OpenACC, OpenCL e OpenARC

Um estudo sobre ferramentas abertas para programação em Sistemas Heterogêneos

Arthur Prado De Fazio

Orientador: Alfredo Goldman Vel Lejbman

Agradecimentos

Aos meus pais, pelo apoio, pelo afeto e pelo amor incondicionais, sem os quais eu não teria chegado até aqui. Foram vocês que pavimentaram esse caminho. Obrigado à minha mãe Denise, que sempre esteve sempre próxima dos meus estudos, e se certificou de que tudo sempre corresse da maneira mais tranquila possível. Obrigado ao meu pai Carlos, que fez questão de me oferecer a melhor educação possível e permitiu que eu sonhasse tão alto quanto eu quisesse.

Obrigado ao meu avô Airton, de quem tenho muitas saudades.

Ao Denis, que me ouviu verdadeiramente e me devolveu a minha força. Não tenho palavras o suficiente pra te agradecer.

Aos meus amigos, com os quais os percalços dessa caminhada são insignificantes perto da felicidade que é estar junto a eles.

Ao meu orientador Alfredo, que tornou este trabalho possível, e me aconselhou ao longo de todo esse processo.

Sumário

1 Introdução

Em sistemas de programação paralela, há diferentes formas de paralelismo, fornecidas por diferentes tipos de *hardware*. Em particular, cada um desses sistemas apresenta uma forma própria para programação, oferecida por vendedores, por exemplo. Podemos inclusive construir um sistema composto de diferentes componentes de *hardware*, cada um com formas diferentes de programação nativa; esses sistemas chamam-se sistemas heterogêneos. Esses sistemas de programação, utilizados em programação de alto desempenho, apresentam dificuldades quanto a sua programação justamente por conta do manejo de diferentes dispositivos que os compõe. Alguns desses entraves são a portabilidade quanto ao código, visto que cada dispositivo pode exigir um estilo de programação, a portabilidade quanto ao desempenho, já que trechos semelhantes podem ser executados mais ou menos rapidamente a depender do tipo de dispositivo, e a necessidade de conhecimento do *hardware* em que se está programando, para que o programa se beneficie das particularidades de cada componente.

Este trabalho apresenta três ferramentas abertas, independentes de vendedores, para programação em sistemas heterogêneos. São elas o OpenACC, o OpenCL e o OpenARC.

Primeiro tratamos do OpenACC, um arcabouço de alto nível para programação paralela. Trata-se de um modelo de programação que utiliza diretivas de compilação (ou pragmas) para realizar delegação de execução a dispositivos, gerenciamento do paralelismo e gerenciamento de memória nos dispositivos. Nesse capítulo, apresentamos alguns conceitos de programação em sistemas heterogêneos, explicamos as diretivas do OpenACC (com exemplo de uso), as funcionalidades e os tipos de paralelismo a que esse arcabouço expõe quem está programando.

Em seguida, tratamos do OpenCL, um outro arcabouço para programação em sistemas heterogêneos. Essa ferramenta, também aberta, expõe a pessoa programadora a muitos mais detalhes de *hardware* do que o OpenACC, como o gerenciamento dos dispositivos e das tarefas que delegamos a eles. Nesse segundo capítulo, discutimos o modelo que a especificação determina do OpenCL; ao longo dessa

explicação, expomos as principais funções que a API desse arcabouço relacionados a cada elemento do modelo, fornecendo exemplos de uso. Explicada a API, tratamos do OpenCL C, a linguagem com que são escritas as funções delegadas a dispositivos no OpenCL (os *kérneis*) e fornecemos um exemplo completo de uso dessa ferramenta, comparando com um programa equivalente em OpenACC e expondo as diferenças entre modelos de programação e as dificuldades que cada um deles apresenta à pessoa programadora.

Depois, nos ocupamos de discutir o OpenARC. O OpenARC é um transpilador de código aberto, desenvolvido no Oak Ridge National Laboratory, que transforma código escrito em OpenACC ou OpenMP em CUDA ou OpenCL a ser executado em uma de 8 opções de plataforma. Discutimos em primeiro lugar seu processo de compilação, ressaltamos algumas de suas principais passadas e transformações. Em seguida, exibimos um programa escrito em OpenACC e o programa produzido pelo OpenARC a partir do primeiro, comparando um com o outro e a maneira como o OpenARC realiza essa transformação de modo a manter um código portável para mais de um tipo de dispositivo.

Finalmente, no capítulo 5, exibimos experimentos que comparam o desempenho de programas escritos em OpenACC e compilados com o NVC com o dos programas compilados a partir do código em OpenCL gerado pelo OpenARC.

2 OpenACC

Em um ambiente de programação paralela, podemos escrever programas paralelos seguindo, grosso modo, 3 abordagens. O primeiro e mais alto nível faz uso de bibliotecas; quando precisamos de tarefas muito frequentemente realizadas, como operações de álgebra linear, essa é uma boa opção. Nesse caso, todo trabalho de paralelização foi feito por quem implementou a biblioteca, de tal maneira que clientes não precisam se preocupar com isso. Num outro extremo, podemos utilizar as próprias linguagens de programação para programar aplicações paralelas, nós mesmos; com isso, explicitamos threads, utilizamos semáforos ou mutexes, e número de threads, *cores* e assim por diante. Essa abordagem, nos dá um programa feito sob medida, extremamente otimizado, para a plataforma em que ele será executado.

Se de um lado temos conveniência e pouca flexibilidade, do outro somos recompensados com alta personalização e eficiência ao custo de precisar entender detalhes de hardware.

Entre essas duas abordagens está o uso de diretivas de compilação, que permitem que construamos nossas próprias soluções, pensadas para nossos propósitos, com portabilidade e delegando os detalhes de nível mais baixo para serem resolvidos pelo compilador. OpenACC é um modelo de programação paralela para sistemas com aceleradores baseado em diretivas que oferece suporte à execução paralela, otimização de laços e gerenciamento de memória.

A especificação do OpenACC é determinada pela OpenACC Organization, e sua primeira versão data de novembro de 2011. O primeiro compilador a oferecer suporte para o OpenACC foi o OpenUH, um compilador de código aberto desenvolvido por 7 pessoas do HPCTools Group da Universidade de Houston; mais especificamente, a primeira versão do OpenUH a oferecer suporte ao OpenACC foi disponibilizada em janeiro de 2014. Atualmente, a OpenACC Organization tem como missão expandir o uso da programação paralela em aceleradores dentro da comunidade científica e entre desenvolvedores. Para isso, além de escrever a especificação do OpenACC, a organização trabalha junto a comitês de padrões de linguagens de programação focados em interoperabilidade de desempenho, orga-

niza as chamadas Open Hackatons e eventos de treinamento para universidades, centros de pesquisa e empresas.

Este capítulo apresenta esse arcabouço e a maneira como ele expõe a quem programa o controle sobre paralelismo, gerenciamento de memória e sincronização de dados no contexto de programação para sistemas heterogêneos. Fornecemos aqui exemplos de uso e comparamos seus níveis de paralelismo. A escrita e os exemplos deste capítulo foram baseados no que está em [22], [CJ17] e [Far17].

2.1 Portabilidade

Uma das vantagens do uso do OpenACC é a possibilidade de se escrever um único código fonte que pode ser executado em diferentes sistemas, ou mesmo dentro de um sistema composto por componentes distintos entre si. Um programa escrito em OpenACC pode ser executado em POWER, Sunway, x86 CPU, AMD GPU, NVIDIA GPU e pezy-sc. Além disso, o uso de pragmas (explicados a seguir) permite que as diretivas do OpenACC possam ser ignoradas (e vistas como comentários) caso o compilador não as conheça e que o programa seja sequencialmente.

2.2 Conceitos de OpenACC

Um *pragma* em C/C++ é uma instrução ao compilador sobre como ele deve lidar com um trecho de código. Compiladores que não a entenderem podem ignorá-la.

Para utilizar pragmas do OpenACC escrevemos `#pragma acc *diretivas* *cláusulas*`, sendo que diretivas indicam comandos para que o compilador altere seu código, cláusulas especificam ou dão mais detalhes a respeito das alterações indicadas pelas diretivas. Podemos pensar que cláusulas estão para diretivas assim como argumentos estão para funções.

As diretivas do OpenACC podem ser divididas em três grupos:

- Diretivas de computação indicam blocos de código cuja execução deve ser delegada a um acelerador e executados em paralelo. São diretivas de computação `parallel`, `kernels`, `routine` e `loop`
- Diretivas de gerenciamento de memória controlam alocação de memória no acelerador e movimento de dados entre a máquina anfitriã e o acelerador. As diretivas para gerenciamento de memória são `data`, `update`, `cache`, `atomic`, `declare`, `enter data` e `exit data`
- Diretivas de sincronização permitem o controle da ordem da execução de tarefas, dentre elas se destaca `wait`, para esperar por uma ou mais tarefas podemos utilizar a diretiva.

O comportamento de diretivas pode ser modificado com o uso de cláusulas, que também podem ser divididas em três grupos. Nem toda cláusula pode ser utilizada com qualquer diretiva. Elas podem ser grosso modo classificadas em:

- Cláusulas de gerenciamento de dados estabelecem acesso e transferência de dados para certas variáveis. São cláusulas de para gerenciamento de dados `default`, `private`, `firstprivate`, `copy`, `copyin`, `copyout`, `create`, `delete` e `deviceptr`

- Cláusulas de gerenciamento de paralelismo modificam distribuição de trabalho nos dispositivos, especificando a organização entre *threads* diferente do comportamento padrão que seria escolhido pelo compilador. Essas cláusulas estabelecem, agrupamento de *threads*, o que por sua vez determina o que é compartilhado entre elas. São cláusulas desse tipo `auto`, `seq`, `gang`, `worker`, `vector`, `tile`, `num_gangs`, `num_workers` e `vector_length`

- Cláusulas de controle de fluxo estabelecem a maneira como a execução deve ocorrer. São cláusulas desse tipo `if`, `if_present`, `reduction`, `independent`, `async`.

2.3 Diretivas para controle de paralelismo

Diretiva `kernel`

Tanto `kernel` quanto `parallel` indicam o despacho de um trecho de código para que um acelerador (como uma GPU ou uma FPGA) o execute. A diretiva `kernel` indica ao compilador que a execução do bloco de código ao qual ela diz respeito deve ser executada pelo acelerador, se possível. Os detalhes exatos quanto à paralelização do código e a corretude desse código ficam a cargo do compilador.

Diretiva `parallel`

Cria um ambiente para paralelismo, com a criação de gangues (a serem explicadas em 2.10) que executarão igualmente. Para execução em CPUs essa diretiva gera *threads*; para CUDA, *thread blocks*; e para OpenCL, grupos de trabalho. Em resumo, essa diretiva permite que mais de uma unidade de processamento execute um trecho de código, gerando as entidades que executarão o código.

Os detalhes do uso dessa diretiva ficam a cargo do programador, e não do compilador. Com seu uso, quem programa é responsável pela especificação do que deve ser executado em paralelo pelo acelerador. Nesse sentido, essa diretiva dá mais controle ao programador do que `kernels`.

Abaixo, o bloco de código entre `{}` será executado de forma redundante, isto é, o mesmo código será executado executado por um certo número de gangues.

```
#pragma acc parallel
{
    for (int i = 0; i < N; i++)
        a[i] += i;
}
```

Assim o *mesmo* laço será executado sequencialmente várias vezes em gangues diferentes. Usualmente não é isso que queremos, gostaríamos que cada iteração desse laço seja feita em paralelo, por um elemento de execução.

A maneira exata como o código acima executaria fica a cargo do compilador. Nesse sentido, alguns compiladores podem gerar programas que geram resultados esperados enquanto outros não. Por exemplo, no trecho de código acima poderia ocorrer que uma entrada do vetor `a` seja somada mais de uma vez com o valor de `i`.

Diretiva `loop`

Ela pode ser utilizada dentro das diretivas `kernels` ou `parallel`. Dentro da diretiva `parallel`, ela indica que cada iteração de um laço pode ser executada independente e paralelamente. Dentro da diretiva `kernels`, ela explicita um paralelismo que o compilador deveria descobrir, caso `loop` não estivesse lá.

```
#pragma acc parallel
{
    #pragma acc loop
    for (int i = 0; i < N; i++)
        ...
}
```

Com o código acima, instruímos a criação de um ambiente paralelo e depois dizemos que queremos dar a cada unidade de paralelismo deve executar uma iteração do laço.

Como, esse padrão é muito recorrente, há uma abreviação pra ele:

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

Vale ressaltar que não se recomenda posicionar laços diferentes numa mesma região delimitada por uma diretiva `parallel`. Isso porque pode ser que o compilador decida gerar um código que será executado de uma forma que o programador não desejava executar. Considere, o seguinte trecho de código:

```
#pragma acc parallel {
    #pragma acc parallel loop
    for (int i = 0; i < N; i++)
        a[i] *= 2;

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

Pode ser que o compilador decida que os dois laços devem ser executados em paralelo. Isto é, o primeiro laço poderia ser executado simultaneamente ao primeiro, quando na realidade desejava-ser que o primeiro laço deveria ser executando antes do segundo, cada um deles sendo executado paralelamente, pois utiliza-se `#pragma acc parallel loop`.

A uma diretiva `parallel loop`, podemos acrescentar uma cláusula para redução. Uma redução é uma cláusula que indica que vamos produzir um único valor a partir de várias parcelas calculadas independentemente umas das outras utilizando um operador binário. Nesses casos, não só o cálculo de cada uma das parcelas como também a aglutinação desses vários valores pode ser paralelizada. Por exemplo. Digamos que temos uma matriz a, $m \times p$, e uma matriz b, $p \times n$. A multiplicação delas será uma matriz c de dimensões $m \times n$ que pode ser calculada de acordo com

```
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < p; k++)
            c[i][j] += a[i][k] * b[k][j];
```

Note que `c[i][j]` é a soma de p valores calculados independentemente. Assim, podemos computar cada uma das parcelas `a[i][k] * b[k][j]` e juntá-las com uma soma. Em OpenACC, isso é feito da seguinte forma:

```
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) {
```

```
3           float tmp = 0.0;
4           #pragma acc parallel loop reduction(+:tmp)
5           for (int k = 0; k < p; k++)
6               tmp += a[i][k] * b[k][j];
7           c[i][j] = tmp;
8       }
```

A cláusula `reduction(+:tmp)` indica que `tmp` deve ser uma redução de acordo com a soma. Além da soma, em OpenACC podemos fazer reduções utilizando as seguintes operações: `*`, `max`, `min`, `&`, `|`, `&&`, `||`.

2.4 Gerenciamento de memória utilizando o OpenACC

Conforme já dito, o OpenACC é um modelo programação que foi criado com o propósito de facilitar a programação em ambientes em sistemas heterogêneos de computação, isto é, ambientes em que mais de um tipo de dispositivo contribui para a execução de uma aplicação. Nesses ambientes, um determinado elemento computacional só pode manipular dados que estejam em sua memória, de modo que só podemos delegar um determinado trecho de código uma vez que o elemento anfitrião, que normalmente é uma CPU, mova esses dados para esse dispositivo. Assim, ambientes de programação para sistemas desse tipo precisam oferecer ferramentas para gerenciamento de memória eficiente. Por exemplo, se estamos programando com GPUs Nvidia, a plataforma oferece a *Cuda Unified Memory*, um espaço único de endereços que permite que quem está programando possa encarar a memórias de sua GPU e CPU como se estivessem juntas e acessíveis aos dois componentes, apesar de serem separadas e haver transferência de dados entre elas. Nesse caso, o sistema operacional e os *drivers* cuidam do movimento, sem que o programador precise se preocupar.

Apesar da conveniência de se utilizar a *Cuda Unified Memory*, ela apresenta algumas inconveniências; por exemplo, o programador frequentemente consegue desempenho melhor ao explicitar quais dados e quando devem ser transferidos e em qual direção isso deve ser feito; além disso, alocação e liberação de memória levam tempo; por fim, a memória gerenciada não permite transferência assíncrona de dados e só está disponível nas GPUs da Nvidia.

Apesar de oferecer controle sobre a alocação nos e movimento de dados entre os dispositivos, o OpenACC especifica alguns aspectos do gerenciamento de memória que são feitos automaticamente. Um deles determina que variáveis de controle de loop, variáveis declaradas dentro de um bloco de código e variáveis declaradas dentro de uma função são privadas (`private`), o que significa que haverá

uma cópia dela por dispositivo de execução, assim pode não haver consistência entre o valor dessas variáveis em diferentes elementos de programação. Um outro aspecto que fica determinado pela especificação do OpenACC é que variáveis que são utilizadas dentro de regiões paralelas também tem uma cópia por elemento de execução. Elas podem ser `firstprivate`, o que significa que o valor delas no início da execução da região paralela é determinado fora da região paralela, ou simplesmente `private`.

Por fim, quando falamos sobre gerenciamento de memória, a especificação do OpenACC define dois conceitos centrais: o de região de dado (*data region*) e o de tempo de vida de dados (*data lifetime*). Uma região de dados é o escopo (léxico ou dinâmico, conforme explicaremos mais adiante) definidos por diretivas `data`. Já o tempo de vida de uma variável para um certo dispositivo compreende o momento em que essa variável fica disponível para ele e termina quando não está mais disponível para esse mesmo acelerador. Para variáveis compartilhadas entre dispositivos, o tempo de vida começa quando são alocadas e termina quando são desalocadas; variáveis estaticamente alocadas têm tempo de vida igual à duração do programa; finalmente, variáveis que não estão na memória compartilhada nem foram alocadas estaticamente têm tempo de vida num dispositivo desde o momento em que são copiadas para ele até suas cópias ficarem indisponíveis no dispositivo

2.5 Diretivas `data`

As diretivas para gerenciamento de dados, todas envolvendo a palavra `data` em seus pragmas, podem ser de dois tipos: estruturadas e não estruturadas. Ambas compartilham as seguintes características: a alocação no e possíveis movimentos de dados para o dispositivo de execução ocorrem na entrada da *data region* enquanto a desalocação no acelerador e possíveis movimentos de dados para o dispositivo anfitrião ocorrem quando a execução deixa a *data region*.

Diretivas `data` estruturadas envolvem um bloco de código com chaves e podemos pensar que as alocações no e movimentos de dados para o dispositivo ocorrem em `{` e desalocações e movimentos de dados para o anfitrião ocorrem quando a execução do programa atinge `}`. Temos portanto algo como:

