Modelagem “Fletcher” acelerada por diretivas OpenACC

Pedro Pais Lopes

Versão 0 de 5 de março de 2018

1 Introdução

Foi incluído no programa “Fletcher”, desenvolvido por Jairo Panetta, diretivas de compilação sob o padrão “OpenACC” para acelerar trechos do código e poderem ser executados primeiramente em GPUs, mas também em arquitetura multi e manycore. Este documento visa explicar o que foi feito, como obter, compilar e executar o programa e resultados dos testes preliminares de desempenho.

2 Breve explicação sobre o padrão OpenACC

O OpenACC é um padrão “*de facto*” desenvolvido por um comitê composto por empresas, universidades e centros de pesquisa com o objetivo de simplificar a programação paralela em ambientes com arquitetura computacional heterogênea, como por exemplo o conjunto CPU/GPU. É muito similar ao estilo de programação do padrão OpenMP, e assim como este o OpenACC é composto por pragmas (em C) ou comentários (em Fortran) que indicam ao compilador que o trecho de código referenciado deverá ser tratado de forma distinta do resto do programa. Este trecho deverá ser acelerado e executado no dispositivo de “destino” (do termo em inglês “*target*”), especificado na compilação.
O compilador trata o trecho referenciado assumindo algumas premissas:

- Possivelmente dados utilizados pelo trecho podem não estar disponíveis na memória do *target*, sendo então necessária movimentação (cópia) de dados entre memórias do ambiente computacional heterogêneo;
- Pode não existir coerência de memória entre os ambientes;
- Podem existir inúmeras memórias disponíveis no *target* (cache, memória de somente leitura, compartilhada, global etc);
- Não há coerência entre *threads* no *target*;
- Grande parte do código é executado no *host* (ambiente onde são disparadas as regiões aceleradas). Trechos acelerados são *offloaded*, ou seja, migrados para o *target*;
- Trechos de código acelerados criam “regiões computacionalmente intensivas”, e que mesmo na execução destas regiões pelo *target* o *host* deverá trabalhar para orquestrar a execução, mover ou copiar dados, realizar sincronismos e mesmo realizar outras operações independentes das enviadas para o target;
- O *target* pode possuir vários graus de paralelismo;
- A execução dos trechos acelerados pode ser feita de forma assíncrona, com retorno imediato após o envio das regiões aceleradas.

Ao tratar o trecho a ser acelerado de forma agnóstica (ou seja, baseada nas premissas listadas anteriormente) o compilador poderá utilizar do seu arcabouço de otimizações para melhor acelerar o código para as arquiteturas definidas na compilação. Ele também poderá realizar diversas operações de forma implícita, como por exemplo movimentar dados de e para o *target* conforme a necessidade e criar graus de paralelismo de acordo

com a complexidade (e aninhamento de laços) presente no trecho a ser acelerado, sem que o programador explicitamente indique ao compilador.
O padrão OpenACC mais novo é a versão 2.6 de novembro de 2017.
Os compiladores que possuem suporte ao OpenACC são:

- PGI, compilador da empresa "Portland Group Inc", adquirida em 2013 pela NVIDIA. É o compilador disponível gratuitamente para uso em arquitetura x86 e PowerPC com melhor suporte ao padrão OpenACC. Gera código para os "target" da NVIDIA;
- GCC, suíte de compiladores desenvolvida sob licença GPL, possui suporte ao padrão OpenACC 2.5 e que está em ativo desenvolvimento. Para que o OpenACC esteja disponível para uso é necessário a compilação da suíte GCC com chaves bastante específicas. Gera código para *targets* da NVIDIA;
- Compilador Cray, presente somente em ambientes computacionais da mesma marca. Gera código para *target* da NVIDIA e possivelmente para aceleradores INTEL de geração passada;
- Compiladores de pesquisa, tais como OpenUH, OpenARK, IPMACC, RoseACC, accULL e Omni, todos com suporte parcial ao padrão OpenACC de versões anteriores.

