizadas por diversas implementações paralela do programa BLAST. Neste caso, a quantidade de memória RAM acumulada de cada um dos nós em um *cluster* permite que a base de dados se mova do disco inteiramente para a memória, reduzindo portanto dramaticamente o tempo requerido, por exemplo, para o mpiBLAST percorrer toda a base de dados (CORREA; SILVA, 2012).

O *speedup* superlinear pode ocorrer também, sob determinadas condições, quando da execução de algoritmos de *backtracking* e *branch and bound* paralelos (SILVA, 2018). A Figura 2.3 ilustra diferentes curvas de *speedups*.

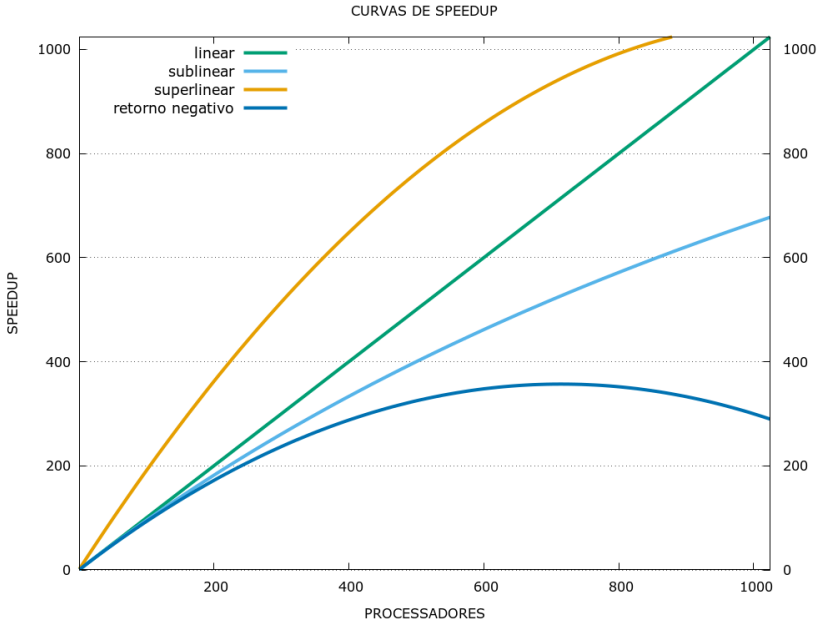


Figura 2.3: Curvas de *speedup*

2.2.1.2. Eficiência

A **eficiência** é uma medida de quão efetiva é a adição de novos processadores para ajudar na resolução de um problema. Seu valor é obtido pela razão entre o *speedup* ($S(P)$) e o número de processadores (P) utilizados para obter este *speedup*, conforme podemos observar na equação 2:

$$E(P) = \frac{S(P)}{P} \tag{2}$$

Novamente, como o *speedup* em geral é menor do que P por conta da sobrecarga do processamento paralelo, a eficiência tipicamente assume um valor menor do que 1, mas, preferencialmente, próximo de 1. Para se obter *speedup* muito próximo de P e, consequentemente, eficiência muito próxima de 1, as seguintes condições devem normalmente ser satisfeitas:

- no código a ser paralelizado, o percentual de código não paralelizável (que continuará sendo executado de forma sequencial) é muito pequeno;
- no processo de paralelização, a distribuição de carga de trabalho entre os P processadores deve ser homogênea;
- os processadores trabalham nos trechos de código executados em paralelo de forma bastante independente, requerendo muito pouca comunicação ou sincronização entre eles.

2.2.1.3. Escalabilidade

Um sistema é dito escalável quando sua eficiência se mantém constante à medida que o número de processadores P aplicado à solução do problema cresce. Se o tamanho do problema é mantido constante e o número de processadores aumenta, o *overhead* de comunicação tende a crescer e a eficiência a diminuir. Na prática, uma análise da escalabilidade deve considerar a possibilidade de se aumentar proporcionalmente o tamanho do problema a ser resolvido à medida que P cresce de forma a contrabalançar o natural aumento do *overhead* de comunicação quando P cresce.

Considere como exemplo um problema de tamanho S. Usando P processadores esse problema leva um tempo T para ser executado. O sistema é dito escalável se um problema de tamanho $2S$ executado em $2P$ processadores leva o mesmo tempo T. Escalabilidade é frequentemente uma propriedade mais desejável que o *speedup*.

É importante ter essas métricas em mente, para sabermos se os programas paralelos que iremos desenvolver em OpenACC estão realmente tendo o desempenho esperado.

2.3. Programação em OpenACC

O OpenACC é um modelo de programação aberta para computação paralela desenvolvido com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e portabilidade entre diversos tipos de arquiteturas: *multicore, manycore* e GPUs.

O OpenACC é compatível com os modelos de programação OpenMP e MPI, ambas as abordagens podem ser combinadas com o OpenACC. Em geral, as diretivas do OpenACC são muito semelhantes às do OpenMP. Em relação ao CUDA, OpenACC é totalmente compatível tornando a necessidade de alteração do código a menor possível.