```
1 #pragma acc data copy(a[0:length]) copy(b[0:length]) copy(c[0:
      ↪ length])
2 {
3     #pragma acc paralell loop
4     for (int i = 0; i < length; i++)
5         c[i] = a[i] + b[i];
```

```
}
```

O dispositivo de execução tem acesso a `a`, `b` e `c` desde o momento em que a execução atinge `{` e não tem mais acesso em `}`. Nesse caso, os conceitos de *data lifetime* e *data region* se confundem visto que temos escopo léxico.

Diretivas não estruturadas, por outro lado, não deixam claro o tempo de vida de uma variável no dispositivo. Quando queremos disponibilizar um conjunto de dados para um dispositivo, utilizamos a diretiva `enter data`, e se queremos fazer desalocações, utilizamos `exit data`. Um exemplo de uso recorrente de diretivas não estruturadas ocorre em C++, nos construtores e destrutores de uma certa classe. Por exemplo, se temos um tipo `Vector`, podemos ter algo como:

```
    class Vector {
        private:
            float *arr;
            int len;
        public:
            Vector(int size) {
                len = size;
                arr = new float[len];
                #pragma acc enter data copyin(this)
                #pragma acc enter data create(arr[0:n])
            }
            ~Vector() {
                #pragma acc exit data delete(arr)
                #pragma acc exit data delete(this)
                delete(arr);
            }
            ...
    }
```

Vale lembrar, entretanto, que o uso de diretivas de dados é opcional para o funcionamento correto de programas, isso porque diretivas `parallel` e `kernels` já realizam movimentações e alocações dados. É claro, entretanto, que o uso manual pode ser considerado como uma otimização já que o programador provavelmente sabe de mais detalhes sobre uso de dados que o compilador.

2.6 Cláusulas de dados do OpenACC

As cláusulas de dados especificam o comportamento do ambiente de execução (o *runtime*) para alocação, desalocação e movimento de dados entre um dispositivo e o dispositivo anfitrião. Entre outros aspectos, o ambiente de execução monitora a quantidade de referências a um certo conjunto de dados que existem num certo dispositivo. Esse monitoramento permite que não sejam feitos movimentos de dados ou alocações desnecessários, ou mesmo incorretos. As seguintes são as cláusulas de dados mais importantes:

- `create(list)` começa verificando quais variáveis da lista fornecida estão disponíveis no dispositivo. Para cada variável não disponível, aloca memória vazia para ela no dispositivo e inicia a contagem de referências do dispositivo a essa variável. Para as variáveis já disponíveis no dispositivo, simplesmente aumenta a contagem de referências delas no componente.

- `copy(list)` é utilizada apenas em diretivas estruturadas. Verifica se a variável está presente no dispositivo. Se não está, aloca memória e copia os conteúdos dessa variável para o dispositivo. Se no final da região em que essa variável está a contagem será 0, então esses dados são copiados do dispositivo para o anfitrião e a memória no dispositivo é desalocada.

- `copyin(list)` verifica se a variável está disponível no dispositivo. Se não está disponível aloca memória e copia os dados do anfitrião para o dispositivo. A desalocação só é feita caso a contagem de referências esteja em zero.

- `copyout(list)` verifica se a variável está disponível no dispositivo. Se não está, aloca memória, sem populá-la. Se utilizada numa diretiva estruturada, quando a execução atinge }, os conteúdos dessa variável são copiados para o anfitrião. Pode também ser utilizada em diretivas `exit data`.

- `delete(list)` força a contagem de referências a zero e desaloca a memória presente lá para objetos que estão presentes no dispositivo.

- `present(list)` sinaliza que estamos supondo que as variáveis listadas estão disponíveis no dispositivo, para que possamos realizar cômputos

É boa prática, e em alguns casos obrigatório, sinalizar o formato do conjunto de dados que estamos manipulando. Em C/C++ isso é feito fornecendo um índice de início e a quantidade de elementos que queremos manipular. Por exemplo, para um array, fazemos `copy(nome_do_array[starting_index:length])`. Para mais dimensões, `copy(nome_do_array[0:N][0:M])` sinaliza uma matriz de N linhas e M colunas.

Vejamos agora alguns exemplos.

Exemplo 01

```
#include <math.h>

int main() {
    float tol = 0.0001;
    float A[1000][1000], Anew[1000][1000];
    int iterations = 1000, current = 0;
    float err = tol;
    while (err >= tol && current < iterations) {
        err = 0.0;

        #pragma parallel loop reduction(max:err) copyin(A[0, n*m
            ↪ ]) copyout(Anew[0, n*m])
        for (int i = 0; i < 1000; i++)
            for (int j = 0; j < 1000; j++) {
                Anew[i][j] = 0.25 * (A[i - 1][j] + A[i][j - 1] +
                    ↪  A[i + 1][j] + A[i][j + 1]);
                err = max(err, abs(Anew[i][j] - A[i][j]));
            }

        #pragma parallel loop reduction(max:err) copyin(Anew[0,
            ↪ n*m]) copyout(A[0, n*m])
        for (int i = 0; i < 1000; i++)
            for (int j = 0; j < 1000; j++)
                A[i][j] = Anew[i][j];
        current++;
    }
    return 0;
}
```

No exemplo acima, em cada iteração do `while`, fazemos uma copia de `A` para o dispositivo, uma cópia de `Anew` para o anfitrião, uma cópia de `Anew` para o dispositivo, e outra de `A` para fora. Note que isso acrescenta latência à execução geral, de modo que é mais interessante fazer a cópia de `A` e de `Anew` antes da primeira iteração.

Exemplo 2

Dentro de uma diretiva estruturada `data`, os dados *pertencem* ao dispositivo, então não há movimento de dados. Assim, todas as iterações do laço interno à diretiva de dado ocorrem dentro do dispositivo.

```
1 #pragma data copyin(a[0:N], b[0,N]) copyout(c[0:N])
2 {
3     // dentro dessa regiao, a, b e c sao elementos dentro do
          ↪ acelerados -> é interessante notar que nos referimos
          ↪  aos vetores com o mesmo nome
4     // mas na verdade são regiões de memória diferentes
5     // assim, essas diretivas nos permitem observar tudo de
          ↪ maneira unificada, ainda que tenhamos de gerenciar o
          ↪  movimento de dados
6     #pragma parallel loop
7     for (int i = 0; i < N; i++)
8         c[i] = a[i] + b[i];
9 }
```

O fluxo de execução do trecho de código acima é o seguinte:

- Aloca espaço para `a` no dispositivo
- Copia os conteúdos do `a` da CPU no `a` do dispositivo
- Faz o mesmo que foi feito acima para `b`
- Aloca espaço no acelerador para `c`, mas não popula a versão de `c` do acelerador
- Quando sair da região entre {}, copia os conteúdos do `c` do dispositivo para o `c` do dispositivo

Observe que o código entre chaves é executado dentro do acelerador, de modo que, nessa região de código, `a`, `b` e `c` são elementos do acelerador. Entretanto, é interessante notar que utilizamos os mesmos nomes de variáveis em trechos que são executados fora de aceleradores. Assim, o OpenACC nos permite observar todos os dados de maneira unificada, ainda que estejamos lidando com regiões de memória distintas entre si tenhamos de gerenciar o movimento de dados entre elas.

Exemplo 3

O exemplo abaixo é uma otimização do que foi feito no exemplo 1. Nele, realizamos um movimento de dados apenas no início do `while`.

```
1  int main() {
2      float tol = 0.0001;
3      float A[1000][1000], Anew[1000][1000];
4      int iterations = 1000, current = 0;
5      float err = tol;
6      #pragma data copy(A[0, n*m]) copyin(Anew[0, n*m])
7      while (err >= tol && current < iterations) {
8          err = 0.0;
9
10         #pragma parallel loop reduction(max:err)
11         for (int i = 0; i < 1000; i++)
12             for (int j = 0; j < 1000; j++) {
13                 Anew[i][j] = 0.25 * (A[i - 1][j] + A[i][j - 1] +
                       ↪  A[i + 1][j] + A[i][j + 1]);
14                 err = max(err, abs(Anew[i][j] - A[i][j]));
15             }
16
17         #pragma parallel loop reduction(max:err)
18         for (int i = 0; i < 1000; i++)
19             for (int j = 0; j < 1000; j++)
20                 A[i][j] = Anew[i][j];
21         current++;
22     }
23     return 0;
24 }
```

2.7 Sincronização de dados

O OpenACC permite atualização de dados para que o que está no dispositivo e o que está no anfitrião concordem entre si.

- `#pragma acc update self(list)` atualiza os dados de quem despachou o trabalho para um acelerador. Isto é, essa diretiva pega dados que estão no aparelho chamados `list` e atualiza a cópia de quem transferiu `list` para o acelerador.

- `#pragma acc update device(list)` coloca os dados da máquina anfitriã no aparelho.

2.8 Diretivas não estruturadas

Conforme exemplificado em 2.5, existem situações em que não podemos definir um trecho de código no qual uma variável ficará disponível para um dispositivo. Para esses casos, podemos utilizar as diretivas `enter` e `exit` precedendo `data` nas cláusulas de dados. Podemos pensar que `enter` é como se fosse `{` das cláusulas estruturadas e `exit` funciona como `}`. `enter` faz alocação de memória *no aparelho* e qualquer cópia *para* o aparelho, já `exit`, desaloca memória *no aparelho* e realiza cópias *do* o aparelho para o anfitrião.

Por esses motivos, a cláusula `enter` só pode vir acompanhada de `copyin` ou de `create`, enquanto `exit`, pode vir acompanhada de `delete` ou `copyout`. Vejamos um exemplo de uso:

```
1 #pragma enter data copyin(a[0:N], b[0,N]) create(c[0:N])
2     #pragma parallel loop
3     for (int i = 0; i < N; i++)
4         c[i] = a[i] + b[i];
5 #pragma exit data copyout(c[0:N]) delete(a[0:N], b[0:N])
```

É claro que o trecho de código acima não justifica o uso de diretivas não estruturadas, diferentemente de quando temos construtores e destrutores. Entretanto, o que queremos ressaltar com o exemplo acima, que inclusive já foi mostrado, é que o uso de cláusulas de dados não estruturadas nos obriga a desalocar memória do acelerador explicitamente, como é feito colo `delete` na linha 7. Note que isso não precisa ser feito com cláusulas estruturadas, utilizando inclusive as mesmas diretivas.

2.9 Trabalhando com structs e classes

Quando lidamos com dados encapsulados em structs ou classes, temos algumas particularidades com o gerenciamento de memória. Se estamos trabalhando com

structs de tamanho fixo, que não possuem membros que são alocados dinamicamente, lidamos com seu gerenciamento da mesma forma como fazemos com tipos nativos, conforme mostramos abaixo:

```
typedef struct {
    float x, y, z
} float3;

int main(int argc, char* argv[]) {
    int n = 10;
    float3* f3 = malloc(n * sizeof(float3));

    #pragma acc enter data create(f3[0:n]);

    #pragma acc kernels
    for (int i = 0; i < n; i++) {
        f3[i].x = 0.0f;
        f3[i].y = 0.0f;
        f3[i].z = 0.0f;
    }

    #pragma acc exit data delete(f3)
    free(f3);
}
```

E se temos um *struct* de tamanho variável[1], como o *struct* abaixo?

```
typedef struct {
    float* arr;
    int len;
} vector;
```

Bem, nessas situações, precisamos aninhar alocações e fazê-las manualmente. Teremos que primeiro alocar primeiro o *struct* em si e depois seus membros, conforme abaixo:

[1]Na realidade, o *struct* em si continua possuindo tamanho fixo, mas possui um ponteiro que aponta para uma região de memória de tamanho não determinado em tempo de compilação.

```
int main(int agrc, char* argv[]) {
    vector v;
    v.len = 10;
    v.arr = (float*) malloc(v.len * sizeof(float));

    #pragma acc enter data copyin(v)
    #pragma acc enter data copyin(v.arr[0:v.len])
    ...
    #pragma acc exit data delete(v.arr)
    #pragma acc exit data delete(v)
    free(v.arr);
}
```

Com classes, podemos fazer algo como:

```
private:
    float* arr;
    int len;
public:
    vector(int n) {
        len = n;
        arr = (float*) malloc(len * sizeof(float));
        #pragma enter data create(this)
        #pragma enter data create(arr[0:len])
    }
    ~vector() {
        #pragma exit data delete(arr[0:len])
        #pragma exit data delete(this)
        delete(arr);
    }
```

Quando estamos realizando o gerenciamento de memória por nós mesmos, isto é, de forma não automática, é boa prática e comum criar métodos de sincronização, para não precisar fazer diretamente invocar várias sincronizações no código e para que as diferentes versões, em diferentes regiões de memória, de uma mesma

variável concordem entre si. Por exemplo, dentro da nossa classe de `vector`, podemos fornecer os seguintes dois métodos:

```
1 void accUpdateSelf() {
2     #pragma acc update self(arr[0:len])
3 }
4
5 void accUpdateDevice() {
6     #pragma acc update device(arr[0:len])
7 }
```

2.10 Níveis de paralelismo em OpenACC: gangues, trabalhadores e vetores

Antes de entrarmos nas funcionalidades a que a API do OpenACC nos expõe, convém expor como está organizado o paralelismo a nível de hardware. Para isso, vamos brevemente memorar como estão organizadas as arquiteturas de uma CPU e de uma GPU.

Uma CPU possui alguns controladores, um cache L3, e alguns cores. Cada core, por sua vez, possui um cache L2, um cache L1, e uma unidade lógico-aritmética (ULA). Cada core, executa uma instrução por vez, cada instrução tipicamente toma um número como entrada e produz outro (utilizando sua ULA) como saída. Esse é o paradigma *single instruction single data* (SISD). Hoje em dia, entretanto, é comum que uma CPU *multicore* ofereça paralelismo a nível de instrução entre seus cores; nesse caso, vários cores podem executar a mesma instrução ao mesmo tempo, cada um atuando sobre um número. Esse paradigma é conhecido como *single instruction multiple data* (SIMD).

Uma GPU, por outro lado, está inerentemente implementada de acordo com o paradigma SIMD. Cada GPU, possui uma série de *streaming multiprocessors* (SMs). Esses vários SMs compartilham entre si um cache L2 e uma memória global do dispositivo. Cada SM é composto por um conjunto de cores, cada um com um cache L1, um controlador e várias ULAs. Isso permite que cada core execute uma instrução em várias entradas numéricas ao mesmo tempo. Do ponto de vista de software, uma função (chamada de *kernel* em CUDA) é executada em vários SMs, cada SM executa um *thread block*, que por sua vez é composto de várias *threads*, cada uma delas executada em um core da GPU. Note portanto, que uma *thread* numa GPU pode não ser exatamente sequencial, na medida em que pode executar instruções vetorizadas. Além disso, *threads* num mesmo *thread block* comparti-

lham alguns de seus dados e podem ser sincronizadas entre si, mas *thread blocks* são executados de maneira independente e sem compartilhamento de memória.

O OpenACC dá a quem está programando o controle sobre a divisão do trabalho no dispositivo em questão, com *gangs* (gangues), *worker* (trabalhadores) e *vectors* (vetores). Uma gangue é um conjunto de trabalhadores. Gangues diferentes são executadas de forma independente, sem compartilhamento de memória. Trabalhadores numa gangue podem ser sincronizados entre si e acessam a mesma memória. Podemos pensar que trabalhadores são *threads* de execução, isto é a unidade que executará um trecho de código. Vetores, por fim, expõe o nível mais baixo de paralelismo, a nível de instrução, e permite que quem está desenvolvendo o programa possa especificar o tamanho da memória sobre a qual um trabalhador está atuando, isto é, a quantidade de dados sobre a qual uma instrução vetorizada executada por um trabalhador agirá.

No contexto de uma CPU moderna, com suporte a SIMD, uma gangue é traduzida para a CPU inteira, um trabalhador, para um core, e um vetor, para um vetor SIMD. Se por outro lado, estamos lidando com GPUs da NVIDIA, uma gangue será um *thread block*, um trabalhador é o que se chama de *warp*, e um vetor é uma *thread*.

Quando utilizamos a diretiva `parallel` são construídas gangues, cada uma com vários trabalhadores, cada um executando operações vetorizadas ou SIMD. O OpenACC permite que definamos o número de gangues, de trabalhadores e o comprimento de vetores com as cláusulas `num_gangs(n)`, `num_workers(n)` e `vector_length(l)`, respectivamente. Além disso, as cláusulas `gang`, `worker` e `vector`, acompanhadas de uma diretiva `parallel`, indicam ao compilador que as iterações do laço seguinte ao pragma em que elas estão sendo utilizadas devem ser distribuídas entre gangues, trabalhadores ou vetores, respectivamente.

3 OpenCL

O OpenCL, ou Open Computing Language, assim como o OpenACC, surgiu com o objetivo de fornecer uma plataforma uniforme e aberta para sistemas computacionais heterogêneos. Sua primeira especificação e sua primeira implementação foram inicialmente desenvolvidas pelas Apple em conjunto com grupos técnicos da AMD, IBM, Qualcomm, Intel e NVIDIA. Posteriormente, a Apple enviou uma primeira proposta de especificação para o Khronos Group, um consórcio de tecnologia sem fins lucrativos. Em junho de 2008, o Khronos Group formou o Khronos Compute Working Group, que lançou a primeira especificação do OpenCL em dezembro de 2008. A primeira implementação, por outro lado, foi lançada em Agosto de 2009, pela Apple, junto com o Mac OS X Snow Leopard. Desde então, a especificação do OpenCL está sob responsabilidade do Khronos Group. Esse consórcio tem como objetivos a criação de padrões portáveis e abertos em áreas como computação gráfica 3D, aprendizado de máquina, visão computacional e programação paralela, é claro. Dentre os produtos mantidos pelo Khronos Group, estão o OpenGL, o WebGL e o Vulkan.

Diferentemente do OpenACC, o modo de se trabalhar com o OpenCL está baseado no uso de funções fornecidas pela sua API e na escrita de funções numa linguagem baseada em C. Além disso, esse arcabouço expõe quem está desenvolvendo a aplicação a muito mais detalhes de *hardware* da plataforma em que a aplicação está sendo executada, conforme veremos a seguir. O conceito central dentro desse arcabouço é o de *kernel*, que é uma função que pode ser executada em diferentes tipos de dispositivos, ou chips, como CPUs, GPUs e FPGAs. Um *kernel* escrito em OpenCL pode ser (mas não obrigatoriamente é) compilado por um *runtime compiler*, que compila o *kernel* em questão enquanto o programa está sendo executado na máquina anfitriã. Isso dá grande flexibilidade aos programas, pois permitem que eles sejam transformados em programas não portáveis em tempo de execução.

Além da linguagem, o OpenCL inclui bibliotecas e um *runtime* para o desenvolvimento de aplicações. Esse arcabouço é estruturado em torno de quatro mo-

delos: o de plataforma, o de execução, o de memória e o de programação (ou o arcabouço propriamente dito).

A escrita deste e os exemplos fornecidos neste capítulo foram baseados em [Gro23b], [Gro23a], [BB13], [Sca11] e [Gui].

3.1 O modelo de Plataforma

O modelo de plataforma é a abstração que o OpenCL dá ao hardware no qual uma aplicação é executada. Uma plataforma no OpenCL é um conjunto formado por uma máquina anfitriã e vários dispositivos. Os dispositivos, são compostos por unidades de computação (UCs), que por sua vez dividem-se em elementos processadores (EPs), que efetivamente levam as computações a cabo. Cada UC tem uma memória que é compartilhada entre seus EPs chamada de memória local, e cada EP tem uma memória própria, chamada de memória privada. Além dessas duas, um dispositivo tem duas memórias compartilhadas por todas as suas UCs e com a máquina anfitriã: as memórias global e constante.

Uma plataforma OpenCL também deve oferecer compiladores para o código dos *kérneis*. Esses compiladores podem ser *online* ou *offline*. Os primeiros permitem que os códigos dos *kérneis* sejam compilados em tempo de execução e sejam invocados pelo programa anfitrião, oferecendo portabilidade até mesmo na compilação dos programas; os segundos, por outro lado, devem ser executados de fora do programa anfitrião, utilizando comandos específicos da máquina anfitriã. A especificação do OpenCL define uma plataforma *full profile* como aquela que fornece um compilador *online* para todos os seus dispositivos.

A especificação do OpenCL não restringe que tipo de componentes de hardware devem ser UCs ou EPs, nem a implementação (em termos de hardware) das memórias listadas. O mapeamento entre as UCs, EPs, memórias e os componentes de *hardware* fica a cargo de quem implementará esse arcabouço, desde que siga as normas determinadas na especificação.

Como podemos ter mais de uma plataforma em nosso sistema, cada uma com suporte a uma versão do OpenCL, por exemplo, a API do OpenCL fornece algumas funções para consultas a respeito da plataforma em que estamos executando nosso programa. Dessas, destacamos duas: `clGetPlatformIDs` e `clGetPlatformInfo`. A primeira tem assinatura `cl_int clGetPlatformIDs (cl_uint num_entries, cl_platform_id *platforms, cl_uint *num_platforms);` ela é utilizada para para obter o número de plataformas que nosso sistema possui e posteriormente um vetor de identificadores de plataforma, para que possamos acessá-las e manipulá-las. A segunda tem assinatura `cl_int clGetPlatformInfo(cl_platform_id platform, cl_platform_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret`

) e é utilizada para obter informações a respeito de uma plataforma uma vez que já tenhamos o seu identificador. Abaixo, mostramos um exemplo de uso das duas funções.