A inclusão de OpenACC na modelagem "Fletcher" foi feita com o compilador PGI, versão não comercial mais atualiza disponível (17.10), tendo como *host* um computador arquitetura x86 e *target* um acelerador arquitetura NVIDIA com "Compute Capability" ajustado de acordo com a placa utilizada.

3 Implementação

(neste texto, por motivos meramente estéticos, se referencia o "acelerador" como *target*, GPU ou placa. O padrão OpenACC não exige uma arquitetura específica, mas no propósito desta primeira implementação o acelerador é um equipamento da NVIDIA contendo arquitetura de "placa de vídeo" com memória RAM externa ao conjunto CPU-memória)
A aceleração do código via OpenACC foi incluído no Fletcher nos locais onde havia uso do OpenMP para paralelização. São duas funções:

- Dentro da "Propagate" (timestep.c), onde se encontra o ninho de três laços que percorre as dimensões do domínio e aplica o equacionamento principal;
- Dentro da "AbsorbingBoundary" (boundary.c).

Abaixo são colocados detalhes da implementação.

3.1 OpenACC dentro da "Propagate"

A função "Propagate" contém o trecho mais computacionalmente demandante do código, que é um ninho de 3 laços que percorre o domínio tridimensional aplicando o estêncil e calculando as derivadas.
Antes do ninho de laço existe uma diretiva "parallel" do OpenMP. Imediatamente antes do laço mais externo (laço em Z) existe uma diretiva "for". Para facilitar a implementação ambas as diretivas foram "fundidas", gerando um "parallel for" imediatamente antes do laço mais externo.
A inclusão de OpenACC ocorreu também no mesmo ponto, com a diretiva "kernels". Esta diretiva indica ao compilador que devem ser gerados um ou mais *kernels* para tratar

do paralelismo do trecho indicado. O compilador tem maior grau de liberdade para aceleração.
Devido ao trecho chamar duas funções – a “Der2” e a “DerCross” – foi necessário sinalizar, ao compilador, que estas funções devem ser executadas no acelerador (*device*).
Observou-se também que o ninho de laços reusa trechos de memória ao final (arrays “pp” e “qp”). Isso não é um limitante, mas o compilador identifica como um potencial problema de dependência entre iterações do laço. Como é sabido, pela leitura do algoritmo, que as iterações são independentes, incluímos pragmas antes dos laços (na linha anterior) indicando ao compilador que o laço é independente.
Com estes pragmas o compilador gerou código acelerado contendo o ninho de laços, mas na execução do programa ocorreu um erro fatal relativo a movimentação de dados. Este problema foi solucionado explicitando, com a opção “copy(...)” do pragma “kernels”, os 19 arrays necessários para a execução do trecho.

3.2 OpenACC dentro da “AbsorbingBoundary”

A AbsorbingBoundary é uma função chamada somente quando o programa é compilado com opção de borda “absorvitiva”, estando desabilitada na versão atual (V0).
Mesmo não sendo utilizada foi incluído pragma OpenACC no laço, que percorre SX*SY*SZ elementos. Da mesma forma que no “Propagate” foi incluída opção para indicar que o laço é seguro para aceleração.

4 Transferência mais eficiente de dados entre host e acelerador

O padrão OpenACC explicita que a memória acessível pelo *target* e pelo *host* podem ser distintas. Quaisquer movimentações de dados que sejam necessárias para execução correta dos trechos acelerados devem ser feitas (não há coerência entre as memórias), mas o compilador pode realizar estas operações de movimentação implicitamente. No caso deste trabalho o acelerador não mapeia diretamente a memória da CPU, portanto é necessário movimentar dados entre as duas memórias.
Existem algumas formas de se realizar essa movimentação. A mais comum é através do próprio compilador, que identifica a memória necessária para o trecho e inclui a movimentação[1]. Isso ocorre a cada execução do trecho acelerado, o que não é interessante pois dados inalterados entre chamadas do ninho de laços são mesmo assim copiados.
Uma forma mais explícita é o programador indicar o que deve ser copiado, podendo também indicar, na execução do trecho acelerado, quais dados já estão presentes no acelerador.
Existe uma terceira forma que deixa a cargo do *driver* do acelerador para que seja feita movimentação de dados sob demanda, utilizando tecnologia como a “*Unified Memory*” da NVIDIA.
As segundas e terceiras formas serão explicadas a seguir.