O modelo de programação OpenACC possui algumas características que o tornam fácil e simples de utilizar, como:

- Independente de fabricante;
- Oculta a complexidade do *hardware* dos programadores;
- Requer poucas modificações ao código fonte;
- Mais fácil de programar e depurar que o CUDA;
- Possui algumas facilidades que o CUDA não oferece;
- Mesmo código por ser usado em *multicore, manycore* e GPUs;
- Similar ao OpenMP (familiaridade);
- Fácil transição para o OpenMP 4.5 (futuro).

Antes de iniciar o processo de programação utilizando o OpenACC é recomendado que se faça uma análise do código a ser paralelizado seguindo o ciclo de desenvolvimento e análise de código conforme descrito:

1. Analisar o código para determinar quais as regiões mais prováveis que podem ser paralelizadas ou otimizadas.
2. Paralelizar o código iniciando com as regiões com o maior tempo de execução.
3. Otimizar o código para melhorar o tempo de execução observado a partir das alterações executadas.

Na Figura 2.4 é apresentado o ciclo de desenvolvimento e análise do código.

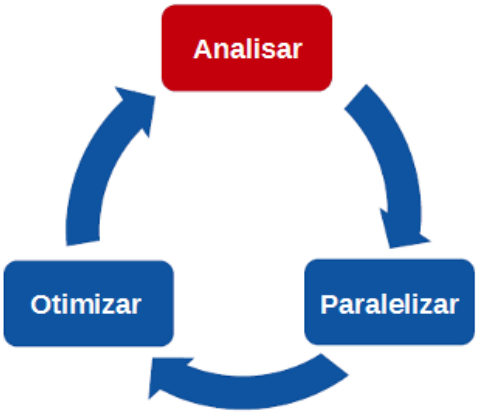


Figura 2.4: Ciclo de desenvolvimento e análise do código

O OpenACC pode ser utilizado em códigos programados em linguagens de programação C/C++ ou Fortran, deve-se atentar a sintaxe correta para tipo de linguagem de programação correta conforme a Figura 2.5.

C/C++

```
#pragma acc data [clause]
{
    code region ...
}
```

Fortran

```
!$acc data
    code region ...
!$acc end data
```

Figura 2.5: Sintaxe do OpenACC em C/C++ e Fortran

2.4. Diretivas e cláusulas

O modelo de programação usado no OpenACC é baseado em diretivas e cláusulas. As diretivas são comandos de instrução passadas pelo programador ao compilador. Cláusulas são os parâmetros adicionais atribuídos às instruções usadas nas diretivas.

As diretivas do OpenACC são muito parecidas com as diretivas do OpenMP. As diretivas são escritas na forma de **pragma**. Existem vantagens em usar diretivas, uma delas é o fato de o código precisar de pequenas modificações, as mudanças podem ser

feitas de forma incremental, um **pragma** de cada vez. Com o uso desse processo torna-se especialmente útil para fins de depuração, já que fazer uma única alteração permite identificar rapidamente um erro.

O uso do OpenACC pode ser desativado em tempo de compilação. Quando o suporte OpenACC está desabilitado na compilação, o **pragma** referente ao OpenACC é considerado um comentário, sendo ignorado pelo compilador. Com isso, um único código pode ser usado para compilar tanto um código com suporte a acelerador quanto uma versão sequencial.

2.4.1. 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).

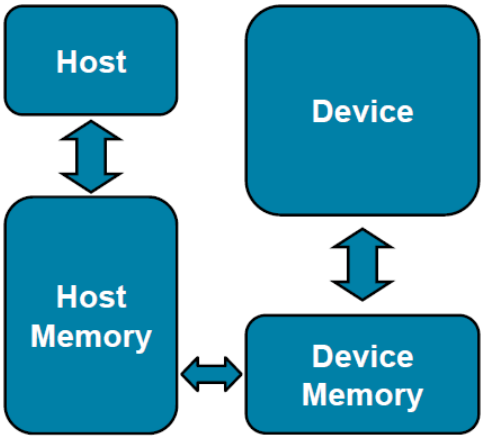


Figura 2.6: Modelo básico de movimentação de dados entre host e o acelerador (CHEN, 2017)

A movimentação de dados entre o *host* 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 *host* 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 **kernels** ou **parallel** e tem como principais caraterísticas:

- Define a região do código na qual os dados permanecem no acelerador;
- Define quais dados são compartilhados entre todos os *kernels* de uma região paralela;

- Realiza transferências de dados explícitas.

```
#pragma acc data [clause]
```

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 *host* 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 *host*.
copyout	Cria espaço para as variáveis listadas no dispositivo, mas não as inicia. No final da região, copia os resultados de volta para o *host* 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 2.1: Cláusulas da Diretiva Data

2.5. Diretiva parallel

A paralelização usando o construtor **parallel** identifica uma região de código que será paralelizada, quando executada em conjunto com a diretiva **loop**, o compilador gerará uma versão paralela do laço para o acelerador. Desta forma o compilador executa o paralelismo do código de forma direta.

```
#pragma acc paralell [clause-list]
```

Neste modo de programação, os laços que serão paralelizados precisam ser definidos no código, o compilador não consegue identificar os laços que precisam ser paralelizados, se a diretiva **loop** não for especificada não será feita a sua paralelização.