```
cl_uint num_platforms;
cl_int cl_Error;
cl_Error = clGetPlatformIDs(0, NULL, &num_platforms);
printf("Temos %d plataformas\n", num_platforms);
cl_platform_id * platforms = (cl_platform_id *)malloc (
    ↪ num_platforms * sizeof(cl_platform_id));
cl_Error = clGetPlatformIDs(num_platforms, platforms, NULL);
char * info = (char *) malloc(1024 * sizeof(char));
for (int i = 0; i < num_platforms; i++) {
   cl_Error = clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME,
       ↪ sizeof(info), (void *)info, NULL);
   printf("Nome da plataforma igual a %s\n", info);

   cl_Error = clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR
       ↪ , sizeof(info), (void *)info, NULL);
   printf("Vendor da plataforma igual a %s\n", info);

   cl_Error = clGetPlatformInfo(platforms[i],
       ↪ CL_PLATFORM_VERSION, sizeof(info), (void *)info, NULL
       ↪ );
   printf("Versao da plataforma igual a %s\n", info);

   cl_Error = clGetPlatformInfo(platforms[i],
       ↪ CL_PLATFORM_PROFILE, sizeof(info), (void *)info, NULL
       ↪ );
   printf("Perfil da plataforma igual a %s\n", info);

   cl_Error = clGetPlatformInfo(platforms[i],
       ↪ CL_PLATFORM_EXTENSIONS, sizeof(info), (void *)info,
       ↪ NULL);
```

```
22      printf("Extensoes da plataforma igual a %s\n", info);
23  }
```

Note que realizamos duas chamadas a `clGetPlatformsID`; a primeira (na linha 3) serve para sabermos a quantidade de plataformas que possuímos e por isso passamos o endereço de `num_platforms`. A segunda (na linha 6) efetivamente obtém as plataformas, colocando seus identificadores em `platforms`. Com `platforms` populado, podemos obter nome, vendedor, versão do OpenCL, perfil e extensões de cada uma das plataformas.

3.2 O modelo de execução

Se por um lado o modelo de plataforma estabelece a abstração que o OpenCL dá ao hardware, o modelo de execução é a abstração segundo a qual o OpenCL determina o gerenciamento da execução de um programa e a divisão do trabalho entre o anfitrião, os dispositivos, as UCs e os EPs. Ele determina que uma aplicação em OpenCL é composta de código anfitrião, a ser executado na máquina anfitriã, e o código dos *kérneis*, que são funções a serem executadas nos dispositivos, sendo as sintaxes dessas duas partes diferentes entre si, como veremos mais adiante. Mais detalhadamente, a execução de uma aplicação é orquestrada pelo código anfitrião e o trabalho propriamente dito está feito no código dos *kérneis*. Assim, o código anfitrião é responsável pelo envio do código dos *kérneis* aos dispositivos e sincronização, e esses, por sua vez, realizam cômputos. Os dispositivos têm bastante liberdade sobre a maneira como o código será executado e distribuído entre suas UCs e EPs.

Para consultas a respeito dos dispositivos que temos numa certa plataforma, temos duas funções da API do OpenCL: `clGetDeviceIDs` e `clGetDeviceInfo`, de assinaturas `cl_int clGetDeviceIDs(cl_platform_id platform, cl_device_type device_type , cl_uint num_entries, cl_device_id *devices, cl_uint *num_devices)` e `cl_int clGetDeviceInfo (cl_device_id device, cl_device_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)`, respectivamente. Enquanto `clGetDeviceIDs` permite realizar consultas sobre um tipo de dispositivo (ou todos, utilizando `device_type` igual a `CL_DEVICE_TYPE_ALL`) de uma certa plataforma, `clGetDeviceInfo` permite obter informações sobre um certo dispositivo. Essas duas funções são utilizadas de maneira bastante semelhante ao que vimos para `clGetPlatformIDs` e `clGetPlatformInfo`. Vejamos um exemplo:

```
1   cl_uint num_devices;
```

```
2  cl_Error = clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, 0,
       ↪ NULL, &num_devices);
3  printf("\n\nTemos %d dispositivos nessa plataforma\n",
       ↪ num_devices);
4  cl_device_id* devices = (cl_device_id*) malloc(num_devices *
       ↪ sizeof(cl_device_id));
5  cl_Error = clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL,
       ↪ num_devices, devices, NULL);
6  char device_info[1024];
7  int queryInt;
8  for (int j = 0; j < num_devices; j++) {
9     cl_Error = clGetDeviceInfo(devices[j], CL_DEVICE_NAME, sizeof
          ↪ (device_info), &device_info, NULL);
10    printf("Nome do dispositivo: %s\n", device_info);
11
12    cl_Error = clGetDeviceInfo(devices[j], CL_DEVICE_VENDOR,
          ↪ sizeof(device_info), &device_info, NULL);
13    printf("Vendor do dispositivo: %s\n", device_info);
14
15    cl_Error = clGetDeviceInfo(devices[j], CL_DRIVER_VERSION,
          ↪ sizeof(device_info), &device_info, NULL);
16    printf("Versao do driver do dispositvo: %s\n", device_info);
17
18    cl_Error = clGetDeviceInfo(devices[j], CL_DEVICE_VERSION,
          ↪ sizeof(device_info), &device_info, NULL);
19    printf("Versao do dispositivo: %s\n", device_info);
20
21    cl_Error = clGetDeviceInfo(devices[j],
          ↪ CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(int), &queryInt,
          ↪ NULL);
22    printf("Quantidade de unidades de computação do dispositivo:
          ↪ %d\n", queryInt);
23 }
```

O despacho de trabalho aos dispositivos é feito utilizando filas de comandos, que podem ser de três tipos: comandos de enfileiramento de *kérneis*, comandos de memória e comandos de sincronização. Comandos de enfileiramento de *kérneis*, como o nome sugere, colocam *kérneis* na fila de comando para serem executados nos dispositivos. Comandos de memória instruem transferência de dados entre o anfitrião e os dispositivos e o gerenciamento de memória nos dispositivos. Comandos de sincronização estabelecem relações entre comandos a fim de que uns sejam executados antes de outros. Essas instruções têm um ciclo de vida que passa por vários estados, e a mudança de um estado a outro é comunicada utilizando objetos evento.

Quando o código anfitrião faz uma chamada a um *kernel*, ele fornece valores para os argumentos do *kernel*. Independentemente do EP em que o *kernel* será executado, os parâmetros fornecidos pela máquina anfitriã são os mesmos. Ora, se há uma única chamada de *kernel*, com os mesmos parâmetros, como cada EP fará o seu trabalho, diferente de outros EPs? A resposta está no conjunto de índices a que uma chamada de *kernel* dá origem. Uma chamada de *kernel* pelo código anfitrião define a quantidade de vezes que um *kernel* será executado e como cada execução será localizada utilizando esse espaço de índices. Então cada execução localiza *a si mesma* e executa o seu trabalho com base em seu índice.

Vejamos abaixo um exemplo de delegação de execução para um dispositivo.

```
1  cl_context context = clCreateContext(NULL, 1, devices, NULL,
       ↪ NULL, &cl_Error);
2  cl_command_queue queue = clCreateCommandQueueWithProperties(
       ↪ context, devices[0], NULL, &cl_Error);
3  const char* kernel_source = "__kernel void kernel_exemplo(
       ↪ __global int* entrada, __global int* saida) { "
4                              "  int id = get_global_id(0); "
5                              "  if (id >= 4) "
6                              "      return;"
7                              "  saida[id] = entrada[id]; "
8                              "} ";
9  cl_program program = clCreateProgramWithSource(context, 1, &
       ↪ kernel_source, NULL, &cl_Error);
10 cl_Error = clBuildProgram(program, 1, devices, NULL, NULL, NULL)
       ↪ ;
11 cl_kernel kernel = clCreateKernel(program, "kernel_exemplo", &
```