[1] No caso do trecho da “Propagate” o compilador não foi suficientemente eficiente para identificar toda a movimentação necessária. Testes preliminares mostraram que, em trechos mais simples, o compilador sim consegue efetuar todas as movimentações implicitamente.

4.1 Criação de trecho de dados no acelerador

Antes da chamada da “Propagate”, anteriormente ao laço que executa os passos de tempo (timesteps) dentro do main, foi incluída diretiva de compilação para que ocorra a cópia para o acelerador dos dados que serão usados posteriormente. São 19 arrays com dimensão SX*SY*SZ. Neste momento não ocorre execução de computação no acelerador, mas ocorre a cópia de dados da memória do *host* para o acelerador.
O *host* executa o laço nos passos de tempo, e a cada iteração executa a “Propagate”. Nesta função é encontrada a região acelerada anteriormente descrita. Foi incluída no pragma “kernels” a opção “present” para identificar que os 19 arrays já estão presentes na memória do acelerador, não sendo necessário copiar os arrays a cada invocação. O trecho acelerado é executado, não ocorrendo qualquer cópia de dados após o fim da execução. A “Propagate” retorna ao main.
A função seguinte (“TimeForward”) é a que realiza a troca de ponteiros do campo atual – pc e qc – com o campo anterior – pp e qp. Esta função é executada no *host*. Os ponteiros trocados no *host* são assim mantidos até a próxima execução desta função na seguinte iteração do laço. Ao serem utilizados pelas regiões aceleradas devem ser “dereferenciados” de acordo com seu estágio atual a cada execução, pois são ponteiros passados para as regiões aceleradas, e não ponteiros declarados internamente nestas regiões. Assim sendo a troca de ponteiros resulta em troca de referenciamento de regiões de memória tanto no *host* quanto no *target*.
A função seguinte (“InsertSource”) é a responsável por inserir a fonte dentro do domínio, aproximadamente no centro do mesmo. Esta função altera dados em campos pc e qc e deverá, portanto, ou ser executada no acelerador ou ocorrer atualização com dados do acelerador na memória do *host*, inserção da fonte e retorno dos dados ao acelerador para a próxima iteração. Optou-se por executar o trecho no acelerador com inserção da mesma diretiva “kernels” nesta função[2]. Da mesma forma a opção “present” foi utilizada. Com esta opção houve necessidade de duplicar a função e inserir o OpenACC somente na cópia que é executada dentro do trecho com os dados no acelerador com a seguinte justificativa: os arrays “p” e “q” são argumentos desta função; esta função também é executada fora da região com a definição de dados no *target* (que circunda o laço nos timestep); opção “present” referencia, na execução externa, dados inexistentes no *target* porque só estarão presentes dentro do trecho com definição de dados[3].
A próxima função é a “AbsorbingBoundary”. Foi feita mesma implementação: pragma “kernels” com opção “present".
As duas últimas são funções de escrita em disco ou de *debug*. São executadas no *host*, obrigatoriamente. Para que sua memória esteja atualizada utilizou-se diretiva “update”, referenciando o array necessário (pc) para tal atualização.

4.2 Uso da tecnologia “CUDA *Unified Memory*”

Da mesma forma que a maneira “automática”, onde não se referencia qualquer movimentação de dados e o compilador adivinha qual movimentação é necessária,

[2] Observou-se que o compilador exige um laço, o que foi solucionado com um laço de somente uma iteração.
[3] Está sendo feita uma investigação do padrão e de sua implementação no compilador para verificar se existe uma condicional, em tempo de execução, para referenciar dados presentes no *target*.

pode-se utilizar a tecnologia UM (*Unified Memory*). Basta incluir uma opção de compilação. Esta opção, no entanto, não foi testada.