2.5.1. Cláusulas da diretiva Parallel

Algumas cláusulas são usadas para melhorar o desempenho da região a ser paralelizada, essas cláusulas permitem ter um maior controle e processamento dos laços.

Cláusula	Descrição
private	A cláusula privada especifica que cada iteração do laço tem a sua própria cópia das variáveis listadas
reduction	A redução é executada com as operações suportadas: + * max min
async	Retira as barreiras implícitas no final da região paralela

Tabela 2.2: Cláusulas da Diretiva Parallel

A cláusula **private** da construção **parallel** vai privatizar as variáveis listadas para cada *gang* (veremos mais detalhes na seção seguinte) na região paralela.

A cláusula **reduction** funciona de forma similar à cláusula **private** de forma que é gerada uma cópia privada das variáveis, mas existe uma redução ao final da região paralela de todas as cópias privadas em um único resultado final, que é retornado ao sair da região paralela. As operações de redução possíveis são soma, multiplicação, máximo e mínimo. O formato para a cláusula de redução é como a seguir:

```
#pragma acc parallel reduction(operator:variable)
```

2.6. Diretiva parallel loop

A construção **loop** fornece ao compilador informações adicionais sobre o próximo laço no código-fonte. A diretiva **loop** será vista nesta seção em conexão com a diretiva **parallel**, embora também seja válida com **kernels**.

Para fazer o paralelismo de vários laços, é necessário que cada laço seja acompanhado de uma diretiva **parallel loop**.

A diretiva **parallel loop** é uma afirmação do programador de que é seguro e desejável paralelizar o laço afetado. Isso depende se o programador identificou corretamente o paralelismo no código e removeu qualquer coisa no código que poderia tornar perigosa a sua paralelização. Se o programador afirmar incorretamente que o laço pode ser paralelo, e ele não pode, a aplicação resultante pode produzir resultados incorretos.

O programador identifica o paralelismo sem dizer ao compilador como explorar esse paralelismo. Isso significa que o código OpenACC pode ser portado para outros dispositivos além do dispositivo para o qual o código foi primeiramente desenvolvido,

porque detalhes sobre como paralelizar o código são deixados para o compilador decidir, ao invés de ser especificado explicitamente no código fonte.

Cada região **parallel loop** pode ter diferentes laços e cada laço pode ser paralelizado e otimizado de forma independente entre eles. Porém alguns cuidados devem ser tomados.

```
#pragma acc parallel loop
for (int i = 0 ; i < N; i++)
   a[i] = 0;
#pragma acc parallel loop
for (int j = 0 ; j < M; j++)
   b[j] = 0;
```

Exemplo 2.1: Diretiva Parallel Loop

Por exemplo, essa é a maneira recomendada de paralelizar vários laços. Tentando paralelizar múltiplos laços dentro da mesma região paralela pode ocasionar problemas de desempenho ou resultados inesperados.

2.6.1. Cláusulas da diretiva parallel loop

Como o OpenACC serve como uma linguagem para aceleradores genéricos existem três níveis de paralelismo que podem ser usados no OpenACC. Eles especificam o nível de paralelismo contidos na rotina, são chamados de **gang, worker e vector**. Uma *gang* é composta por um ou vários *workers*. Todos os *workers* de uma *gang* podem compartilhar os mesmos recursos, como memória cache ou processador.

Os níveis de paralelismo usado no OpenACC podem ser comparados ao níveis de execução usados na programação CUDA. Podendo assim admitir a relação entre eles: **gang = block**, **worker = warp** e **vector = threads**.

Cláusula	Descrição
gang	Particiona o laço entre as *gangs*
worker	Particiona o laço entre os *workers*
vector	Vetoriza o laço
seq	Não particiona o laço, que é executado sequencialmente

Tabela 2.3: Cláusulas da Diretiva Parallel Loop

Essas diretivas também podem ser combinadas em um laço específico. Por exemplo, um laço **gang vector** pode ser particionado entre *gangs*, cada uma delas com 1 *worker* 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.

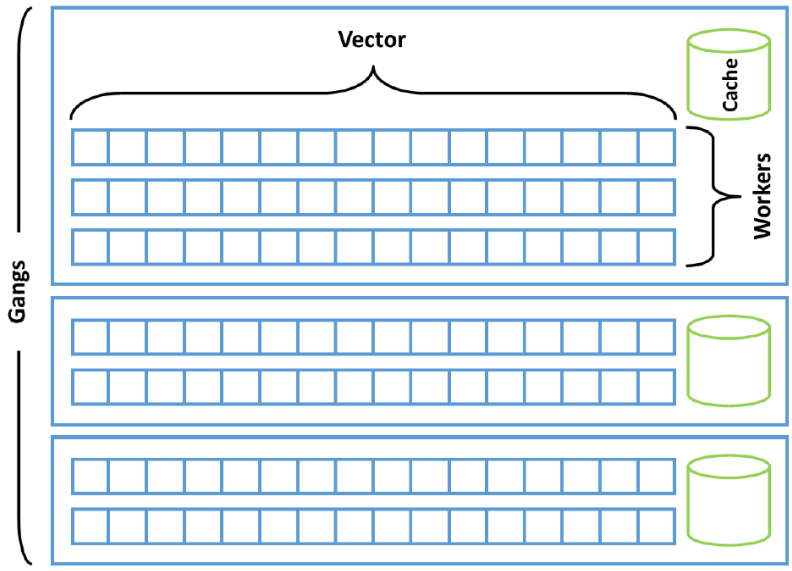


Figura 2.7: Gangs x Workers x Vector (OPENACC-STANDARD.ORG, 2015)