```
    ↪ cl_Error);
12  int entrada[] = {1, 2, 3, 4};
13  cl_mem buffer_entrada = clCreateBuffer(context, CL_MEM_READ_ONLY
    ↪ , 4 * sizeof(int), NULL, &cl_Error);
14  cl_Error = clEnqueueWriteBuffer(queue, buffer_entrada, CL_TRUE,
    ↪ 0, 4 * sizeof(int), entrada, 0, NULL, NULL);
15  cl_int saida[4];
16  cl_mem buffer_saida = clCreateBuffer(context,  CL_MEM_WRITE_ONLY
    ↪ , 4 * sizeof(cl_int), NULL, &cl_Error);
17  clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &
    ↪ buffer_entrada);
18  clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *) &buffer_saida
    ↪ );
19  size_t globalWorkSize = 4;
20  clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalWorkSize,
    ↪ NULL, 0, NULL, NULL);
21  clEnqueueReadBuffer(queue, buffer_saida, CL_TRUE, 0, 4 * sizeof(
    ↪ cl_int), saida, 0, NULL, NULL);
```

Em primeiro lugar, precisamos de um contexto, que é um objeto responsável por englobar alguns dispositivos de uma plataforma; fazemos isso com `clCreateContext`, função à qual passamos uma lista de dispositivos e seu comprimento (são o terceiro e segundo argumento). Em seguida, criamos uma fila, para enviar comandos a um certo dispositivo, como na linha 2. Depois, criamos um programa, responsável por carregar em si um conjunto de *kérneis*, com `clCreateProgramWithSource` (linha 9). A essa função passamos uma lista de strings com os códigos dos *kérneis* (o terceiro argumento), o comprimento dessa lista (segundo argumento) e um contexto (primeiro argumento). Depois, fazemos a construção do programa e do *kernel*. Depois criamos dois *buffers* (explicados na Seção 3.3), com `clCreateBuffer` (nas linhas 13 e 16) para passar os argumentos aos dispositivos e configuramos os argumentos dos *kérneis* com `clSetKernelArg` (linhas 17 e 18). Por fim, enfileiramos a execução do *kernel* numa fila com `clEnqueueNDRangeKernel` (linha 20).

Uma *instância de kernel* é definida pela tripla código do *kernel*, os valores dos argumentos passados a ele e um espaço de índices. Um item de trabalho (ou *work item*) é uma execução de uma instância de um *kernel* e fica determinada pela instância e suas coordenadas no espaço de índices. Nesse sentido, podemos pensar que um item de trabalho tem dois tipos de parâmetros: os da lista de argumentos

do *kernel*, passados pelo anfitrião e iguais entre todos os itens de trabalho, e índices que localizam esse item no espaço de índices. Note como o código do *kernel* (linhas 3 a 8) no exemplo acima utiliza seu índice, que obtém com `get_global_id` na linha 4, para determinar a entrada do *array* de entrada que será copiada pelo item de trabalho; é assim que ocorre o paralelismo em OpenCL.

Vejamos agora em mais detalhes como fica determinado o espaço de índices.

NDRange: o espaço de índices do OpenCL

Um NDRange é um espaço n-dimensional, sendo n igual a 1, 2 ou 3, definido por 3 n-uplas:

- Uma n-upla que define o tamanho global em cada dimensão, denotada por $G = (G_1, \ldots, G_n)$.
- Uma n-upla de offsets ao longo de cada dimensão, denotada por $F = (F_1, \ldots, F_n)$; essa n-upla é a origem, por padrão.
- Uma n-upla com o tamanho dos grupos de trabalho em cada dimensão, denotada por $S = (S_1, \ldots, S_n)$.

Os espaço de identificadores globais, ou o espaço de índices, é definido como o seguinte conjunto: $\{(g_1, \ldots, g_n) : g_i \in \{F_i, \ldots, F_i + G_i - 1\}\}$. Por exemplo, digamos que n seja 3, G seja $(5, 7, 2)$ e F seja $(0, 9, 10)$, então o espaço de índices é $\{0, \ldots, 4\} \times \{9, \ldots, 15\} \times \{10, 11\}$. Cada item de trabalho é identificado com uma n-upla desse pertencente a esse espaço, chamado de identificador global (*work ID*). O NDRange também é divido em grupos de trabalho, de tal maneira que o número de grupos de trabalho é dado por $\frac{G_1}{S_1} \cdot \ldots \cdot \frac{G_n}{S_n}$; por simplicidade, vamos supor que os tamanhos globais são divisíveis pelos tamanhos dos grupos em cada dimensão, mas vale observar que desde de o OpenCL 2.0 há suporte para grupos de trabalho não uniformes. Grupos também possuem identificadores, que são n-uplas chamadas de *work-group ID* e serão denotadas por w. Além de identificadores globais, cada item de trabalho possui um identificador local, único por grupo, denotado por s. Para um certo item de trabalho, dados S, s, F e w, podemos obter seu identificador global g com a seguinte relação: $g_i = s_i + w_i \cdot S_i + F_i$. Ou seja, o identificador global é obtido tomando o índice do grupo a que o item pertence, multiplicando pelo seu tamanho, ajustando com o *offset* e finalmente adicionando o identificador local.

Retomando de forma mais específica, uma instância de *kernel* é definida pelo código do *kernel*, os valores de seus parâmetros, e um NDRange. Um item de trabalho é determinado por uma instância de *kernel* e um índice no NDRange.

3.3 O modelo de memória

O modelo de memória fica encarregado de precisar a diagramação (a nível de software, é claro) da memória, os tipos de dados permitidos por um programa, a forma como são armazenados e a maneira como são acessados, tanto para leitura, quanto para escrita. Os modelos de plataforma e execução estabelecem vários níveis de paralelismo segundo os quais uma aplicação é executada no OpenCL, e é responsabilidade do modelo de memória determinar como os seus objetos podem ser vistos por cada uma das entidades dos dois modelos já mencionados e as operações que realizam sobre ele. Isso permite que o desenvolvedor projete seus programas corretamente. O modelo de memória é dividido em quatro partes:

- Regiões de memória
- Objetos de memória
- Memória compartilhada virtual
- Modelo de consistência

Regiões de memória

A memória de uma plataforma está dividida em duas partes: a memória do anfitrião, disponível somente para ele, e a memória dos dispositivos, acessível a esses componentes do modelo de plataforma. A memória acessível a dispositivos é dividida em quatro partes: a memória global, a memória constante, a memória local e a memória privada. As memórias global e constante são acessíveis a todos os dispositivos de um dado contexto, sendo que a memória global oferece acesso de leitura e escrita, enquanto a constante permite apenas leitura. É por essas regiões que informações são enviadas do anfitrião para dispositivos, sendo responsabilidade única do anfitrião a alocação e inicialização de objetos na memória constante. A memória local fica disponível para consulta e modificação de todos os itens de trabalho de um certo grupo de trabalho, enquanto que a memória privada tem seu acesso restrito a um item de trabalho. É importante salientar que foram mencionados grupos e itens de trabalho, e não UCs e EPs; em outras palavras, mais do que associadas a componentes de hardware, essas memórias estão acessíveis dentro de um contexto de execução, e não são persistentes. Vale mencionar, que a divisão entre as regiões de memória é lógica e não necessariamente física, isso fica a cargo de quem implementa o OpenCL.

Além dessas regiões de memória, o OpenCL oferece também um espaço genérico de endereços para acesso às memórias global, locais e privadas. Esse recurso

não flexibiliza as regras de acesso nem as regras de persistência mencionadas anteriormente, mas permite que ponteiros para memórias global, locais e privadas sejam explicitamente convertidos para ponteiros nesse espaço genérico de endereços (e vice-versa) e permite a conversão implícita de ponteiros das memórias global, local e privada para o espaço genérico de endereços.

Objetos de memória

Objetos de memória são objetos do tipo `cl_mem` e podem ser *buffers*, imagens (*images*) ou *pipes*, são esses objetos que populam a memória global. *Buffers* são regiões contíguas de memória que permitem acesso "aleatório", como *arrays*, com suporte a alocação de diferentes tipos. Imagens são abstrações dos tipos e formatos mais conhecidos de imagens. *Pipes*, por sua vez, são abstrações de filas de dados nos quais um (e apenas um) ente de execução enfileira um dado por vez para ser lido e dos quais um (e apenas um) ente de execução lê um dado previamente enfileirado por vez.

Para o manejo de objetos de memória destacamos as seguintes funções: `clCreateBuffer`, `clEnqueueWriteBuffer` e `clEnqueueReadBuffer`. Para criar um *buffer*, fornecemos um contexto, suas propriedades (se é um *buffer* para leitura ou escrita), seu tamanho e possivelmente um *array* do anfitrião. Para escrever num *buffer*, fornecemos uma fila, um *buffer*, um *offset* da base do *buffer* a partir da qual será feita a escrita, a quantidade de bytes que serão lidos e um ponteiro para a região do anfitrião de onde vamos transferir os dados para o *buffer*. A leitura é análoga. O uso dessas funções está exemplificado no exemplo dado na seção 3.2; `clCreateBuffer` é chamada nas linhas 13 e 16; `clEnqueueWriteBuffer` é utilizada na linha 14, para copiar para o *kernel* os dados que serão utilizados como entrada; já `clEnqueueReadBuffer` é utilizada na linha 21 para copiar para o anfitrião a saída do *kernel*.

Modelo de consistência de memória no OpenCL e sincronização

Um modelo de consistência de memória é um conjunto de afirmações a respeito da ordem em que operações de memória (leituras e escritas) são executadas e de quais são os valores devolvidos por uma leitura feita por um item de trabalho num certo instante da execução. Dentre esses modelos, destaca-se o o modelo de consistência sequencial, segundo o qual um programa executado em várias *threads* é equivalente a um programa sequencial obtido por uma ordenação de todas as instruções executadas por todas as *threads* que mantém a ordem das instruções executadas por uma *thread*; mais ainda toda alteração realizada na memória por uma *thread* é visível a todas as leituras seguintes (segundo a ordenação dada) realizadas por outras *threads*. Apesar de facilitar o trabalho de clientes de uma plataforma que segue esse

modelo, sua implementação é difícil e limita o paralelismo de uma aplicação. Por esse motivo, o OpenCL segue um modelo de consistência de memória relaxado, mas oferece mecanismos de sincronização e ordenação de instruções de memória para seus clientes.

As operações de sincronização do OpenCL são chamadas de operações atômicas ou muros (*fences*), também chamados de barreiras (*barriers*). Operações atômicas são aquelas que acontecem completamente ou não são iniciadas; no OpenCL, elas estão associadas a localizações na memória e só podem ser utilizadas em tipos atômicos (como as versões atômicas dos tipos `int`, `uint`, `float`, etc). Muros, por outro lado, não estão associados a acessos à endereços de memória, e sim à ordenação (parcial) entre instruções executadas por EPs.

Essas operações, no OpenCL são parametrizadas por (pelo menos) dois argumentos: uma ordenação de memória (*memory ordering*) e um escopo.

O parâmetro de ordenação de memória é uma constante nomeada, e pode assumir os seguintes valores: `memory_order_relaxed`, `memory_order_release`, `memory_order_acquire`, `memory_order_acq_rel` ou `memory_order_seq_cst`. A constante `memmory_order_relaxed` não impõe limitações à ordem e serve apenas para que seja possível executar uma operação de sincronização; a ordenação `memory_order_release` faz com que outros itens de trabalho que executarem futuramente alguma operação de sincronização com `memory_order_acquire` percebam os efeitos da operação parametrizada com `memory_order_release`; naturalmente, `memory_order_acq_rel` produz o efeito combinado das duas anteriores; e, por fim, uma operação parametrizada com `memory_order_seq_cst` busca reproduzir o comportamento sequencial.

Ora, o custo de tornar visíveis (segundo o parâmetro de ordenação) a todos os itens de trabalho num certo contexto os efeitos de operações de sincronização limitaria a velocidade do programa. Por esse motivo, o OpenCL oferece o escopo como segundo parâmetro para operações de sincronização. Com ele, limitamos a visibilidade que queremos dar aos efeitos das operações de memória cujo comportamento foi estipulado pelo parâmetro de ordenação. O escopo também é uma constante nomeada e pode assumir sete valores, dos quais destacamos três: `memory_scope_work_group`, `memory_scope_device` e `memory_scope_all_devices`. Quando o escopo é igual a `memory_scope_work_group`, as restrições de ordenação de memória aplicam-se somente ao grupo de trabalho do item de trabalho que a executou; se for igual a `memory_scope_device`, todos os itens de trabalho do dispositivo em que foi executada a operação de sincronização ficam sujeitos à ordenação de memória; quando o escopo vale `memory_scope_all_devices`, as regras são impostas a todos o itens de trabalho da plataforma e, quando estamos utilizando a memória virtual compartilhada, ao programa anfitrião também.

3.4 Modelo de programação, ou arcabouço

A especificação do OpenCL determina que o arcabouço é composto de três componentes: a camada de plataforma, o *runtime* e o compilador. A camada de plataforma permite que o programa anfitrião tenha acesso ao modelo de plataforma; o que se quer dizer com isso é que o anfitrião pode fazer consultas sobre os dispositivos que estão executando os *kérneis*. O *runtime* permite modificação dos contextos criados. E o compilador cria os executáveis dos *kérneis*, de maneira *online* ou *offline*.

3.5 OpenCL C

Agora que temos uma visão de alto nível de como o OpenCL está organizado, vejamos um pouco da sintaxe utilizada para escrever *kérneis* em sistemas heterogêneos utilizando o OpenCL. Mais especificamente, trataremos do OpenCL C.

O primeiro aspecto que devemos ter claro sobre OpenCL C é que é uma linguagem diferente da linguagem C. Nesse sentido, OpenCL C não é nem um subconjunto nem superconjunto da linguagem C. Em outras palavras, OpenCL não oferece todos os recursos que a linguagem C oferece, mas oferece alguns outros que a linguagem C não possui. Por exemplo, OpenCL C não oferece dois recursos bastante utilizados na linguagem C: recursão e ponteiros para funções. Essa diferença entre o OpenCL C e a linguagem C é decorrência do fato de que os *kérneis* do OpenCL C são executados em dispositivos de propósito mais específico, e não em CPUs de propósito geral, que possuem suporte a uma larga gama de instruções. Também devemos levar em consideração o fato de que o OpenCL deve ser suportado por um grande número de dispositivos de arquiteturas diferentes entre si.

Tipos

A grande diferença entre os tipos da linguagem C e os tipos do OpenCL C está na especificação dos tamanhos. Diferentemente da linguagem C, o OpenCL C determina seus tamanhos. Os tipos suportados são `bool`, `char`, `short`, `int`, `long`, `float`, `double`, `half` (utilizado para números em ponto flutuante, com a metade da precisão de um `float`), `size_t`, `ptrdiff_t`, `intptr_t`, `void`, e as versões sem sinal dos tipos inteiros.

Uma das consequências envolvidas nessa especificação mais restritiva é o fato de que pode haver diferenças entre os tipos do anfitrião e os tipos dos dispositivos. Por exemplo, se escrevemos `int d` num *kernel*, sabemos que temos um inteiro de 32 bits representado com complemento de 2; se escrevemos `int h`, no programa

anfitrião, a representação exata desse inteiro depende da máquina. Assim, fazer `d = h`, ou o contrário, não é permitido! Uma implementação em concordância com a especificação acusa erro de compilação. Pois bem, se a especificação não permite esse tipo de atribuição entre dados de um *kernel* e dados do programa anfitrião, como podemos trocar dados entre eles? Para isso, a API do OpenCL (disponível para uso no código anfitrião) têm tipos definidos. Basta prefixar o nome dos tipos mencionados no parágrafo anterior com `cl_`. Os tipos `bool`, `size_t`, `ptrdiff_t`, `intptr_t` e `void` não têm esse tipo de suporte.

O OpenCL também dá suporte a tipos vetorizados de alguns de seus tipos nativos, são eles `charn`, `ucharn`, `shortn`, `intn`, `uintn`, `ulongn`, `floatn`, `doublen`, sendo que `n` deve ser igual a `2`, `3`, `4`, `8` ou `16`. Como é de se esperar, um tipo vetorial (*vector type*, em inglês) é composto de `n` valores do tipo. Mais uma vez, temos que tomar cuidado ao nos referirmos a tipos vetoriais no programa anfitrião, devemos utilizar o prefixo `cl_`. Essa classe de dados permite também a realização de operações entre eles. Essa é a forma mais "granular" de paralelismo, o paralelismo a nível de operações. Se `u` e `v` são dados de um mesmo tipo vetorial e $\odot$ é uma operação aritmética, relacional, lógica ou *bitwise*, então `u` $\odot$ `v` é o vetor obtido aplicando a operação $\odot$ componente a componente. Mais ainda, se $\odot$ é aritmética, lógica ou *bitwise* então, o resultado tem o mesmo tipo dos operandos, enquanto que, se o operador for relacional, o resultado é do tipo `booln`. Além de operar entre dois vetores do mesmo tipo, podemos também operar entre um escalar e um vetor. Como é de se esperar, o resultado é igual ao de usar o escalar em todas as componentes na operação em questão. Por fim, vale mencionar que podemos acessar as componentes de um vetor, se `v` é um vetor, podemos fazer `v.sn` para acessar sua `n`-ésima componente. Aqui, `n` é um número que varia de 0 a 15 (desde que o vetor seja do tamanho apropriado, é claro) em notação hexadecimal, ou seja, `n` deve ser 1, 2, 3, 4, 5, 6, 7, 8, 9, `A`, `B`, `C`, `D`, `E` ou `F`. Apesar de permitir o acesso às componentes de um vetor, o OpenCL não permite o acesso ao endereço dos componentes de um vetor.

Qualificadores do espaço de endereços

Conforme exposto na Seção 3.3, a memória de uma plataforma OpenCL não é uniforme, no sentido de que há regiões diferentes, com propósitos e acessos diferentes. Por esse motivo, a declaração de uma variável no OpenCL não é unicamente composta de seu tipo e de seu nome, ela é composta da região, do tipo e do nome da variável. As palavras reservadas que indicam a região de memória onde o valor de uma variável será armazenado são `global`, `constant`, `local` e `private`. Assim, se queremos declarar um inteiro na memória privada de um EP, fazemos algo como `private int x`. Se queremos fazê-lo na memória global, escrevemos `global int x` e assim por diante.

Quando tratamos de ponteiros, temos duas regiões a serem consideradas: a região em que o ponteiro será armazenado e a região onde está o valor apontado. Consequentemente, declarações de ponteiros têm (como boa prática) o seguinte formato: `regiao_do_valor_apontado tipo *regiao_do_ponteiro nome_da_variavel`. Por exemplo, `local int *private ptr` declara um ponteiro que está na memória privada de um EP e aponta para um valor inteiro na memória local de uma UC. Quando o ponteiro e o valor apontado estão na mesma região de memória, o formato da declaração é `regiao tipo *nome_do_ponteiro`.

Por fim, é importante mencionar que podemos realizar transferências de dados entre regiões de memória utilizando ponteiros. Por exemplo, o código a seguir realiza uma transferência de dados entre a memória privada de um certo EP para a memória global. De tal maneira que esse tipo de código deve ser escrito e colocado com cuidado, visto que há latência envolvida nessa transferência.

```
1 global int *global x;
2 private int *private y;
3 *y = *x;
```

Funções

Quanto às funções do OpenCL (e isso se aplica a *kérneis*), não temos grandes diferenças em relação às funções do C99, exceto pelo fato de não podemos utilizar recursão. Um outro detalhe importante diz respeito aos argumentos, vejamos um exemplo:

```
1 float4 add(float4 x, float4 y) {
2     return x + y;
3 }
```

A função acima soma dois vetores de *floats* de tamanho 4. Mas, da mesma forma que variáveis precisam ter qualificadores do espaço de endereços, os parâmetros de uma função também precisam indicar a região de memória a que pertencem. Se não o fizermos, a região padrão de memória é `private`. Assim, uma descrição mais precisa da função acima diria que ela soma dois vetores de *floats* de tamanho 4 da memória privada de um certo EP e devolve um vetor de *float* de tamanho 4. Logo, se queremos uma função que realize o mesmo para parâmetros globais, teremos de ter a função abaixo.

```
1 float4 add(global float4 x, global float4 y) {
2     return x + y;
```

```
}
```

Essa restrição, além de inconveniente, pode levar à duplicação de código, como mostrado.

Kérneis

Agora vejamos algumas especificidades dos *kérneis*. O primeiro detalhe sobre a escrita de *kérneis* diz respeito às suas assinaturas. A assinatura de uma dessas funções deve ser iniciada pela palavra reservada `kernel`. Além disso, todo *kernel* tem `void` como seu valor de retorno, o que já nos diz que toda comunicação entre quem faz a chamada do *kernel* e quem o executa ocorre por efeito colateral. Vejamos agora um primeiro exemplo, que soma dois *arrays*:

```
kernel void add(global float* x, global float* y, global float*
    ↪ z, uint32_t len) {
    size_t i = get_global_id(0);
    if (i >= len)
        return;
    z[i] = x[i] + y[i];
}
```

Vamos ainda supor que esse *kernel* esteja sendo executado num NDRange de dimensão 1, ou seja, o espaço de índices é grosso modo um conjunto de inteiros. A função `get_global_id` devolve um dos componentes do índice global de um item de trabalho e seu argumento é o componente que se quer, de modo que os valores válidos para o seu argumento vão de 0 até a dimensão do NDRange menos um. Como nesse caso o espaço de índices tem dimensão 1, estamos simplesmente pegando seu identificador. Se esse identificador não corresponde a alguma entrada do *array* `z`, não fazemos nada, do contrário obtemos seu valor. Agora note a diferença entre um *kernel* do OpenCL a ser executado em um dispositivo e um trecho de código envolvido por uma diretiva `parallel loop` do OpenACC: enquanto o próprio *kernel* realiza o trabalho de descobrir qual a entrada de que está encarregado, é o anfitrião do OpenACC, utilizando um pragma e usualmente um índice de um laço, que fornece esse índice para o dispositivo.

Além de `get_global_id`, são costumeiras chamadas a `get_global_offset`, `get_local_id` e `get_group_id`. Ambas têm um único argumento, o componente do NDRange que queremos. As funções `get_global_id` e `get_local_id` também possuem versões lineares, que devolvem identificadores que são números (mais precisamente `size_t`); são elas `get_global_linear_id`, `get_local_linear_id`.

3.6 Um exemplo completo

Agora estamos em condições de examinar um programa completo escrito em OpenCL e entendê-lo completamente.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #ifdef __APPLE__
4  #include <OpenCL/cl.h>
5  #else
6  #include <CL/cl.h>
7  #endif
8  #define VECTOR_SIZE 1024
9
10 const char *saxpy_kernel = "__kernel void saxpy_kernel(float
       ↪ alpha, global float *A, global float *B, global float *C
       ↪ ) { \n"
11                          "    int i = get_global_id(0);\n"
12                          "    C[i] = alpha* A[i] + B[i];\n"
13                          "}\n"
14
15 int main(void) {
16     int i;
17     float alpha = 2.0;
18     float *A = (float*) malloc(sizeof(float) * VECTOR_SIZE);
19     float *B = (float*) malloc(sizeof(float) * VECTOR_SIZE);
20     float *C = (float*) malloc(sizeof(float) * VECTOR_SIZE);
21     for(i = 0; i < VECTOR_SIZE; i++) {
22         A[i] = i;
23         B[i] = VECTOR_SIZE - i;
24         C[i] = 0;
25     }
26
27     cl_uint num_platforms;
28     cl_int clStatus = clGetPlatformIDs(0, NULL, &num_platforms);
```

```
29      cl_platform_id * platforms = (cl_platform_id *) malloc(
            ↪ sizeof(cl_platform_id) * num_platforms);
30      clStatus = clGetPlatformIDs(num_platforms, platforms, NULL);
31
32      cl_device_id *device_list = NULL;
33      cl_uint num_devices;
34      clStatus = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU,
            ↪ 0, NULL, &num_devices);
35      device_list = (cl_device_id *) malloc(sizeof(cl_device_id) *
            ↪  num_devices);
36      clStatus = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU,
            ↪ num_devices, device_list, NULL);
37
38      cl_context context = clCreateContext(NULL, num_devices,
            ↪ device_list, NULL, NULL, &clStatus);
39
40      cl_command_queue command_queue = clCreateCommandQueue(
            ↪ context, device_list[0], 0, &clStatus);
41
42      cl_mem A_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,
            ↪ VECTOR_SIZE * sizeof(float), NULL, &clStatus);
43      cl_mem B_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,
            ↪ VECTOR_SIZE * sizeof(float), NULL, &clStatus);
44      cl_mem C_clmem = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
            ↪ VECTOR_SIZE * sizeof(float), NULL, &clStatus);
45      clStatus = clEnqueueWriteBuffer(command_queue, A_clmem,
            ↪ CL_TRUE, 0, VECTOR_SIZE * sizeof(float), A, 0, NULL,
            ↪  NULL);
46      clStatus = clEnqueueWriteBuffer(command_queue, B_clmem,
            ↪ CL_TRUE, 0, VECTOR_SIZE * sizeof(float), B, 0, NULL,
            ↪  NULL);
47
48      cl_program program = clCreateProgramWithSource(context, 1, (
```

```
        ↪ const char **) &saxpy_kernel, NULL, &clStatus);
49      clStatus = clBuildProgram(program, 1, device_list, NULL,
        ↪ NULL, NULL);
50      cl_kernel kernel = clCreateKernel(program, "saxpy_kernel", &
        ↪ clStatus);
51
52      clStatus = clSetKernelArg(kernel, 0, sizeof(float), (void *)
        ↪  &alpha);
53      clStatus = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void
        ↪ *) &A_clmem);
54      clStatus = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void
        ↪ *) &B_clmem);
55      clStatus = clSetKernelArg(kernel, 3, sizeof(cl_mem), (void
        ↪ *) &C_clmem);
56
57      size_t global_size = VECTOR_SIZE;
58
59      clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 1,
        ↪ NULL, &global_size, &local_size, 0, NULL, NULL);
60      clStatus = clEnqueueReadBuffer(command_queue, C_clmem,
        ↪ CL_TRUE, 0, VECTOR_SIZE * sizeof(float), C, 0, NULL,
        ↪  NULL);
61      clStatus = clFlush(command_queue);
62      clStatus = clFinish(command_queue);
63
64      for(i = 0; i < VECTOR_SIZE; i++)
65          printf("%f * %f + %f = %f\n", alpha, A[i], B[i], C[i]);
66
67      clStatus = clReleaseKernel(kernel);
68      clStatus = clReleaseProgram(program);
69      clStatus = clReleaseMemObject(A_clmem);
70      clStatus = clReleaseMemObject(B_clmem);
71      clStatus = clReleaseMemObject(C_clmem);
```

```
72      clStatus = clReleaseCommandQueue(command_queue);
73      clStatus = clReleaseContext(context);
74      free(A);
75      free(B);
76      free(C);
77      free(platforms);
78      free(device_list);
79      return 0;
80  }
```

Vamos terminar este capítulo com um exemplo de um programa completo escrito em OpenCL. Este exemplo possui um *kernel* bastante simples, ele atribui a `C`, um vetor, o valor de `alpha * A + B`, com `alpha` um número real, e `A` e `B` dois vetores de reais, esse exemplo é amplamente conhecido com o SAXPY, um acrônimo para *"Single-precision AX plus Y"*.

O programa começa alocando e populando os vetores dentro da máquina anfitriã (linhas 18 a 25). Em seguida, a máquina anfitriã "explora" sua plataforma; para isso, faz-se uma chamada a `clGetPlatformIDs` (linha 28), que preenche `num_platforms` com o número de plataformas, é claro. De posse da quantidade de plataformas, fazemos uma outra chamada a `clGetPlatformIDs`, dessa vez para popular o vetor `platforms` com os seus identificadores (linha 30). Entre as linhas 32 a 36, temos um padrão semelhante: escolhendo a primeira plataforma de `platforms`, fazemos uma primeira chamada a `clGetDeviceIDs` para descobrir o número de dispositivos que `platform[0]` possui (linha 34) e uma outra chamada à mesma função para popular o vetor de dispositivos `device_list` na linha 36.

Nas linhas 42 a 44 criamos *buffers* para a troca de informações entre o anfitrião e seus dispositivos. Mais especificamente, criamos um *buffer* para cada argumento do *kernel*, um padrão bastante comum. Em seguida, utilizando a fila criada na linha 40, enviamos comandos de preenchimento dos *buffers* de `A` e `B` nas linhas 45 e 46, respectivamente. Uma vez que os dados já estão disponibilizados na memória global, podemos criar nosso *kernel*, o que é feito com as chamadas a `clCreateProgramWithSource` (linha 48), `clBuildProgram` (linha 49) e `clCreateKernel` (linha 50). Depois, fornecemos os argumentos que serão utilizados por eles com `clSetKernelArg` (linhas 52 a 55). Assim, podemos já enfileirar um comando de execução de *kernel* com `clEnqueueNDRangeKernel` (linha 60) e posteriormente a transferência do que foi calculado pelo *kernel* para a cópia de `C` da máquina anfitriã com `clEnqueueReadBuffer` (linha 61).

Utilizamos então `clFlush` (linha 62) para que os comandos enfileirados em `command_queue`

sejam efetivamente executados e `clFinish` (linha 63) para bloquear a execução do programa na máquina anfitriã e aguardar o término da execução de todos comandos na fila. Por fim, fazemos a "limpeza" do ambiente com comandos desde a linha 68 até 79.

Esse programa, apesar de efetuar computações bastante simples, é bastante ilustrativo quanto à maneira como se trabalha com o OpenCL. Assim como o programa acima, o esquema geral de um programa no OpenCL segue, grosso modo, o seguinte roteiro:

1. Exploração da plataforma e dispositivos, com a escolha da plataforma, dispositivos para execução dos *kérneis* e criação de um contexto
2. Criação de filas de comandos
3. Criação de *buffers* e transferência de dados do anfitrião para a memória global ou constante
4. Criação, compilação e execução do *kernel*
5. Espera da execução do *kernel*
6. Transferência dos dados da memória global para a máquina anfitriã
7. Liberação dos *buffers* e memória utilizados

Portanto um parte significativa da aplicação se ocupa de detalhes a respeito do ambiente fornecido pelo OpenCL, e não da execução da função que realiza os cômputos relevantes para o programa. Além disso, toda a configuração do ambiente é bastante repetitiva entre programas e suscetível a erro; sendo assim, seria interessante alguma forma de automatizá-la. A título de comparação, fornecemos uma implementação do SAXPY escrito em OpenACC.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <openacc.h>
4  #define VECTOR_SIZE 1024
5
6  void saxpy(float alpha, float* A, float* B, float* C) {
7      #pragma acc parallel loop copyin(A[0:VECTOR_SIZE], B[0:
           ↪ VECTOR_SIZE]) copyout(C[0:VECTOR_SIZE])
8      for (int i = 0; i < VECTOR_SIZE; i++)
```

```
        C[i] = alpha * A[i] + B[i];
}

int main(void) {
    int i;
    float alpha = 2.0;
    float *A = (float*) malloc(sizeof(float) * VECTOR_SIZE);
    float *B = (float*) malloc(sizeof(float) * VECTOR_SIZE);
    float *C = (float*) malloc(sizeof(float) * VECTOR_SIZE);
    for (i = 0; i < VECTOR_SIZE; i++) {
        A[i] = i;
        B[i] = VECTOR_SIZE - i;
        C[i] = 0;
    }

    saxpy(alpha, A, B, C);

    for(i = 0; i < VECTOR_SIZE; i++)
        printf("%f * %f + %f = %f\n", alpha, A[i], B[i], C[i]);

    free(A);
    free(B);
    free(C);
    return 0;
}
```

Observemos que o programa em OpenACC tem menos da metade de linhas de código do que o programa que foi escrito em OpenCL; é claro que esse não é um indicativo de qualidade de código ou de desempenho. Entretanto, isso demonstra que o código escrito em OpenCL é bastante mais verboso do que o código em OpenACC. Notemos também que é bastante inconveniente o manejo da memória em OpenCL. É necessário criar um *buffer* para cada argumento de cada *kernel*; mais ainda, para a transferência de dados de um argumento, é preciso utilizar uma fila, e possivelmente checar o *status* da operação de enfileiramento de escrita ou leitura para utilizar o *buffer* corretamente. Um outro aspecto possivelmente ino-

portuno quando programamos com o OpenCL é o gerenciamento de dispositivos quando tempos mais de uma plataforma do OpenCL; nesses casos, temos que criar um contexto diferente para cada plataforma e gerenciar manualmente a execução de *kérneis* entre dispositivos de diferentes plataformas. Além disso, o OpenCL C, utilizado para escrita de *kérneis* é bastante limitado e restritivo; em particular fica a cargo do programador lidar com possíveis diferenças entre os tipos da máquina anfitriã e os tipos suportados pelo OpenCL C para os dispositivos. Uma outra dificuldade da programação em OpenCL é a coordenação das diferentes regiões de memória, cada uma com especificidades quanto ao acesso e a persistência de dados entre execuções ou grupos de trabalho.

É lógico que se de um lado temos inconveniência, de outro ganhamos controle sobre a execução. Aliás, o próprio gerenciamento de dispositivos e plataformas do OpenCL pode ser a única alternativa possível quando temos queremos otimizar uma tarefa e criar um *kernel* sob medida para um certo tipo de dispositivo, no caso em que temos mais de uma plataforma, ou dispositivos diferentes. Uma outra funcionalidade oferecida pelo OpenCL e não presente no OpenACC é o suporte ao processamento de imagens. Esse arcabouço possui tipos dedicados e funções direcionadas para manejo de imagens. Mais ainda, o OpenCL dá suporte diretamente para o controle preciso de grupos de trabalho, itens de trabalho e gerenciamento de memória. Esse tipo de ajuste fino não é oferecido pelo OpenACC e pode ser bastante relevante para melhora do desempenho de uma aplicação. Em termos de memória, o OpenCL dá ao programador a oportunidade de posicionar de maneira mais precisa os seus dados, nas memórias privadas, local, global e constante; esse tipo de otimização pode melhorar a localidade de uma aplicação, tornando-a mais eficiente.

Em resumo, o OpenACC nos fornece uma ferramente de aprendizado rápido e uso fácil para sistemas paralelos, enquanto o OpenCL nos oferece controle e possibilidade de otimizações em detalhe sobre a execução de uma aplicação.

4 OpenARC

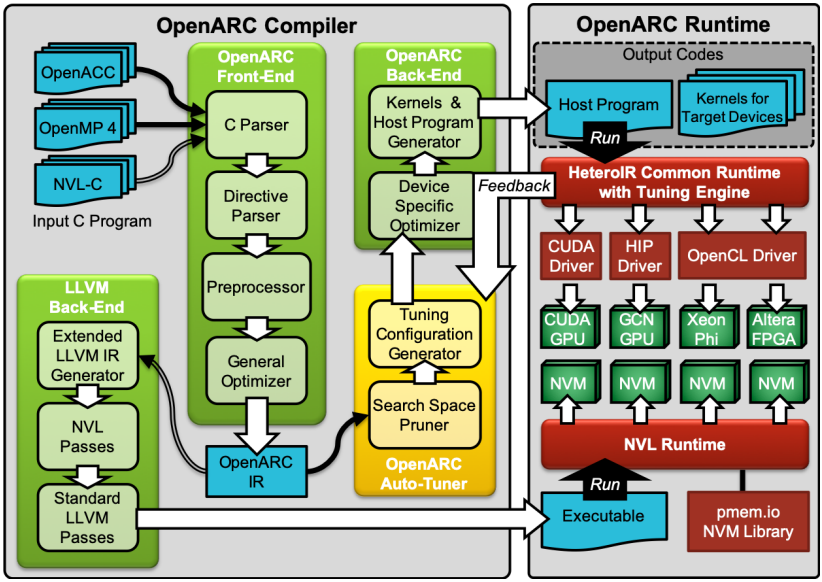
Foi com as dificuldades de utilizar arcabouços de mais baixo nível para sistemas heterogêneos, como OpenCL e CUDA, que [LV14] introduz o OpenARC, um transpilador que recebe programas escritos em OpenACC e/ou OpenMP e escreve programas em OpenCL ou CUDA. Nesse mesmo sentido, os autores de [LV12] e de [Vet+11] entendem que há entraves para o uso mais amplo de sistemas computacionais heterogêneos, entre as quais estão a programabilidade, a portabilidade e consistência em desempenho entre plataformas. Em linha com isso, a [Owe+07] aponta para necessidade de pessoal especializado na arquitetura de uma GPU para reescrever um programa escrito para CPU. Ainda, é necessário lembrar que muitas das pessoas que podem se beneficiar do uso de sistemas heterogêneos têm formação em meteorologia, química, física, biologia, etc. e não são especialistas em Ciência da Computação. Por isso, os autores de [LV14] desenvolveram um compilador que não só realiza *source-to-source transformation* (ou transpilação) partindo de um arcabouço de alto nível (o OpenACC), como também oferece ferramentas para depuração e ajuste para compilação de um programa específico (*tuning*).

O OpenARC foi desenvolvido com base no compilador Cetus, introduzido em [Dav+09], e por isso, alguns de seus benefícios e funcionalidades foram herdados pelo OpenARC. Entre eles, estão a representação intermediária de um programa, a interface `Traversable` e a interface `Annotation`. O primeiro *commit* da *branch master* existente no repositório disponibilizado data de abril de 2015, e é a release da versão 0.3 do OpenARC. Todos os *commits* disponibilizados na *branch master* até o *commit* de *sha* `c69d8d0e078146ddf2a6f` são de autoria do Dr. Seyong Lee. A maior parte da documentação no repositório está feita em comentários, muitos dos quais estão desatualizados. Há um pequeno arquivo README com instruções sobre a compilação do OpenARC. O uso desse compilador e de suas opções opções precisa ser deduzido principalmente a partir dos *scripts* de compilação e *Makefile*s disponíveis nos exemplos. Por esses motivos, utilizá-lo e entendê-lo por completo é bastante difícil; vale lembrar que um projeto do tamanho do OpenARC, de cerca de 220.000 linhas de código, que é desenvolvido por uma única

pessoa muito possivelmente terá seu desenvolvimento parado, além de não ter seu funcionamento revisado.

4.1 O funcionamento geral do OpenARC

Vamos aqui fornecer um panorama geral e de alto nível do processo de compilação realizado pelo OpenARC, ressaltando alguns dos principais momentos e passadas realizadas por ele. Para melhor proveito desta seção, recomenda-se clonar o repositório do OpenARC, e acompanhar com o código aberto. Mais especificamente, a escrita desta seção foi feita com base no *commit* cujo *sha* é `c69d8d0e078146ddf2a6f`. Para baixar o repositório do OpenARC, disponível em `https://code.ornl.gov/f6l/OpenARC`, é preciso possuir uma conta do GitLab do Laboratório Nacional Oak Ridge, o que pode ser obtido contatando `lee2@ornl.gov`. Abaixo, exibimos uma imagem, disponível em [Lee+21]. Ela exibe os principais componentes e as principais etapas executadas na compilação do OpenARC. Aqui, trataremos de etapas da compilação que na imagem são chamadas de *OpenARC Front-End* e *OpenARC Back-End*. Mais especificamente, tratamos das etapas indicadas por *C Parser*, do *Directive Parser*, do *General Optimizar*, do *Device Specifica Optimizer* e do *Kernels & Host Program Generator*.



Naturalmente, o processo começa com a função `main`, localizada no arquivo `ACC2GPUDriver.java`, instanciando um objeto da classe `ACC2GPUDriver`. Essa classe é utilizada independentemente da plataforma-alvo em que será executado o programa. Com um objeto de `ACC2GPUDriver` instanciado, o programa realiza o processo de *parse* do que foi digitado na linha de comando com a chamada ao método `parseCommandLine`, que faz a análise das opções fornecidas e nomes de arquivo fornecidos pelo usuário. Depois são "parseadas" as diretivas definidas pelo usuário em `parseUserDirectiveFile` e opções de ajuste fino em `parseTuningConfig`. A maior parte do trabalho é então feita nos métodos `parseFiles` (chamado na linha 915 de `ACC2GPUDriver`), `setPasses` (chamado na linha 929 de `ACC2GPUDriver`), `runPasses` (chamado na linha 931 de `ACC2GPUDriver`) e `CodeGenPass.run` (chamado na linha 987 de `ACC2GPUDriver`), os quais detalhamos a seguir.

O método `parseFiles` instancia um objeto da classe `CetusCParser` para realizar seu trabalho e então itera sobre os arquivos listados para *parsear* cada um deles, chamando o método `parseFile` do objeto instanciado. A função `parseFile`, na realidade, não faz muito além de chamar `parseAntlr`, que realiza efetivamente o trabalho. Esse método cria a representação intermediária (RI) de cada arquivo (ou unidade de tradução) do programa, o que é feito em três etapas. Na primeira etapa, cada arquivo de cabeçalho (arquivos cujo nome termina com `.h`) é anotado com pragmas para definir seu início e seu fim. Na segunda etapa, um preprocessador de C, externo ao OpenARC, é utilizado. Na terceira etapa, ocorre o *parseamento* propriamente dito utilizando objetos de `NewCLexer` e `NewCParser`. O `NewCLexer` fica responsável pelo registro do léxico, isto é, das palavras reservadas, e pela *tokenização* (segmentação do arquivo em elementos que possuem significado com base no léxico definido) do arquivo. Objetos de `NewCParser` , por outro lado, ficam responsáveis pela criação de uma primeira RI de cada arquivo, o que é feito modificando a unidade de tradução (um arquivo) fornecida a ele. É importante ressaltar que unidades de tradução implementam a interface `Traversable`, o que permite a iteração de diferentes formas em cada uma das futuras passadas do compilador por elas, entre essas formas estão o padrão de percorrimento em largura e em profundidade, por exemplo.

Voltando ao método `parseFiles`, uma segunda passagem para o *parse* de anotações (como comentários e diretivas) é executada. Essa passagem é realizada no método `start`, de `AnnotationParser`. Essa segunda passada ocorre da seguinte maneira: iteramos sobre as unidades de tradução do programa, cada unidade é percorrida em profundidade. Quando encontramos uma anotação, seja ela um comentário ou diretiva, substituímos o nó que a representava por um novo, da classe `Annotation` e *parseado*. Além disso, pragmas acionam o valor booleano `attach_to_next_annotatable`, o que faz com que o próximo objeto da classe `Statement` (como objetos da classe `ForLoop`) encontrado seja anexado ao pragma. É por esse motivo que o padrão de

travessia adequado aqui é o padrão em profundidade. Isso porque precisamos que o que anexar-se-á ao pragma seja seu descendente na árvore sintática que representa o programa.

O método `setPasses` é bastante simples, ele configura algumas variáveis (armazenadas em algo parecido com uma tabela de símbolos) sobre as passadas com base em opções fornecidas como argumento de linha de comando para o compilador. Já `runPasses`, executa um conjunto de passadas pelo código; nesse caso, estamos falando de passadas que dizem respeito às opções que foram passadas como argumento de linhas de comando.

Na linha 987 do arquivo `ACC2GPUDriver.java`, realizamos as transformações finais, para obter o código gerado pelo OpenARC, que pode ser em CUDA ou OpenCL. Dentre essas transformações finais, destacamos três: a privatização de variáveis (linha 606 de `acc2gpu.java`), reconhecimento de reduções (linha 610 de `acc2gpu.java`) e suas respectivas transformações no código de saída.

A privatização de variáveis consiste na técnica de paralelização que limita o escopo de uma variável a um certo contexto de execução. Se estamos falando de OpenACC, podemos estar nos referindo a uma variável que só será utilizada dentro de uma gangue, ou trabalhador; no ambiente do OpenCL, estamos falando de variáveis a que apenas um certo grupo de trabalho ou item de trabalho tem acesso. Esse tipo de técnica é importante tanto para a corretude do programa sendo escrito quanto para sua eficiência. No que diz respeito à corretude, limitar o acesso de "entidades de execução" a variáveis evita condições de corrida e inconsistências nos dados causadas por escritas sucessivas nos dados. Quanto à melhora do desempenho do programa, essa é decorrência do fato de que memórias mais restritas costumam estar fisicamente mais próximas aos componentes que acessam seus conteúdo e do fato de que são mais rápidas mais rápidas. A privatização de variáveis no OpenARC ocorre iterando sobre cada uma das funções definidas num programa (criando para isso um iterador em profundidade `DFIterator`). Para cada uma das funções, analisa-se cada um de seus laços, com os métodos `analyzeProcedure` (linha 204 de `ArrayPrivatization.java`) e `analyzeLoop` (linha 228 de `ArrayPrivatization.java`). Depois de `analyzeLoop`, são adicionados os resultados das passagens com `addAnnotation` para sinalização de variáveis que devem ser `firstprivate` ou `private`, o que é especificamente feito em `postPrivatize`, em `ArrayPrivatization.java`.

Reconhecimento de reduções (explicadas em 2.3), assim como a privatização, também é uma importante técnica para paralelização de um programa. A privatização no OpenARC começa com a classe `ACCReduction`, que por sua vez instancia um objeto de `Reduction`, que tão logo invoca seu método `start`. Esse método itera sobre cada um dos `ForLoop`s do programa e, para cada um deles, sobre suas expressões (isso é feito em `analyzeStatement`, na linha 94 de `Reduction.java`). Expressões de atribuição, chamadas de funções e expressões unárias são investigadas para busca de

variáveis candidatas a redução. Determinadas quais variáveis são passíveis de redução, associamos ela a um operador e conjunto de variáveis de redução, ainda dentro de `analyzeStatement`. Voltando a `start` de `Reduction`, cada laço é marcado com uma anotação, construída a partir da associação devolvida por `analyzeStatement`. Finalmente, terminada a execução do método `Reduction.start`, as anotações de diretivas `loop` e `kernels` do OpenACC (pertencentes a `ACCAnnotation`) são acrescidas com as informações sobre reduções que `Reduction.start` obteve.

Voltando agora a `acc2gpu.java`, observa-se a transformação final realizada pelo OpenARC, que transforma OpenACC num programa escrito no arcabouço escolhido, como CUDA ou OpenCL. Essa última passada ocorre a partir da linha 838 do arquivo `acc2gpu.java`. Antes de proceder à tradução efetiva, o programa escolhe qual tradutor vai utilizar com base na variável `OPENARC_ARCH`, configurada na compilação do OpenARC. Trataremos aqui do caso em que o sistema-alvo, no qual o programa será executado, é um FPGA Intel Altera, o que ocorre quando `OPENARC_ARCH` vale 3. Nesse caso, instanciamos um objeto de `ACC2OPENCLTranslator` e utilizamos seu método `start` para realizar a tradução (linhas 902 e 903 de `acc2gpu.java`). A instanciação desse objeto realiza uma parte da configuração de variáveis e de objetos que serão posteriormente utilizados na tradução propriamente dita; nesse sentido, o construtor dessa classe faz com que o objeto armazene em si um conjunto de variáveis escolhidas pelo usuário na invocação do compilador via linha de comando, o que é feito invocando o construtor da classe pai de `ACC2OPENCLTranslator`, `ACC2GPUTranslator`. Feita essa configuração, o construtor de `ACC2OPENCLTranslator` invoca `OpenCLInitialize`, que é responsável por encontrar a função `main` do programa sendo compilado. Além disso, esse método adiciona cabeçalhos do OpenARC, com funções definidas na pasta `openarcrt`, e chamadas a funções desse diretório para inicialização, exploração da plataforma e término do programa; os nomes das funções adicionadas são prefixados com `HI_` e mostraremos algumas de suas implementações mais adiante.

Mais uma vez de volta a `acc2gpu.java`, passamos à linha 903, na qual chama-se o método `start` do objeto de `ACC2OPENCLTranslator`, herdado de `ACC2GPUTranslator`. Esse método itera sobre as várias anotações construídas nas passadas anteriores, separando-as entre anotações de gerenciamento de memória, de chamadas de funções, de operações atômicas e de regiões de computação. Feita essa separação, itera-se sobre todas as definições de função (encapsuladas em objetos da classe `Procedure`) e é feita a tradução para o OpenCL. Dentre as transformações realizadas, destacam-se as chamadas às funções `handleEnterExitData` e `convComputeRegionsToGpuKernels` (linhas 393 e 396 de `ACC2GPUTranslator`, respectivamente).

A função `handleEnterExitData` faz a tradução entre as diretivas de dados do OpenACC e as funções para gerenciamento de memória do OpenCL. Para isso, cada uma das anotações de dados é analisada e transformada numa chamada de

função definida no OpenARC (linhas 1301 a 1320 de `ACC2GPUTranslator`); uma cláusula `copyin` é substituída por uma chamada a `acc_copyin_async_wait`, cláusulas `create` são traduzidas em chamadas a `acc_create_async_wait`, e assim por diante. Essas funções estão definidas no diretório `openarcrt`, no arquivo `openacc.cpp` e são como *wrappers* para a API do OpenCL. Cada uma das chamadas de função fica associada a uma lista de argumentos, que contem o tamanho do objeto com que ela lidará, um ponteiro, se a chamada será assíncrona ou não, entre outros detalhes (linhas 1321 a 1386 de `ACC2GPUTranslator`).

Finalmente, `convComputeRegionsToGpuKernels` transforma definições de funções em *kérneis*, como sugere seu nome. Essa transformação pode ser grosso modo separada em 4 partes importantes. A primeira delas é iniciada na linha 1956 do arquivo `ACC2GPUTranslator.java` e termina na linha 2004; nesse trecho de código, o programa itera sobre as anotações e, caso sejam anotações sobre paralelismo que vieram de diretivas `parallel` ou `kernels` do OpenACC, gera um elemento da RI representando o ponto ótimo (segundo os autores) de configuração do *kernel*. A segunda parte inicia-se na 2010 e vai até a linha 2116 do mesmo arquivo; esse trecho de código fica encarregado de mover cláusulas `private` ou `reduction` que estiverem aninhadas em laços para os laços mais externos o possível, mantendo a corretude do código. Num terceiro trecho, destaca-se em específico a função `extractComputeRegion`, responsável por escrever o código de um *kernel* num arquivo dedicado, por determinar a que região de memória pertencem os argumentos da função e como são compartilhados, pela configuração do NDRange do *kernel*, pela inserção de chamadas de sincronização, entre outros aspectos.

4.2 Estudo de caso

Vamos nesta seção comparar um programa fornecido como entrada para o OpenARC com a saída produzida por ele. Mais especificamente, vamos examinar um programa escrito em C e OpenACC e a saída produzida pelo OpenARC, que utiliza o OpenCL.

Para nossa comparação, vamos refletir sobre um programa um escrito em OpenACC para multiplicar duas matrizes. Vejamos o programa que serve de entrada para o OpenARC. Ele é composto de um único arquivo e duas funções. Abaixo, temos a função principal, na qual alocamos memória na máquina anfitriã e populamos as matrizes.