5 Disponibilidade do código, compilação e execução

O código está disponível em /home/pedro.lopes/fletcher-openacc-v0.
A compilação exige o uso do compilador PGI. Na blaise sugiro fortemente utilizar os módulos do EasyBuild:

- source /home/pedro.lopes/eb/ebrc
- module load PGI (ou “ml PGI”, são equivalents)

Com o PGI carregado em seu ambiente prossiga com a compilação no diretório “src”. Altere (descomentando) a linha no início do Makefile conforme desejado. A compilação é feita de forma usual (make).
A execução é feita no diretório “run” com o comando “EXEC.sh”. Para opções de execução verificar na documentação do Fletcher, feita pelo Jairo.

6 Resultado da aceleração implementada

Testes preliminares foram feitos utilizando a seguinte linha de comando:

```
time ./ModelagemFletcher.exe "VTI" 241 241 241 16 12.5 12.5 12.5 0.0010 2.0
```

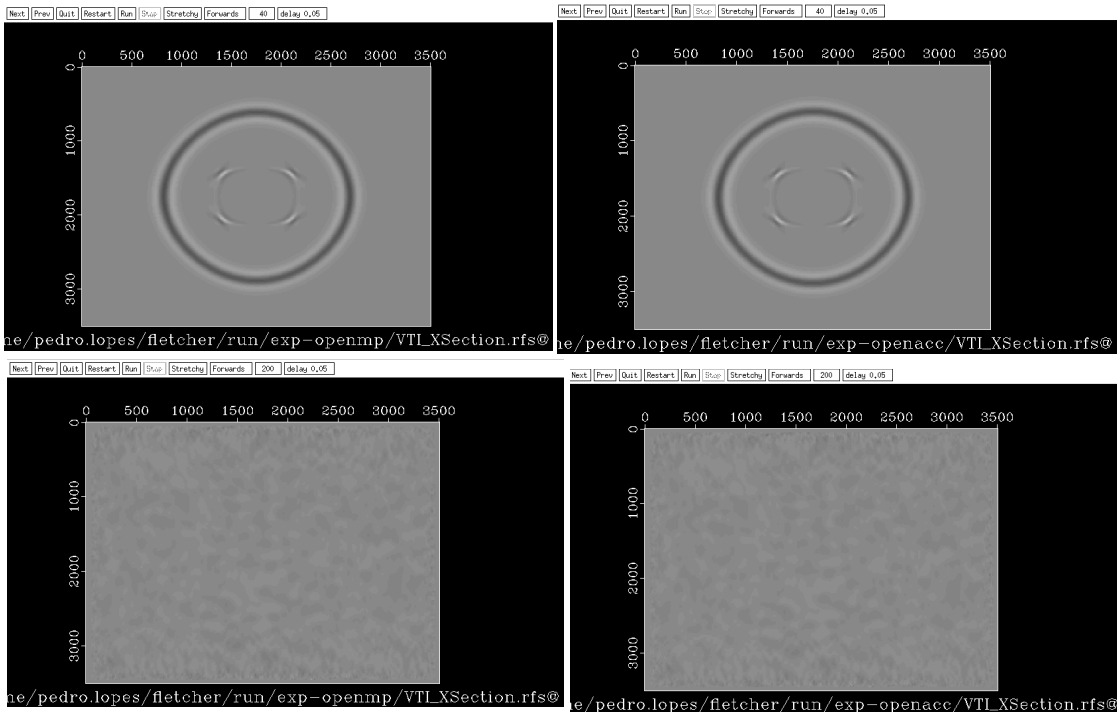
A aceleração do código provocou redução no tempo de execução total do programa. Mas esta redução deve ser assegurada pela corretude do resultado, que não deve apresentar diferenças entre a referência – programa sem diretivas do OpenACC – e o código acelerado. A seguir são colocadas comparações entre resultados e o detalhamento da redução do tempo de execução.