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 2.2: Cláusulas da Diretiva Parallel Loop

2.6.2. Cláusula reduction

A construção **parallel loop** possui uma cláusula com o mesmo nome **reduction** e com funcionamento similar a construção **parallel**, ou seja, uma cópia privada da variável é gerada para cada iteração do laco, mas é feita uma redução em um único resultado final, que é retornado da região.

Por exemplo, o valor máximo de todas as cópias privadas pode ser necessário ou talvez a soma. A redução pode apenas ser especificada em uma variável **escalar** e apenas as operações **+,*, max e min** e algumas operações bit a bit, tais como **&, |, ^, &&,** ||.

O formato da cláusula **reduction** é mostrado a seguir, onde *operador* deve ser substituído com a operação desejada e *variavel* deve ser substituída com a variável sendo reduzida.

```
reduction(operador:variavel)
```

Um exemplo pode ser visto a seguir:

```
{}
sum = 0.0;
#pragma acc parallel loop reduction(+:soma)
for( int i = 1; i < n-1; i++ )
    soma = A[i] + B[i];

   printf("Soma= %6f\n", soma);
}
```

2.6.3. Cláusula private

A cláusula **private** especifica que cada iteração do laço requer sua própria cópia das variáveis listadas. Por exemplo, se cada laço contiver uma matriz pequena e temporária denominada *tmp* usada durante seu cálculo, essa variável deverá ser tornada privada para cada iteração do laço, a fim de garantir resultados corretos. Se *tmp* não for declarada privada, as *threads* que executam diferentes iterações do laço podem acessar essa variável *tmp* compartilhada de maneiras imprevisíveis, resultando em uma condição de corrida e em resultados potencialmente incorretos. Abaixo está a sintaxe da cláusula privada.

```
private (variavel)
```

Há alguns casos especiais que devem entendidos sobre variáveis escalares em lacos. Primeiro, os iteradores do laco são privatizados por padrão, então eles não precisam ser listados na diretiva **private**. Segundo, a menos que seja especificado em contrário, qualquer variável escalar acessada dentro de um laco paralelo será definida *first private* por padrão, o que significa que uma cópia será feita para cada iteração do laço e que terá um valor inicial igual ao que havia na variável escalar ao entrar na região. Finalmente, quaisquer variáveis (escalares ou não) que são declaradas dentro de um laço em C ou C++ serão feitas privadas para as iterações do laço por padrão.

Nota: a construção **parallel** também tem uma cláusula **private** que vai privatizar a lista de variáveis para cada *gang* na região paralela.

2.7. 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 2.3: Diretiva Routine

Quando a rotina *foo* é chamada a partir do código do *host*, ele será executado no *host*, incrementando os valores do *host*. 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.

2.8. Diretiva kernels

Quando é utilizada a diretiva **kernels**, isto significa que é informado ao compilador que existem regiões do código que podem ser paralelizadas e que o compilador será o responsável por identificar quais são as regiões e qual a estratégia que será utilizada. A estratégia definida pelo compilador pode ser extrair o máximo de paralelismo do código ou executar somente o mínimo de paralelismo.

Com uso da diretiva **kernels**, o compilador analisará o código e apenas paralelizará quando tiver certeza de que é seguro fazê-lo. Em alguns casos, o compilador pode não ter informações suficientes para determinar se é seguro paralelizar um laço, neste caso essa paralelização não será feita. Os passos do processo de compilação do código usando a diretiva **kernels** são:

- Analisar o código para identificar regiões de paralelismo
- Se encontrado, identificar quais dados devem ser transferidos
- Criar um *kernel*
- Movimentar os dados para dispositivo

```
#pragma acc kernels [clause-list]
```

2.9. Diretiva kernels vs. diretiva parallel

Um dos maiores pontos de confusão para os novos programadores de OpenACC é o motivo pelo qual a especificação possui as diretivas **paralell** e **kernels**, que parecem fazer a mesma coisa. Embora ela estejam fortemente relacionadas, existem diferenças sutis entre elas, que possuem características distintas para cada tipo de aplicação e são usadas de acordo com a necessidade de execução do código.

Existem códigos que são fáceis de alterar e obtém melhor desempenho usando a diretiva **parallel**, porém existem códigos que possuem grande dificuldade de alteração não sendo possível usar diretiva **parallel**, neste caso é usado a diretiva **kernels**, pois as alterações são as mínimas possíveis.