```
1 #include <stdlib.h>
2 #include <sys/time.h>
3 #include <stdio.h>
4 #include <math.h>
```

```
#ifdef _OPENACCM
#include <openacc.h>
#endif

#ifndef _N_
#define _N_ 512
#endif

int N = _N_;
int M = _N_;
int P = _N_;

int main() {
  float *a, *b, *c;
  int i, j;

  a = (float *) malloc(M * N * sizeof(float));
  b = (float *) malloc(M * P * sizeof(float));
  c = (float *) malloc(P * N * sizeof(float));

  for (i = 0; i < M * N; i++)
    a[i] = (float) 0.0F;
  for (i = 0; i < M * P; i++)
    b[i] = (float) i;
  for (i = 0; i < P * N; i++)
    c[i] = (float) 1.0F;

  MatrixMultiplication_openacc(a,b,c);
  free(a);
  free(b);
  free(c);

  return 0;
```

```
38 }
```

Ressaltamos a inclusão do cabeçalho `openacc.h`, que está definido, assim como o seu respectivo `.cpp`, no diretório `openarcrt`, na raiz do diretório do OpenARC. Nesse arquivo, há uma série de funções utilizadas pelo OpenARC para transformar diretivas do OpenACC em chamadas de funções da API do OpenCL.

Agora vejamos a função que efetivamente realiza a multiplicação de matrizes, definida no mesmo arquivo:

```
1  void MatrixMultiplication_openacc(float * a, float * b, float *
       ↪ c) {
2    int i, j, k;
3    #ifdef _OPENACCM
4    acc_init(acc_device_default);
5    #endif
6    #pragma acc kernels loop independent gang worker collapse(2)
         ↪ copyout(a[0:(M * N)]), copyin(b[0:(M * P)], c[0:(P * N
         ↪ )])
7    for (i = 0; i < M; i++)
8      for (j = 0; j < N; j++) {
9        float sum = 0.0 ;
10       #pragma acc loop seq
11       for (k = 0; k < P; k++)
12         sum += b[i * P + k] * c[k * N + j];
13       a[i * N + j] = sum;
14     }
15
16   #ifdef _OPENACCM
17   acc_shutdown(acc_device_default);
18   #endif
19 }
```

Em primeiro lugar, realizamos uma chamada a `acc_init` (linha 4), declarada em `openacc.h`. Essa função configura algumas variáveis para uso futuro, como número de *threads*, tamanho de grupos de trabalho, tipo de acelerador, número de dispositivos, entre outros aspectos relacionados ao sistema que executará o programa. Além disso, essa função constrói um objeto do tipo `HostConf`, que é uma classe

definida em `openaccrt_ext.h` de `openarcrt`. Objetos dessa classe armazenam em si informações como nomes de *kérneis* a serem executados e seus respectivos códigos e listas de parâmetros.

Depois, merecem atenção os dois pragmas utilizados. Em `#pragma acc kernels loop independent gang worker collapse(2)copyout(a[0:(M * N)]), copyin(b[0:(M * P)], c[0:(P * N)])`, temos:

- `kernels` dá ao compilador (o OpenARC) a liberdade de encontrar paralelizações e otimizações que julgar eficientes no bloco de código a seguir;
- `loop` indica explicitamente a paralelização do loop seguinte;
- `independent` indica a independência total entre iterações do laço seguinte, de tal maneira que o compilador não faça análise de dependência entre iterações;
- `gang` compartilha as iterações de um laço entre as diferentes gangues que os estão executando;
- `worker` compartilha as iterações de um laço entre as diferentes trabalhadores de uma certa gangue;
- `collapse(2)` utiliza a mesma diretiva para os 2 laços depois dela, o laço indexado por `i` e o indexado por `j`. Isso é feito transformando os dois laços num único laço que possui `M * N` iterações.

As cláusulas de gerenciamento de dados foram explicadas em 2.5.

O segundo pragma a que damos destaque é `#pragma acc loop seq`, que pede a execução sequencial do laço seguinte, o que talvez suscite a pergunta do por quê não utilizar uma cláusula de redução. Nesse caso, o uso da cláusula redução não faz diferença, visto que, com ou sem ela, o *kernel* gerado pelo OpenARC é o mesmo (vamos mostrá-lo a seguir).

Agora vejamos o código gerado pelo OpenACC. Ele está dividido em dois arquivos, um com o código do anfitrião, escrito para linguagem C++, e um outro com o código do *kernel*. Vamos exibir o código tal qual ele é produzido, em particular, não há indentação. O código do anfitrião é o seguinte:

```
1 int main()
2 {
3 float * a;
4 float * b;
```

```
5  float * c;
6  int i;
7  int _ret_val_0 = 0;
8
9  //////////////////////////
10 // Device Initialization //
11 //////////////////////////
12
13 std::string kernel_str[1];
14 kernel_str[0]="MatrixMultiplication_openacc_kernel0";
15 acc_init(acc_device_default, 1, kernel_str, 4, "openarc_kernel")
       ↪ ;
16 a=((float *)malloc(((M*N)*sizeof (float))));
17 b=((float *)malloc(((M*P)*sizeof (float))));
18 c=((float *)malloc(((P*N)*sizeof (float))));
19 for (i=0; i<(M*N); i ++ )
20 {
21 a[i]=((float)0.0F);
22 }
23 for (i=0; i<(M*P); i ++ )
24 {
25 b[i]=((float)i);
26 }
27 for (i=0; i<(P*N); i ++ )
28 {
29 c[i]=((float)1.0F);
30 }
31 MatrixMultiplication_openacc(a, b, c);
32 free(a);
33 free(b);
34 free(c);
35 _ret_val_0=0;
36 acc_shutdown(acc_device_default);
```

```
37 return _ret_val_0;
38 }
```

Também como código do anfitrião, temos a definição de `MatrixMultiplication_openacc`:

```
1  void MatrixMultiplication_openacc(float * a, float * b, float *
       ↪ c)
2  {
3  float * gpu__b;
4  float * gpu__c;
5  float * gpu__a;
6  gpuBytes=(sizeof (float)*(M*P));
7  HI_malloc1D(b, ((void * *)( & gpu__b)), gpuBytes, DEFAULT_QUEUE,
       ↪  HI_MEM_READ_WRITE);
8  HI_memcpy(gpu__b, b, gpuBytes, HI_MemcpyHostToDevice, 0);
9  gpuBytes=(sizeof (float)*(N*P));
10 HI_malloc1D(c, ((void * *)( & gpu__c)), gpuBytes, DEFAULT_QUEUE,
       ↪  HI_MEM_READ_WRITE);
11 HI_memcpy(gpu__c, c, gpuBytes, HI_MemcpyHostToDevice, 0);
12 gpuBytes=(sizeof (float)*(M*N));
13 HI_malloc1D(a, ((void * *)( & gpu__a)), gpuBytes, DEFAULT_QUEUE,
       ↪  HI_MEM_READ_WRITE);
14 size_t dimGrid_MatrixMultiplication_openacc_kernel0[3];
15 dimGrid_MatrixMultiplication_openacc_kernel0[0]=((int)ceil((((
       ↪ float)(M*N))/32.0F)));
16 dimGrid_MatrixMultiplication_openacc_kernel0[1]=1;
17 dimGrid_MatrixMultiplication_openacc_kernel0[2]=1;
18 size_t dimBlock_MatrixMultiplication_openacc_kernel0[3];
19 dimBlock_MatrixMultiplication_openacc_kernel0[0]=32;
20 dimBlock_MatrixMultiplication_openacc_kernel0[1]=1;
21 dimBlock_MatrixMultiplication_openacc_kernel0[2]=1;
22 gpuNumBlocks=((int)ceil((((float)(M*N))/32.0F)));
23 gpuNumThreads=32;
24 totalGpuNumThreads=(((int)ceil((((float)(M*N))/32.0F)))*32);
```