6.1 Análise de corretude

As duas execuções – com OpenMP e com OpenACC – foram confrontadas. Uma análise objetiva simples (comparação dos arquivos de saída via comando “diff”) resultou inválida, o que é esperado pois o *target* e *host* são de arquiteturas diferentes e que podem realizar operações de ponto flutuante com arredondamentos diferentes. Optou-se por uma comparação visual.
Os resultados foram cortados no eixo X no meio do campo (posição 140). O corte, em cada *timestep*, foi visualizado com programas da suíte Madagascar:

```
sfgrey < arquivo.rfs gainpanel=all wantframenum=no polarity=y | sfpen
```

Foram extraídos dois gráficos de cada execução: um no instante 40 e outro no final (instante 200). Os gráficos estão colocados a seguir (do lado esquerdo estão os resultados com OpenMP, do lado direito os resultados com OpenACC):



De forma visual os campos apresentados são os mesmos.

6.2 Análise da redução do tempo de execução

O tempo de cada execução foi obtido em dois relógios: um, externo, que mede o tempo total de execução do programa desde a invocação na linha de comando até o seu retorno, medido pelo comando “time”. O outro relógio foi inserido dentro da rotina Propagate, que mede somente o tempo de execução do ninho de laços acelerado (pelo OpenACC) ou paralelizado (pelo OpenMP).

As execuções se deram em dois ambientes. O primeiro ambiente é externo à UFRGS (denominado “externo”) que possui dois soquetes XEON com 20 núcleos reais de 2,2Ghz e um acelerador NVIDIA Tesla K80 (contém duas GPUs similares à K40; os testes utilizaram somente uma dessas GPUs). Outro ambiente é o servidor “blaise”, com dois soquetes XEON com 44 núcleos reais de 2,2Ghz e quatro aceleradores NVIDIA Tesla P100. Os testes utilizaram apenas um P100.

A tabela a seguir apresenta os tempos encontrados, em segundos. As execuções OpenMP utilizaram todos os núcleos físicos disponíveis nas máquinas.

Ambiente	Tempo total OpenMP	Tempo total OpenACC	Speedup	Tempo timestep OpenMP	Tempo timestep OpenACC	Speedup
externo	1100	308	3,57	1066	265	4,02
blaise	502	182	2,76	483	100	4,83

Observa-se que em ambos os relógios – total e timestep – ocorre redução significativa do tempo de execução, refletida nos valores de Speedup. Observa-se também que o mesmo código é 2,65 vezes mais rápido executando-se na P100 que na K80, explicado em grande parte pelo maior poder computacional da P100. Já os tempos de execução OpenMP foram inversamente proporcionais ao número de núcleos físicos nas duas máquinas. Os 44 núcleos x86 da blaise resultaram em tempo ligeiramente menor que a metade do tempo obtido nos 20 núcleos x86 da máquina externa.

7 Comentários

Esta primeira implementação de OpenACC na modelagem “Fletcher” resultou em um código que produz resultados visualmente iguais ao da referência. Sendo assim o objetivo desta implementação – o da corretude – foi alcançado.

Também é uma implementação totalmente portável, pois não foi alterada a estrutura do programa original nem foram reduzidas suas funcionalidades, pois a implementação OpenMP continua funcional. Para uso do OpenACC, por um compilador compatível, basta utilizar a chave de compilação adequada.

O tempo de execução do código com a primeira otimização (movimentação mais eficiente de dados) é aproximadamente 3 vezes mais rápido no tempo total, ou 5 vezes no tempo do trecho computacionalmente intenso, comparado ao tempo com OpenMP. Isso também mostra que o OpenACC conseguiu extrair desempenho do acelerador empregado (acelerador NVIDIA) com mínima alteração do código e com manutenção total da sua portabilidade, já que em ambas as execuções – com OpenMP e com OpenACC – houve somente a inclusão de uma chave de compilação, com o mesmo código fonte.

Não foi feita busca por outros *hotspots* no código (talvez apareçam com outras configurações de execução ou compilação). Possivelmente existam trechos computacionalmente demandantes no programa que possam ser acelerados *no target*. Da mesma forma podem existir operações com tempo de execução significativo, tais como cópias, início e término de trechos de dados onde, com análise aprofundada de uma ferramenta da *profiling*, tenham seu tempo de execução reduzido ou eliminado.