A construção **kernels** fornece ao compilador uma margem de manobra maior para paralelizar e otimizar o código da forma que ele considerar adequada para o tipo de acelerador utilizado, mas também depende muito da capacidade do compilador de paralelizar automaticamente o código. Como resultado, o programador pode ver diferenças de desempenho e de nível de otimização em compiladores diferentes.

A diretiva **parallel loop** é uma afirmação do programador de que é seguro e desejável paralelizar o laço afetado. Isso depende do programador ter identificado corretamente o paralelismo no código e removido qualquer coisa no código que não seja segura para

paralelizar. Se o programador afirmar incorretamente que o laço pode ser paralelizado, o programa resultante poderá produzir resultados incorretos. Em outras palavras: a construção **kernels** pode ser pensada como uma dica para onde o compilador deve procurar paralelismo, enquanto a diretiva **paralell** é uma afirmação para o compilador onde há paralelismo.

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 *restrit* 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 *host* 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 *host* e o dispositivo entre os dois laços, o que pode resultar em menor movimentação de dados por padrão.

Nos Exemplos 2.4 e 2.5 usamos um trecho de um código que possui dois laços comparando o processo de paralelização utilizando as diretivas de **kernels** e **parallel**.

Note-se que no processo de execução usando a diretiva **kernels** são gerados dois *kernels*, um para cada laço. Existe um barreira entre os laços, deste modo o segundo laço só será iniciado quando o primeiro laço terminar.

No processo de execução usando a diretiva **parallel** é gerado um único *kernel*. Não existe barreira entre os laços, assim podem ser executados de forma independente.

```
#pragma acc kernels
{
 for (i=0; i<n; i++)
  a[i] = 3.0f*(float)(i+1);
 for (i=0; i<n; i++)
  b[i] = 2.0f*a[i];
}
```

Exemplo 2.4: Diretiva kernels

```
#pragma acc parallel
{
 #pragma acc loop
 for (i=0; i<n; i++)
 a[i] = 3.0f*(float)(i+1);
 #pragma acc loop
 for (i=0; i<n; i++)
 b[i] = 2.0f*a[i];
}
```

Exemplo 2.5: Diretiva parallel

2.10. 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* (gang, work ou vector) executando a operação nela contida simultaneamente. O uso da diretiva **atomic** é às vezes 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 2.6: 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.

2.11. Cláusula device_type

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 arquiteturas específicas.

2.12. Cláusula vector_length

Outra cláusula importante é a **vector_length**, esta cláusula é utilizada para especificar o tamanho do vetor que será usado no laço.

No exemplo abaixo foi especificado um comprimento de vetor, dependendo do tipo de acelerador que será usado.

```
#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 2.7: Suporte a múltiplos aceleradores

Neste 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.

2.13. Cláusula tile

Existe um novo recurso disponível no OpenACC 2.0, 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 2.8: 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.

2.14. 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.

```
#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++)...
```

Exemplo 2.9: Cláusula collapse

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.

2.15. Compilação

Antes de compilar qualquer código é importante saber quais dispositivos aceleradores estão configurados para uso no sistema. Existem alguns comandos que fornecem informações de modelos e características desses dispositivos.

Para aceleradores da NVIDIA existes os comandos *nvidia-smi* e *nvidia-settings*, estes comandos fornecem de informações de configuração como: modelo, cpus, cuda core, memória. Com o compilador da PGI também existe um comando chamado *pgaccelinfo* que fornece as principais características dos aceleradores instalados no sistema.

```
# nvidia-smi -q | grep "Product Name"
    Product Name                    : Quadro K420
    Product Name                    : Tesla K80
    Product Name                    : Tesla K80

# pgaccelinfo | grep "Device Name"
    Device Name:                   Tesla K80
    Device Name:                   Tesla K80
    Device Name:                   Quadro K420
```

Para compilar os códigos feitos em OpenACC, gerando código para execução em GPUs, é necessário o uso de compiladores que suportem OpenACC. A PGI (Portland Group), tem um versão disponibilizada para uso público sem a necessidade de licença, também existe uma versão com uso de licença que permite ter acesso à equipe de suporte da PGI.

A Cray também tem seu compilador o CCE 8.6.5, para usá-lo é necessário a aquisição de uma licença, não existe um licença para uso público.

Nos exemplos apresentados neste minicurso usaremos o compilador da PGI disponibilizado para a comunidade, o compilador pode ser baixado do sítio da PGI através no endereço https://www.pgroup.com/products/community.htm.

Para compilar códigos em C usaremos o comando *pgcc*, e para compilação de códigos em C++ usar o comando *pgc++*. Algumas parâmetros básicos devem ser usados durante a execução dos comandos *pgcc* e *pgc++*. Esses parâmetros definem em qual dispositivo o código é executado de acordo com a arquitetura. Relação dos principais parâmetros usados no comando *pgcc*:

Parâmetro	**Descrição**
-fast	faz a otimização do código
-acc	habilita o uso de diretivas OpenACC
-Minfo=accel	informações sobre quais partes do código foram aceleradas
-Minfo=opt	informações sobre todas as otimizações de código
-Minfo=all	informações sobre todas as saídas de código
-ta=host	compila o código em modo serial
-ta=multicore	compila o código usando *threads* em CPU
-ta=nvidia	compila o código usando NVIDIA

Tabela 2.4: Parâmetros de Compilação pgcc

Alguns exemplos de uso do compilador PGI e seus parâmetros básicos.