```
25 HI_register_kernel_numargs("MatrixMultiplication_openacc_kernel0
       ↪ ",6);
26 HI_register_kernel_arg("MatrixMultiplication_openacc_kernel0",0,
       ↪ sizeof(void*),( & gpu__a),1,2,sizeof (float));
27 HI_register_kernel_arg("MatrixMultiplication_openacc_kernel0",1,
       ↪ sizeof(void*),( & gpu__b),1,0,sizeof (float));
28 HI_register_kernel_arg("MatrixMultiplication_openacc_kernel0",2,
       ↪ sizeof(void*),( & gpu__c),1,0,sizeof (float));
29 HI_register_kernel_arg("MatrixMultiplication_openacc_kernel0",3,
       ↪ sizeof (int),( & M),0,0,sizeof (int));
30 HI_register_kernel_arg("MatrixMultiplication_openacc_kernel0",4,
       ↪ sizeof (int),( & N),0,0,sizeof (int));
31 HI_register_kernel_arg("MatrixMultiplication_openacc_kernel0",5,
       ↪ sizeof (int),( & P),0,0,sizeof (int));
32 HI_kernel_call("MatrixMultiplication_openacc_kernel0",
       ↪ dimGrid_MatrixMultiplication_openacc_kernel0,
       ↪ dimBlock_MatrixMultiplication_openacc_kernel0,
       ↪ DEFAULT_QUEUE,0,NULL);
33 HI_synchronize(0);
34 gpuNumBlocks=((int)ceil((((float)(M*N))/32.0F)));
35 gpuBytes=(sizeof (float)*(M*N));
36 HI_memcpy(a, gpu__a, gpuBytes, HI_MemcpyDeviceToHost, 0);
37 HI_free(a, DEFAULT_QUEUE);
38 HI_free(c, DEFAULT_QUEUE);
39 HI_free(b, DEFAULT_QUEUE);
40 return ;
41 }
```

Conforme explicado no fim do capítulo anterior, grande parte do código que se escreve em OpenCL é repetitivo e serve para inicialização ou finalização do ambiente de programação. A função acima, que foi em grande parte gerada automaticamente pelo OpenARC para realizar a inicialização e finalização do ambiente do OpenCL, mostra que esses trechos de código são passíveis de automação. Vejamos como estão definidas algumas das funções chamadas acima.

A função `HI_malloc1D`, chamada no código da função de multiplicação de ma-

trizes, na linha 10, está definida no arquivo `openaccrt.cpp`, no diretório `openarcrt`. Ela começa obtendo a configuração do anfitrião, encapsulada num objeto da classe `HostConf` chamado `tconf` e faz uma chamada para `tconf->device->HI_malloc1D`, responsável por de fato realizar a alocação. Note como o uso desse ponteiro, possibilita que a função ser chamada seja determinada em tempo de execução; em particular, isso permite que o OpenARC gere saídas bastante parecidas, ainda que os dispositivos nos quais essas saídas serão executadas sejam diferentes. No caso de código gerado utilizando OpenCL, que pode ser executado em FPGAs da Intel, a implementação da função (definida na linha 1160 de `openarcrt/opencldriver.cpp`, a partir da raiz do diretório em que o OpenARC foi clonado) citada é a seguinte:

```
1  HI_error_t  OpenCLDriver::HI_malloc1D(const void *hostPtr, void
       ↪ **devPtr, size_t count, int asyncID, HI_MallocKind_t
       ↪ flags, int threadID) {
2  #ifdef _OPENARC_PROFILE_
3      if( HI_openarcrt_verbosity > 2 ) {
4          fprintf(stderr, "[OPENARCRT-INFO]\t\tenter OpenCLDriver
               ↪ ::HI_malloc1D(hostPtr = %lx, asyncID = %d, size
               ↪ = %lu, flags = %d, thread ID = %d)\n",(long
               ↪ unsigned int)hostPtr, asyncID, count, flags,
               ↪ threadID);
5      }
6  #endif
7      HostConf_t * tconf = getHostConf(threadID);
8
9      if( tconf->device->init_done == 0 ) {
10         //fprintf(stderr, "[in HI_malloc1D()] : initing!\n");
11         tconf->HI_init(DEVICE_NUM_UNDEFINED);
12     }
13 #ifdef _OPENARC_PROFILE_
14     double ltime = HI_get_localtime();
15 #endif
16     HI_error_t result = HI_error;
17     cl_int  err = CL_SUCCESS;
18     void * memHandle;
```

```
19      cl_mem_flags mem_flags = convert2CLMemFlags(flags);
20
21      if(HI_get_device_address(hostPtr, devPtr, NULL, NULL,
            ↪ asyncID, tconf->threadID) == HI_success ) {
22          if( unifiedMemSupported ) {
23              result = HI_success;
24          } else {
25              fprintf(stderr, "[ERROR in OpenCLDriver::HI_malloc1D
                    ↪ ()] Duplicate device memory allocation for
                    ↪ the same host data (%lx) by thread %d is not
                    ↪  allowed; exit!\n",(long unsigned int)
                    ↪ hostPtr, tconf->threadID);
26              exit(1);
27          }
28      } else {
29          memPool_t *memPool = memPoolMap[tconf->threadID];
30          std::multimap<size_t, void *>::iterator it = memPool->
                ↪ find(count);
31          if( it != memPool->end()) {
32  #ifdef _OPENARC_PROFILE_
33              if( HI_openarcrt_verbosity > 2 ) {
34                  fprintf(stderr, "[OPENARCRT-INFO]\t\
                        ↪ tOpenCLDriver::HI_malloc1D(%d, %lu, %d)
                        ↪ reuses memories in the memPool\n",
                        ↪ asyncID, count, flags);
35              }
36  #endif
37              *devPtr = it->second;
38              memPool->erase(it);
39              HI_set_device_address(hostPtr, *devPtr, count,
                    ↪ asyncID, tconf->threadID);
40          } else {
41              memHandle = (void*) clCreateBuffer(clContext,
```

```
                ↪ mem_flags, count, NULL, &err);
#ifdef _OPENARC_PROFILE_
            tconf->IDMallocCnt++;
            tconf->IDMallocSize += count;
#endif
            if(err != CL_SUCCESS) {
                //fprintf(stderr, "[ERROR in OpenCLDriver::
                    ↪ HI_malloc1D()] : Malloc failed\n");
#ifdef _OPENARC_PROFILE_
                if( HI_openarcrt_verbosity > 2 ) {
                    fprintf(stderr, "[OPENARCRT-INFO]\t\
                        ↪ tOpenCLDriver::HI_malloc1D(%d, %lu,
                        ↪ %d) releases memories in the memPool
                        ↪ \n", asyncID, count, flags);
                }
#endif
                HI_device_mem_handle_t tHandle;
                void * tDevPtr;
                for (it = memPool->begin(); it != memPool->end()
                    ↪ ; ++it) {
                    tDevPtr = it->second;
                    if( HI_get_device_mem_handle(tDevPtr, &
                        ↪ tHandle, tconf->threadID) ==
                        ↪ HI_success ) {
                        err = clReleaseMemObject((cl_mem)(
                            ↪ tHandle.basePtr));
#ifdef _OPENARC_PROFILE_
                        tconf->IDFreeCnt++;
#endif
                        if(err != CL_SUCCESS) {
                            fprintf(stderr, "[ERROR in
                                ↪ OpenCLDriver::HI_malloc1D()]
                                ↪  : failed to free on OpenCL\
```

```
                                    ↪ n");
64                          }
65                          free(tDevPtr);
66                          HI_remove_device_mem_handle(tDevPtr,
                                ↪ tconf->threadID);
67                      }
68                  }
69                  memPool->clear();
70                  memHandle = (void*) clCreateBuffer(clContext,
                        ↪ mem_flags, count, NULL, &err);
71              }
72              if(err == CL_SUCCESS) {
73  #if defined(OPENARC_ARCH) && OPENARC_ARCH == 3
74                  if( HI_openarcrt_memoryalignment > 0 ) {
75                      posix_memalign(devPtr, AOCL_ALIGNMENT, count
                            ↪ );
76                  } else {
77                      *devPtr = malloc(count); //redundant malloc
                            ↪ to create a fake device pointer.
78                  }
79  #else
80                  *devPtr = malloc(count); //redundant malloc to
                        ↪ create a fake device pointer.
81  #endif
82                  if( *devPtr == NULL ) {
83                      fprintf(stderr, "[ERROR in OpenCLDriver::
                            ↪ HI_malloc1D()] :fake device malloc
                            ↪ failed\n");
84                      exit(1);
85                  }
86                  HI_set_device_address(hostPtr, *devPtr, count,
                        ↪ asyncID, tconf->threadID);
87                  HI_set_device_mem_handle(*devPtr, memHandle,
```

```
                    ↪ count, tconf->threadID);
88              }
89          }
90          if(err == CL_SUCCESS) {
91              result = HI_success;
92          } else {
93              fprintf(stderr, "[ERROR in OpenCLDriver::HI_malloc1D
                    ↪ ()] : Malloc failed\n");
94              exit(1);
95          }
96      }
97  #ifdef _OPENARC_PROFILE_
98      tconf->totalMallocTime += HI_get_localtime() - ltime;
99  #endif
100 #ifdef _OPENARC_PROFILE_
101     if( HI_openarcrt_verbosity > 2 ) {
102         HI_print_device_address_mapping_summary(tconf->threadID)
                ↪ ;
103         fprintf(stderr, "[OPENARCRT-INFO]\t\texit OpenCLDriver::
                ↪ HI_malloc1D(hostPtr = %lx, asyncID = %d, size =
                ↪ %lu, flags = %d, thread ID = %d)\n",(long
                ↪ unsigned int)hostPtr, asyncID, count, flags,
                ↪ threadID);
104     }
105 #endif
106     return result;
107
108 }
```

Essa função primeiro verifica se o ponteiro com dados do anfitrião `hostPtr` possui uma entrada na tabela de endereços , para ponteiros que ainda não foram alocados (o que é visto na condição que utiliza `HI_get_device_address`, na linha 21), chama a função `clCreateBuffer` (linha 41), nativa do OpenCL e responsável pela criação de um espaço de memória para compartilhamento de memória, conforme explicado na seção 3.3. Para ponteiros que já foram alocados, ela devolve sucesso.

Outro trecho de interesse no código do anfitrião é o que registra os argumentos dos *kérneis*, entre as linhas 25 e 31, e a chamada ao *kernel*, na linha 32. A função de registro dos argumentos dos *kérneis*, essencialmente chama o método `HI_register_kernel_arg` (definida na linha 2579 de `opencldriver.cpp`), de `tconf->device`, dado a seguir para o caso em que temos o OpenCL como linguagem alvo para o código:

```
1  HI_error_t OpenCLDriver::HI_register_kernel_arg(std::string
       ↪ kernel_name, int arg_index, size_t arg_size, void *
       ↪ arg_value, int arg_type, int arg_trait, size_t unitSize,
       ↪  int threadID)
2  {
3  #ifdef _OPENARC_PROFILE_
4      if( HI_openarcrt_verbosity > 2 ) {
5          fprintf(stderr, "[OPENARCRT-INFO]\t\tenter OpenCLDriver
               ↪ ::HI_register_kernel_arg()\n");
6      }
7  #endif
8      HostConf_t * tconf = getHostConf(threadID);
9      kernelParams_t * kernelParams = tconf->kernelArgsMap.at(this
           ↪ ).at(kernel_name);
10     cl_int err;
11     if( arg_type == 0 ) { //scalar variable
12         err = clSetKernelArg((cl_kernel)(tconf->kernelsMap.at(
               ↪ this).at(kernel_name)), arg_index, arg_size,
               ↪ arg_value);
13         *(kernelParams->kernelParams + arg_index) = arg_value;
14         *(kernelParams->kernelParamsOffset + arg_index) = 0;
15         *(kernelParams->kernelParamsInfo + arg_index) = (int)
               ↪ arg_size;
16         *(kernelParams->kernelParamSubBuffers + arg_index) =
               ↪ NULL;
17     } else { //pointer variable
18         HI_device_mem_handle_t tHandle;
19         size_t dataSize = 0;
```

```
20          if( HI_get_device_mem_handle(*((void **)arg_value), &
                ↪ tHandle, &dataSize, tconf->threadID) ==
                ↪ HI_success ) {
21              void * localBasePtr = tHandle.basePtr;
22              if( tHandle.offset == 0 ) {
23                  *(kernelParams->kernelParams + arg_index) =
                        ↪ localBasePtr;
24                  *(kernelParams->kernelParamsOffset + arg_index)
                        ↪ = 0;
25                  *(kernelParams->kernelParamsInfo + arg_index) =
                        ↪ arg_trait;
26                  *(kernelParams->kernelParamSubBuffers +
                        ↪ arg_index) = NULL;
27                  err = clSetKernelArg((cl_kernel)(tconf->
                        ↪ kernelsMap.at(this).at(kernel_name)),
                        ↪ arg_index, arg_size, &localBasePtr);
28              } else {
29                  size_t localOffset = tHandle.offset;
30                  *(kernelParams->kernelParams + arg_index) =
                        ↪ localBasePtr;
31                  *(kernelParams->kernelParamsOffset + arg_index)
                        ↪ = localOffset/unitSize;
32                  *(kernelParams->kernelParamsInfo + arg_index) =
                        ↪ arg_trait;
33                  cl_buffer_region localRegion;
34                  localRegion.origin = localOffset;
35                  localRegion.size = dataSize - localOffset;
36                  cl_mem_flags flags;
37                  if( arg_trait == 0 ) { //read-only
38                      flags = CL_MEM_READ_ONLY;
39                  } else if( arg_trait == 1 ) { //write-only
40                      flags = CL_MEM_WRITE_ONLY;
41                  } else if( arg_trait == 2 ) { //read-write
```

```
42                      flags = CL_MEM_READ_WRITE;
43                  } else { //unknown or temporary
44                      flags = CL_MEM_READ_WRITE;
45                  }
46                  cl_mem subBuffer = clCreateSubBuffer((cl_mem)
                        ↪ localBasePtr, flags,
                        ↪ CL_BUFFER_CREATE_TYPE_REGION, &
                        ↪ localRegion, &err);
47                  *(kernelParams->kernelParamSubBuffers +
                        ↪ arg_index) = subBuffer;
48                  err = clSetKernelArg((cl_kernel)(tconf->
                        ↪ kernelsMap.at(this).at(kernel_name)),
                        ↪ arg_index, arg_size, &subBuffer);
49              }
50          } else {
51              fprintf(stderr, "[ERROR in OpenCLDriver::
                    ↪ HI_register_kernel_arg()] Cannot find a
                    ↪ device pointer to memory handle mapping;
                    ↪ failed to add argument %d to kernel %s (
                    ↪ OPENCL Device)\n", arg_index, kernel_name.
                    ↪ c_str());
52  #ifdef _OPENARC_PROFILE_
53              HI_print_device_address_mapping_entries(tconf->
                    ↪ threadID);
54  #endif
55              exit(1);
56          }
57      }
58      if(err != CL_SUCCESS)
59      {
60          fprintf(stderr, "[ERROR in OpenCLDriver::
                ↪ HI_register_kernel_arg()] failed to add argument
                ↪  %d to kernel %s with error %d (%s)\n",
```

```
            ↪ arg_index, kernel_name.c_str(), err,
            ↪ opencl_error_code(err));
61          exit(1);
62          return HI_error;
63      }
64  #ifdef _OPENARC_PROFILE_
65      if( HI_openarcrt_verbosity > 2 ) {
66          fprintf(stderr, "[OPENARCRT-INFO]\t\texit OpenCLDriver::
            ↪ HI_register_kernel_arg()\n");
67      }
68  #endif
69      return HI_success;
70  }
```

A função chamada pelo OpenARC antes de efetivamente registrar o argumento no *kernel* atualiza suas estruturas, encapsuladas em `tconf`, e depois faz o registro de fato. De toda forma, todos os ramos de execução dessa função que são bem-sucedido chamam, de uma forma ou de outra, `clSetKernelArg`, função também nativa do OpenCL para registro de argumentos.

Com esses dois trechos de código, queremos apontar para o fato de que as funções introduzidas pelos autores do OpenARC em seu âmago fazem chamadas a funções nativas do OpenCL, ainda que sejam mais rebuscadas, façam mais verificações e guardem mais informação do que guardaria um código escrito por uma pessoa. Isso vai ao encontro da ideia de que a configuração do ambiente OpenCL é código *boilerplate* e portanto passível de automação.

Por fim, apresentamos o *kernel* produzido pelo OpenARC a partir do programa escrito em OpenACC. Vamos aqui mostrá-lo formatado, mas o OpenARC não o escreve com indentações.