```
$ pgcc -acc -ta=nvidia -Minfo=accel main.c
$ pgc++ -acc -ta=nvidia -Minfo=accel main.cpp
```

Uma outra alternativa é o uso do compilador GNU GCC a partir da versão 6 o compilador tem suporte ao OpenACC. Como o GNU GCC é um compilador *opensource* não requer licença para uso. Para usar o GNU GCC especificar o parâmetro *-fopenacc*.

```
$ gcc -fopenacc main.c
```

2.16. Exemplos

Após a introdução do conceito e das principais diretivas usadas no OpenACC, veremos alguns exemplos da aplicações dessas diretivas, como elas se comportam e quais as melhores opções de uso das diretivas para gerar maior ganho.

Os códigos foram executados em um servidor com dois processadores Intel Xeon E5-2609 (2,40 GHz, 4 núcleos cada, cache de 10 MB), com 128 GB de memória compartilhada, discos locais de alta velocidade SSD (Solid-State Drive) e um acelerador NVIDIA Tesla K80 (24 GB de memória, 4992 CUDA cores). O sistema operacional usado foi a

versão 7.3 da distribuição Centos Linux de 64 bits. Todos os códigos foram compilados usando o PGI Community Edition Version 19.4.

2.16.1. Cálculo de Pi

Iniciaremos com o exemplo básico do cálculo do número Pi. O valor de Pi é definido pela relação entre o perímetro de uma circunferência e seu diâmetro, para a maioria dos cálculos simples é comum usar a aproximação do valor de Pi para 3,1415. Em computação existem algoritmos que podem ser utilizados para o cálculo aproximado do Pi, como: Gauss-Legendre e Monte Carlo. A seguir é apresentada uma implementação simples para o cálculo sequencial (serial) de Pi.

```
#include <stdio.h>
#define N 1000000000

int main(void) {
   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 2.10: Cálculo de Pi sequencial

A execução do cálculo de Pi dentro do laço na versão sequencial será feito por uma *thread*, independente de quantidade de processadores que existam no sistema.

Podemos paralelizar o código em OpenMP adicionando a linha **#pragma omp parallel for reduction(+: pi)** antes do laço, desta forma pode-se utilizar mais de uma *thread* para o cálculo.

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

Exemplo 2.11: Cálcudo de Pi com OpenMP

Para que o cálculo seja executado pelo acelerador usaremos a diretiva **parallel** do OpenACC. Da mesma forma que foi usado no OpenMP, adicionaremos a linha **#pragma acc parallel loop reduction(+: pi)** antes do laço.

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

Exemplo 2.12: Cálculo de Pi com OpenACC

Para avaliar o tempos de execução do cálculo de Pi, foram executados os códigos sequencial, com OpenMP e com OpenACC. Para o cálculo em OpenMP foram utilizadas 16 *threads*, a quantidade máxima de processadores no servidor.

O tempo de execução sequencial do cálculo de Pi foi de 5,6 segundos, o tempo com OpenMP foi de 0,60 segundos e com OpenACC foi de 0,15 segundos.

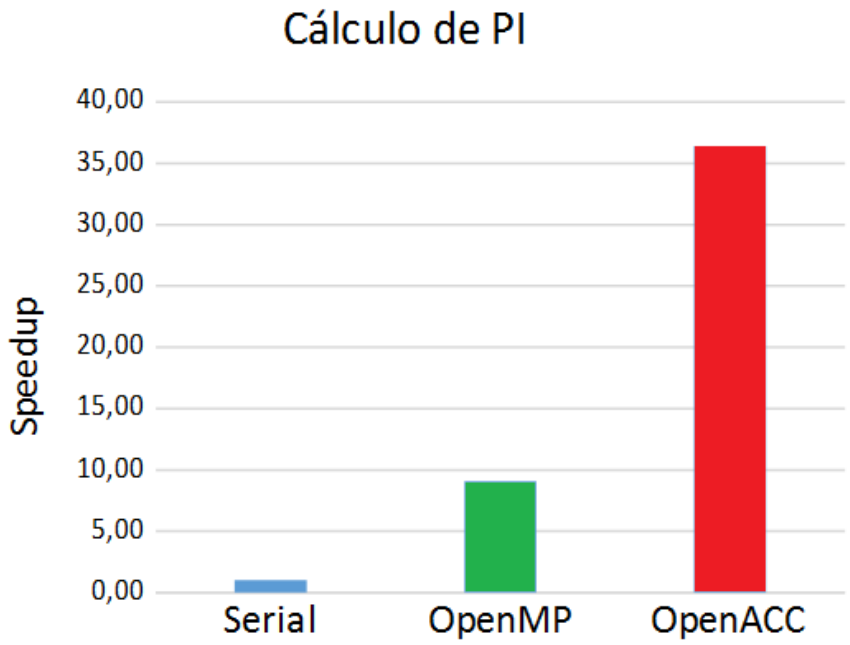
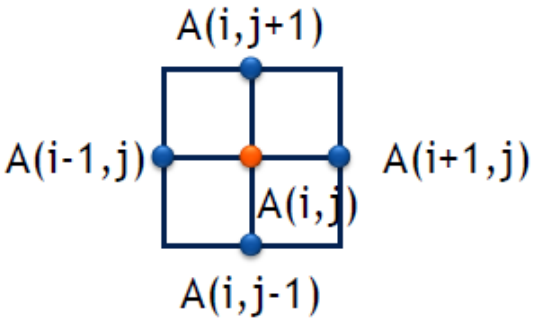


Figura 2.8: Seepdup - Cálculo do valor de Pi

O *speedup* encontrado usando programação OpenMP foi de 9,10 em comparação a execução em sequencial, enquanto que usando OpenACC o *speedup* foi de 36,40 em relação ao executado em sequencial. Os resultados são apresentados na Figura 2.8

2.16.2. Método Jacobi

O Método de Jacobi é um procedimento iterativo para a resolução de sistemas lineares. Converte 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: $\nabla^2 f(x,y) = 0$.



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

A seguir é apresentada uma implementação sequencial (serial) para o cálculo da temperatura da placa usando o método Jacobi.

```
#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];
double A[ROWS+2][COLUMNS+2];

void initialize();
void track_progress(int iter);

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

    int i, j;
    int max_iterations=1000;
    int iteration=1;
    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);
}

void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            A[i][j] = 0.0;
        }
    }

    for(i = 0; i <= ROWS+1; i++) {
        A[i][0] = 0.0;
        A[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    for(j = 0; j <= COLUMNS+1; j++) {
        A[0][j] = 0.0;
        A[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
```

Exemplo 2.13: Método de Jacobi Sequencial

O primeiro laço dentro do *while* de convergência calcula o novo valor para cada elemento com base nos valores atuais de seus vizinhos. Armazenando em uma matriz

temporária, garantindo que todos os valores sejam calculados usando o estado atual de **A** antes que **A** seja atualizado. Como resultado, cada iteração do laço é completamente independente uma da outra.

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

A execução do cálculo da temperatura dentro dos laços serão feitos por uma *thread*, independente de quantidade de *threads* que existam no sistema.

Podemos paralelizar o código em OpenMP adicionando as linhas **#pragma omp parallel for** antes do primeiro laço e a linha **#pragma omp parallel for reduction(max:dt)** antes do segundo laço, desta forma pode-se utilizar mais de uma *thread* para o cálculo.

```
        #pragma omp parallel for
        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 omp parallel for 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];
            }
        }
```

Exemplo 2.14: Método de Jacobi com OpenMP

Para que o cálculo seja executado pelo acelerador usaremos a diretiva **parallel** do OpenACC. Da mesma forma que foi usado no OpenMP, adicionar a linha **#pragma acc parallel loop**, e a linha **#pragma acc parallel loop reduction(max:dt)** no segundo laço.

```
        #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];
            }
        }
```

Exemplo 2.15: Método de Jacobi com OpenACC - Versão 1

Para avaliar os tempos de execução dos cálculos da temperatura, foram executadas as versões sequencial, com OpenMP e com OpenACC. Para o cálculo em OpenMP foram utilizadas 16 *threads*, quantidade máxima de processadores no servidor.

Foram observados os seguintes tempos de execução: o código sequencial teve um tempo total de 11 segundos, sendo que com OpenMP o tempo de execução foi de 4,30 segundos e com OpenACC o valor medido foi de 10 segundos.

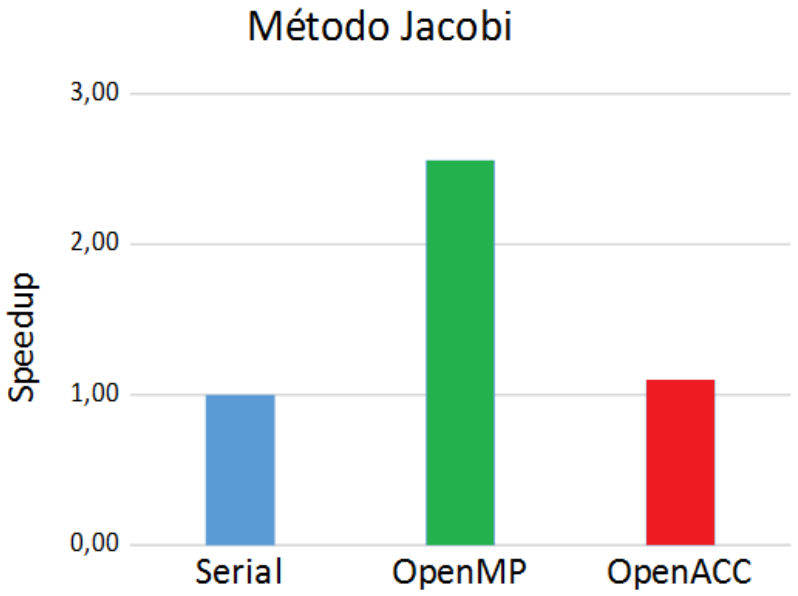


Figura 2.9: Seepdup - Método Jacobi

O *speedup* encontrado na execução do código com OpenMP foi de 2,56 em comparação à execução em sequencial, enquanto que usando OpenACC o *speedup* foi de 1,10. Os resultados são apresentados na Figura 2.9.