```
1   __kernel void __attribute__((reqd_work_group_size(32, 1, 1)))
        ↪ MatrixMultiplication_openacc_kernel0(__global float *
        ↪ restrict a, __global float * restrict b, __global float
        ↪ * restrict c, int M, int N, int P)
2   {
3       int lwpriv___ti_100_0;
4       float sum;
5       int lwpriv__i;
```

```
6       int lwpriv__j;
7       int lwpriv__k;
8       lwpriv___ti_100_0=get_global_id(0);
9       if (lwpriv___ti_100_0<(M*N))
10      {
11          sum=0.0;
12          lwpriv__j=(lwpriv___ti_100_0%N);
13          lwpriv__i=(lwpriv___ti_100_0/N);
14          for (lwpriv__k=0; lwpriv__k<P; lwpriv__k ++ )
15          {
16              sum+=(b[((lwpriv__i*P)+lwpriv__k)]*c[((lwpriv__k*N)+
                    ↪ lwpriv__j)]);
17          }
18          a[((lwpriv__i*N)+lwpriv__j)]=sum;
19      }
20  }
```

Esse *kernel* foi escrito num outro arquivo texto, diferente de onde está o código a ser executado pela máquina anfitriã. Conforme podemos observar, ele toma um índice `lwpriv___ti_100_0` retirado do NDRange (linha 8 do trecho acima) que deve estar entre 0 e `M * N - 1`. Com esse índice, o *kernel* calcula a entrada na linha `lwpriv___ti_100_0 / N` (linha 12) e coluna `lwpriv___ti_100_0 % N` (linha 13) da matriz resultado.

5 Experimentos

Neste capítulo, vamos executar 4 programas transpilados para OpenCL pelo OpenARC. Além disso, vamos comparar a execução desses programas escritos em OpenCL pelo OpenARC com a execução dos programas dados a ele como entrada, escritos em OpenACC. Para os programas em OpenCL, utilizamos a versão 9.4.0 do GCC para gerar os executáveis; já para os programas em OpenACC, utilizamos o compilador NVC fornecido na versão 23.9 do NVIDIA HPC SDK para gerar os executáveis. Com isso, comparamos a execução de um programa escrito em OpenCL com a execução de um programa equivalente escrito em OpenACC. Os programas utilizados estão disponíveis no repositório do OpenARC, no diretório `test`.

Em termos de *hardware*, executamos os *kérneis* dos programas escritos em OpenCL numa GPU da NVIDIA de modelo GP108M. Quanto às CPUs, foram utilizadas 8, do modelo Intel© Core ™ i7-8565U CPU @ 1.80GHz. Para utilizar o NVC para compilar os programas escritos em OpenACC, também foi necessário baixar o *driver* apropriado da NVIDIA. Foi baixada a versão 11.4.0 do CUDA Toolkit. Para instalar o NVIDIA HPC SDK, seguimos o que está descrito em [Cor23]; para instalar o CUDA Toolkit, utilizamos [Cor22].

Por fim, para realizar os experimentos, precisamos compilar o OpenARC para gerar os programas em OpenCL. Para isso, começamos definindo duas variáveis de ambiente: `openarc` e `OPENARC_ARCH`. A primeira constante armazena o caminho do diretório em que o repositório OpenARC foi clonado. Já `OPENARC_ARCH`, define a arquitetura em que o programa será executado e se a saída produzida será escrita em CUDA ou OpenCL; para os testes feitos, foi utilizado o valor 1 para essa constante, o que determina que estamos utilizando um sistema com alguma distribuição do Linux e que a saída produzida será escrita em OpenCL. Por fim, para compilar o OpenARC, basta executar `make` na raiz do seu repositório.

A tabela abaixo resume os resultados dos experimentos realizados. Os detalhes sobre cada um dos programas executados, sobre como cada um desses resultados foi obtido e sobre o que significam os tempos mostrados são dados nas seções a seguir.

Programas	CPU OpenMP	GPU OpenACC	GPU OpenCL
MatMul	88,35 s	0,88 s	0,81 s
Jacobi	100,79 s	16,17 s	16,85 s
VecAdd	0,018 s	0,34 s	0,28 s
Lulesh	—	41,80 s	4,97 s

5.1 Multiplicação de matrizes

Para a multiplicação de matrizes, utilizamos o programa `matmul`, no diretório `test/examples/openarc`, em relação ao diretório raiz do OpenARC. Esse programa multiplica duas matrizes quadradas de ordem 2048 e foi executado 10 vezes. Para compilar a entrada, que utiliza OpenMP (para executar nas CPUs) e OpenACC, com saída possuindo OpenMP e OpenCL, utiliza-se o script `O2GBuild.script`, que grosso modo executa

```
1 java -classpath ../../../../openarcrt/cetus.jar:../../../../lib/
    ↪ antlr.jar openacc.exec.ACC2GPUDriver -verbosity=0 -
    ↪ gpuConfFile=openarcConf.txt matmul.c
```

Conforme explicado na seção anterior, a execução do OpenARC gera dois arquivos, um com o código do anfitrião e outro do *kernel*. Para gerar um executável desses dois arquivos, executamos

```
1 g++ -D_N_=2048 -DOMP=0 -D_OPENACC=201306 -D OPENARC_ARCH=1 -O3
    ↪ -I $(openarc)/openarcrt   -I ../ -o .././bin/matmul_ACC
    ↪ matmul.cpp  -L $(openarc)/openarcrt -lopenaccrt_opencl -
    ↪ lomphelper -lOpenCL -lpthread
```

A execução utilizando as CPUs tem média igual a aproximadamente 88,35 segundos, com desvio padrão igual a 10,37 segundos. A execução utilizando os *kérneis* em OpenCL despachando o cálculo para a GPU leva em média 0,81 segundos, com desvio padrão de 0,003 segundos. Já a versão utilizando OpenACC e compilando com o NVC executando

```
1 nvc -acc -gpu=cuda11.4 -D_N_=2048 -DOMP=0 -D_OPENACC=201306 -D
    ↪ OPENARC_ARCH=1 -O3 matmul.c
```

teve tempo médio igual a 0,88 segundos com desvio padrão igual a 0,0002 segundos.

5.2 Jacobi

Considere $f : \mathbb{R}^n \to \mathbb{R}$. Definimos o Laplaciano de f, denotado por Δf ou $\nabla^2 f$, como div(grad$f$), que também tem domínio igual $\mathbb{R}^n$ e toma valores reais. Se $q : \mathbb{R}^n \to \mathbb{R}$, a equação de difusão é a equação diferencial parcial

$$\nabla^2 f + q = 0$$

Levando em consideração $n = 2$, podemos discretizar o domínio de f e de q com coordenadas inteiras; nesse sentido, f e q podem ser modeladas como matrizes sobre o conjunto de pares de coordenadas inteiras. Se A_0 é um palpite inicial para a solução f da equação, podemos obter uma solução mais próxima de f, A_k, obtida a partir de $k \geq 1$ iterações a partir de A_0, de acordo com o seguinte:

$$A_k[i][j] = \frac{A_{k-1}[i-1][j] + A_{k-1}[i][j-1] + A_{k-1}[i+1][j] + A_{k-1}[i][j+1] + q[i][j]}{4}$$

Como q é constante ao longo das iterações, podemos considerar $q = 0$ sem prejuízo significativo para a análise de desempenho do algoritmo que aproxima a solução da equação diferencial. Com isso, temos

$$A_k[i][j] = \frac{A_{k-1}[i-1][j] + A_{k-1}[i][j-1] + A_{k-1}[i+1][j] + A_{k-1}[i][j+1]}{4}$$

Para os experimentos, esse algoritmo foi executado 10 vezes com 10000 iterações em matrizes de ordem 2048. Para CPUs, obtivemos tempo de execução médio igual a 100,79 segundos e desvio padrão igual a 2,08 segundos. Executando numa GPU o programa escrito em OpenCL, o tempo médio de execução foi igual a 16,85 com desvio padrão igual a 0,11 segundos. Finalmente, executando numa GPU o programa escrito em OpenACC, obtivemos tempo médio de execução igual a 16,17 segundos e desvio padrão igual a 0,04 segundos.

5.3 Adição de vetores

Para um programa que realiza adição de vetores de tamanho $2^{26} = 67108864$ executado 10 vezes, obtivemos execução em GPU com tempos médios iguais a 0,28 segundos e 0,34 segundos, para os programas escritos em OpenCL e OpenACC, respectivamente. Os desvios padrão foram de 0,001 segundos e 0,002, na mesma ordem. Para execução em CPU obtivemos tempo médio menor, igual a 0,018 segundos e desvio padrão igual a 0,0003 segundos.

5.4 Lulesh

Lulesh é um acrônimo para *Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics* e é um conhecido *benchmark* para programação de alta performance desenvolvido no Laboratório Nacional Lawrence Livermore. Em particular, o problema trata da "modelagem de eventos envolvendo deformação severa utilizando a hidrodinâmica de choque Lagrangiana", conforme escrevem seus criadores em [HKG11]. Ainda de acordo com os autores, Lulesh resolve o problema da onda de choque de Sedov para um material em três dimensões, que se resume na resolução de um sistema de equações diferenciais parciais.

Mais importante do que a resolução do problema em si, é o fato de que o algoritmo proposto pelos autores, disponível em [Sch+], realiza computações e transferências de dados bastante comuns em programação de alto desempenho e pode ser escalado conforme a necessidade de se medir o desempenho de um programa.

Para a execução desse programa, utilizou-se uma versão sem modificações. Assim, temos apenas medidas de tempo fazendo o despacho de seções paralelas para GPU e não para CPUs. Com 10 execuções, obtivemos tempo médio de execução igual a 4,97 segundos e desvio padrão de 0,06 segundos para o programa produzido pelo OpenARC e tempo médio de 41,80 segundos e desvio padrão de 0,3 segundos para o programa escrito em OpenACC.

6 Conclusão

Com este trabalho, apresentamos três ferramentas abertas e multi-plataforma para programação em sistemas heterogêneos: uma de alto nível e fácil aprendizado, outra de mais baixo nível com mais controle sobre a execução e um transpilador que converte a primeira na segunda.

O OpenACC se mostra como ferramenta de uso simples e controle parcial sobre paralelismo, divisão do trabalho (em gangues, trabalhadores e grupos de trabalho), gerenciamento de dados e sincronização. Entretanto, apresenta limitações quanto à delegação de tarefas quando temos um sistema mais complexo.

O OpenCL é uma ferramenta completa e multi-plataforma que permite controle bastante granular sobre o despacho de trabalho e sincronização de tarefas, além de oferecer suporte para processamento de imagens. A contrapartida desse controle é a dificuldade de aprendizado e a produção de códigos possivelmente repetitivos e menos portáveis.

Assim, o OpenARC, que pode converter um programa em OpenACC num programa em OpenCL, se mostra como ferramenta bastante útil. Isso porque vimos que há casos em que a execução de um programa escrito em OpenACC pode ser bastante semelhante à execução de um programa escrito em OpenCL. É lógico que em certas situações esse não é o caso; é justamente nessas ocasiões que o OpenARC pode se mostrar como ferramente útil para programação. A pessoa programadora pode escrever seu primeiro programa em OpenACC e, caso precise de ajuste mais fino, pode transpilá-lo para OpenCL com o OpenARC, e então fazer o ajuste fino a partir do código produzido pelo OpenARC. Com isso, temos um esforço direcionado à otimização, e não à escrita de tarefas repetitivas e suscetíveis a erro; parte significativa do código foi produzida de maneira simples (na escrita de um programa em OpenACC) e a outra foi automatizada (com a transpilação pelo OpenARC) e no entanto obtém-se em mãos um programa correto de baixo nível, pronto para otimização mais sofisticada.

Mostramos portanto três ferramentas de fácil acesso e que têm utilidade para propósitos, programadores, programas, ou etapas de trabalho diferentes. Todas

elas mostram-se como alternativas que facilitam o trabalho de programar em sistemas heterogêneos utilizando plataformas abertas, direcionadas a uma variedade de dispositivos.

7 Apreciação Pessoal

Esse trabalho foi certamente um desafio. Eu e o professor Alfredo não tínhamos logo de início um projeto pronto ou qual caminho tomar. Foi apenas ao longo do desenvolvimento que o trabalho foi tomando forma e descobrimos qual seria exatamente o tema que ele teria e qual seria o nosso percurso. Além disso, conciliar esse último ano de faculdade com um trabalho presencial e cansativo demandou disciplina e resiliência.

A ideia do trabalho começou, na realidade, com o OpenARC e a possibilidade de escreve código de alto nível que pudesse ser executado num FPGA. Lendo os artigos sobre o OpenARC, percebi que seria interessante ter um entendimento maior sobre o OpenACC, um dos arcabouços a partir dos quais o OpenARC escreve programas.

Começar a estudar o OpenACC foi num certo sentido um alívio. Isso porque passei de estudar um transpilador com pouca documentação e material, para um arcabouço de uso simples que possui material vasto. Comecei fazendo um curso de treinamento oferecido pela OpenACC Organization e logo eu já conseguia escrever programas funcionais para serem executados na GPU da minha máquina pessoal. A sensação que tive foi a de estudar uma ferramenta poderosa, apesar de simples. Fiquei com a impressão de que essa ferramenta deveria ser muito mais difundida do que ela é; isso porque um mesmo programa simples poderia ser executado em diferentes plataformas sem precisar de anos de treinamento e com desempenho bastante eficiente. Tão logo decidi começar a escrever a monografia expondo o OpenACC. Para isso, busquei material mais extenso, li boa parte da especificação do OpenACC e um par de livros que encontrei pesquisando sobre esse arcabouço.

Quando terminei, pareceu natural que estudássemos um dos arcabouços alvo do OpenARC. Seguindo com a ideia de explorar ferramentas abertas, passei a ler sobre o OpenCL. Mais uma vez me deparei com extenso material para leitura. Em especial, a especificação e a documentação oficiais do OpenCL são bastante descritivas, claras e possuem muitos exemplos. Apesar de um arcabouço bastante verboso, menos intuitivo e de aprendizado mais lento, uma vez que li sobre os mo-

delos de memória, execução, plataforma e programação do OpenCL, todo o estilo de programação pareceu mais intuitivo. E nesse sentido, [Gro23b] discute extensivamente esses modelos que definem o OpenCL. Apesar de intuitivo, o estilo de programação com o OpenCL é bastante chato e tem uma parte bastante repetitiva e muito parecida entre os programas escritos nele; nos momentos em que escrevi códigos com OpenCL, passei muito mais tempo escrevendo detalhes para lidar com o arcabouço e seu estilo de programação do que de fato me preocupei com os *kérneis* que estava escrevendo. Por esse motivo, me parece que utilizar essa plataforma só realmente vale a pena quando precisamos de otimização muito específica ou para manipulação de imagens, por exemplo. Parece melhor escrever um programa em OpenACC e, se necessário, transpilá-lo para OpenCL.

Depois de ter passado pelo OpenCL, voltei a estudar o OpenARC, agora entendendo melhor seu propósito. A dificuldade continuou, visto que o pouco material disponível para seu estudo se resume a artigos, que não têm o propósito de explicar sua implementação, e os comentários disponíveis no repositório dele. A solução que eu o professor Alfredo encontramos para compreender melhor seu funcionamento foi executá-lo em modo *debug* com um programa simples, de adição de vetores. A maneira de utilizá-lo e o que fazem cada uma das opções que podemos passar a ele também tiveram que ser deduzidas da leitura direta do código e de como cada uma das opções modifica a execução do compilador. Aliás, uma outra crítica quanto ao compilador diz respeito a seu acesso; os autores chamam-no de aberto, e no entanto, para podermos acessar seu repositório, devemos mandar uma mensagem a um de seus autores pedindo uma conta. Um projeto como esse, com um único autor de *commits*, pouca documentação e limitação ao acesso do repositório precisa de muito mais divulgação, documentação e autores para seguir crescendo.

Com tudo isso, esse trabalho foi bastante útil para mim. Tive de manter organizado e persistir pelas dificuldades. Além disso, em tempos de expansão da inteligência artificial, sinto que o conhecimento sobre programação paralela e heterogênea que o professor Alfredo me passou e os conteúdos que estudei para esse projeto serão importantíssimos para a minha carreira.

Referências

[Owe+07] John Owens et al. "A Survey of General-Purpose Computation on Graphics Hardware". Em: *Computer Graphics Forum* 26 (mar. de 2007), pp. 80–113. DOI: 10.1111/j.1467-8659.2007.01012.x.

[Dav+09] Chirag Dave et al. "Cetus: A Source-to-Source Compiler Infrastructure for Multicores". Em: *Computer* 42.12 (2009), pp. 36–42. DOI: 10.1109/MC.2009.385.

[HKG11] R. D. Hornung, J. A. Keasler e M. B. Gokhale. "Hydrodynamics challenge problem". Em: LLNL-TR-490254 (jun. de 2011). DOI: 10.2172/1117905. URL: https://www.osti.gov/biblio/1117905.

[Sca11] Matthew Scarpino. *OpenCL in action*. Manning Publications, nov. de 2011.

[Vet+11] Jeffrey S. Vetter et al. "Keeneland: Bringing Heterogeneous GPU Computing to the Computational Science Community". Em: *Computing in Science Engineering* 13.5 (2011), pp. 90–95. DOI: 10.1109/MCSE.2011.83.

[LV12] Seyong Lee e Jeffrey S. Vetter. "Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing". Em: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012. ISBN: 9781467308045.

[BB13] Ravishekhar Banger e Koushik Bhattacharyya. *OpenCL programming by example*. Birmingham, England: Packt Publishing, dez. de 2013.

[LV14] Seyong Lee e Jeffrey S. Vetter. "OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing". Em: *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014,

pp. 115–120. ISBN: 9781450327497. DOI: 10.1145/2600212.2600704. URL: https://doi.org/10.1145/2600212.2600704.

[LKV16] Seyong Lee, Jungwon Kim e Jeffrey S. Vetter. "OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing". Em: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016, pp. 544–554. DOI: 10.1109/IPDPS.2016.28.

[CJ17] Sunita Chandrasekaran e Guido Juckeland. *OpenACC for Programmers*. en. Boston, MA: Addison-Wesley Educational, set. de 2017.

[Far17] Rob Farber. *Parallel Programming with OpenACC*. Ed. por Rob Farber. Elsevier, 2017.

[Lee+21] Seyong Lee et al. *OpenARC: Open Accelerator Research Compiler*. Acessado em 2023-23-12. Fev. de 2021. URL: https://csmd.ornl.gov/project/openarc-open-accelerator-research-compiler.

[Cor22] NVIDIA Corporation. *CUDA Toolkit Installation Guide*. Acessado em 2023-11-15. Ago. de 2022. URL: https://docs.nvidia.com/cuda/archive/11.4.0/.

[22] *The OpenACC Application Programming Interface*. Versão 3.3. Nov. de 2022. URL: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf.

[Cor23] NVIDIA Corporation. *NVIDIA HPC SDK Installation Guide*. Acessado em 2023-11-15. Out. de 2023. URL: https://docs.nvidia.com/hpc-sdk/archive/23.9/compilers/openacc-gs/index.html.

[Gro23a] Khronos® OpenCL Working Group. *The OpenCL™ C Specification*. Acessado em 2023-11-12. Abr. de 2023. URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_C.pdf.

[Gro23b] Khronos® OpenCL Working Group. *The OpenCL™ Specification*. Acessado em 2023-11-12. Abr. de 2023. URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

[Gui] Adrien Joel Guillon. *AJ's Blog*. Acessado em 2023-11-12. URL: https://ajguillon.com/.

[Sch+] Joseph Schuchart et al. *Repositório do Lulesh*. Acessado em 2023-11--15. URL: https://github.com/LLNL/LULESH.