Como visto, o resultado do *speedup* encontrado usando OpenACC foi quase idêntico aos resultados encontrados na execução do código sequencial. Isso ocorre porque 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 *host*.

Para resolver este problema é necessário fazer a cópia da matriz para o acelerador de modo que não seja mais necessário gravar as informações no *host* toda vez que for realizada uma operação pelo acelerador.

Usaremos a diretiva **data** do OpenACC. Adicionar a linha **#pragma acc data copy(A) create (Anew)** antes dos dois laços para fazer a cópia da matriz para a memória do acelerador.

```
#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];
       }
   }
```

Exemplo 2.16: Método de Jacobi com OpenACC - Versão 2

O total de tempo para a execução sequencial foi de 11 segundos, sendo que o tempo total com OpenMP se manteve em 4,30 segundos e o tempo de execução com OpenACC foi reduzido para 0,80 segundos.

O *speedup* para a execução com OpenMP continou em 2,56 em comparação a execução sequencial, enquanto que usando OpenACC o *speedup* aumentou para 13,75. Os resultados são apresentados na Figura 2.10.

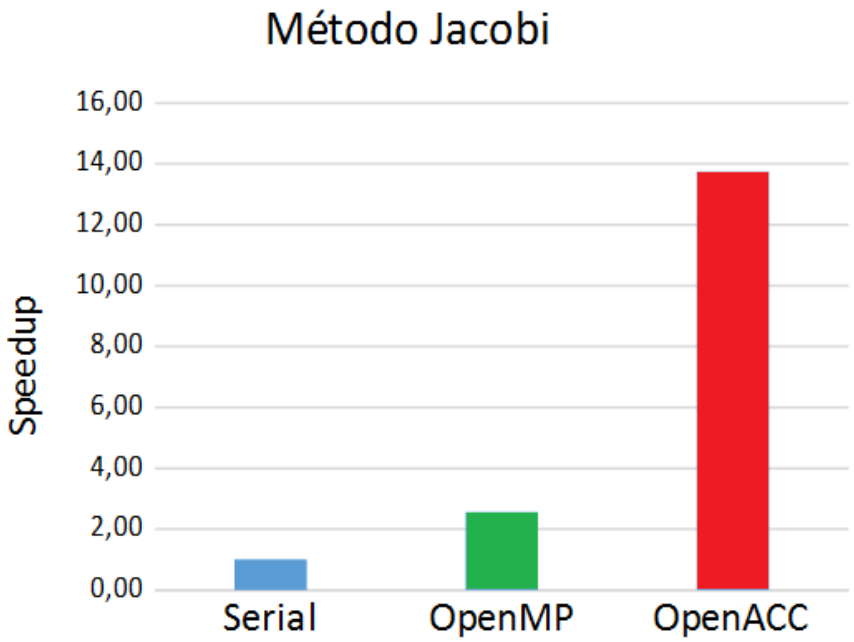


Figura 2.10: Seepdup - Método Jacobi

2.17. Conclusão

O modelo de programação OpenACC foi desenvolvido pelos principais fabricantes de hardware e software do mercado, com o objetivo de simplificar a programação paralela, tornando possível a portabilidade do código.

Com o uso do OpenACC é possível atingir alto níveis de paralelismo usando arquitetura baseada em aceleradores. Entre suas principais características se destacam:

- O OpenACC é fácil de usar;
- Usa uma abordagem baseada em diretivas de compilação;
- Em alguns casos são feitas pequenas alterações no código;
- O código pode ser implementado em qualquer acelerador.

Referências

A., Z. L. F.; M., M. *Arquitetura e programação de GPU nvidia*. [S.l.]: UNICAMP, 2012.

ABBOTT, S. *Advanced OpenACC*. [S.l.]: NVIDIA Corporation, 2017.

CHEN, S. *Introduction to OpenACC*. [S.l.]: Research Computing Services Information Services and Technology Boston University, 2017.

CORREA, J. C.; SILVA, G. P. Analysis and performance evaluation of parallel BLAST. *I. J. Comput. Appl.*, v. 20, n. 2, p. 112–122, 2012.

COSTA, E. B.; SILVA, G. P.; TEIXEIRA, M. Avaliação de desempenho do montador daligner em arquiteturas manycore. In: *WSCAD 2018 - WCH ()*. São Paulo - SP, Brazil: [s.n.], 2018.

LARKIN, J. *Introduction to OpenACC*. [S.l.]: NVIDIA, 2018.

OPENACC-STANDARD.ORG. *OpenACC Programming and Best Practices Guide*. [S.l.]: OpenACC-Standard.org, 2015.

RAHMAN, R. *Intel Xeon Phi Coprocessor Architecture and Tools The Guide for Application Developers*. [S.l.]: Apress Open, 2013.

SILVA, G. P. *Programação Paralela com MPI Um Curso Introdutorio*. [S.l.]: Amazon, 2018.