Descrição do programa Fletcher-2.0

Jairo Panetta / Paulo Souza / Pedro Lopes / Rodrigo Machado

Versão 1 de 17 de julho de 2018

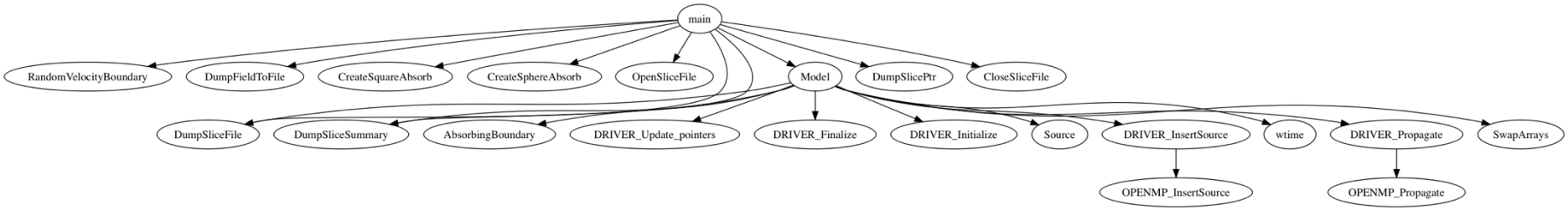
1 Introdução

O programa de Modelagem Fletcher, feito pelo Jairo originalmente em OpenMP, foi otimizado e acelerado em mais três maneiras utilizando OpenACC, OpenMPtarget e CUDA. Estas quatro implementações foram reunidas em uma única árvore, com reuso máximo de funções comuns e trechos computacionalmente demandantes, mas permitindo a diferenciação no que tange a compilação.
Este documento visa introduzir a nova versão, layout de diretórios, padronização de nome de arquivos, fluxo de execução, mecanismo de controle de compilação e comparação de desempenho.

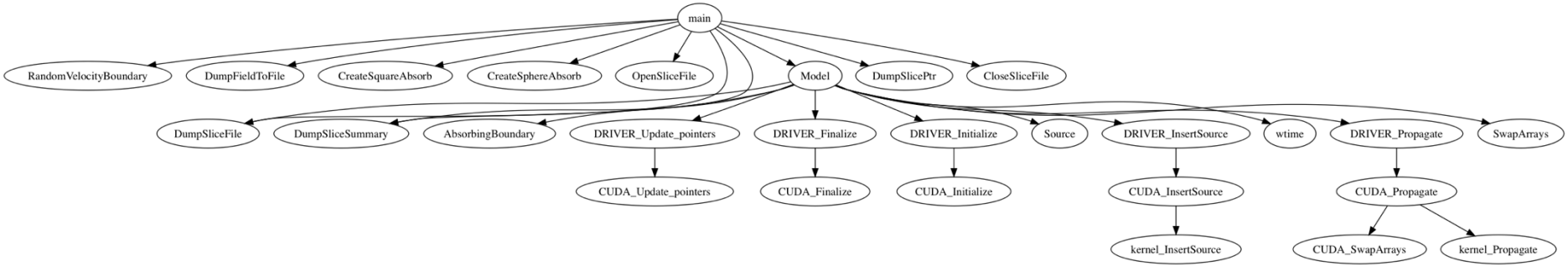
2 O Fletcher e suas quatro acelerações

O programa Fletcher possui quatro acelerações: OpenMP, OpenMPtarget, OpenACC e CUDA. As quatro estão dispostas em uma árvore de diretórios que maximiza o reuso de código comum, evitando multiplicar por quatro o esforço de modificação do código.
Foram geradas, através dos programas “cscope”, “tceetree” e “dot” (GraphWiz) as seguintes árvores de chamada do Fletcher, uma para cada aceleração.

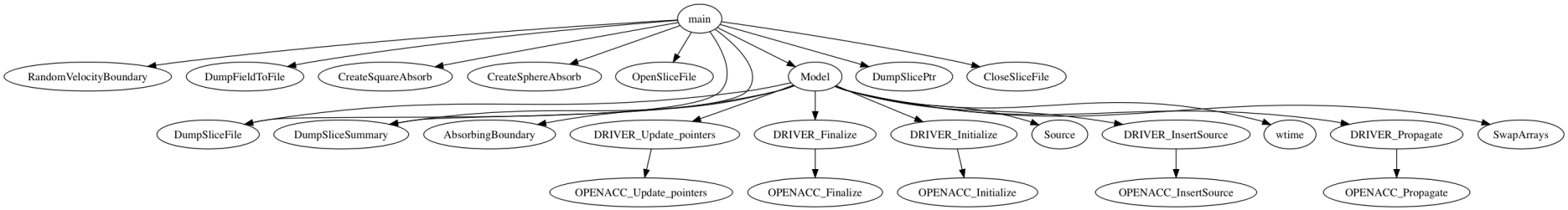
(árvore de chamadas para aceleração OpenMP)



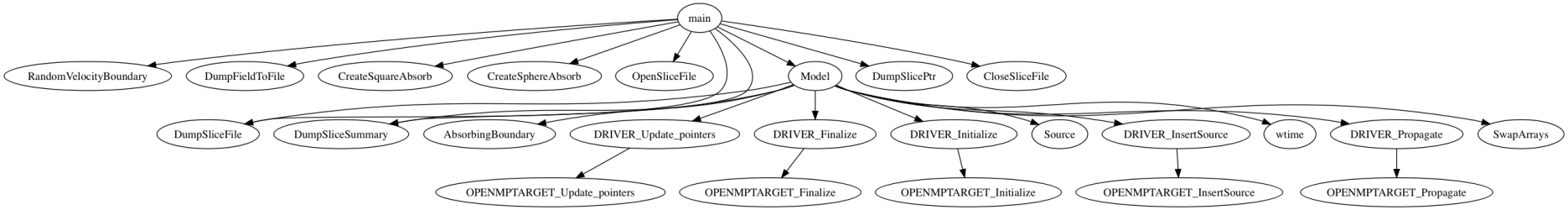
(árvore de chamadas para aceleração CUDA)



(árvore de chamadas para aceleração OpenACC)



(árvore de chamadas para aceleração OpenMPtarget)



As quatro acelerações iniciam a execução no main. Este realiza algumas atividades de *setup* – cria borda absortiva, abre arquivo de saída, escreve estado inicial em disco entre outras atividades – e chama a função model, responsável pela modelagem.
A model inicializa a aceleração (DRIVER_Initialize). Depois procede para o laço principal nos timesteps.
Em cada iteração do laço ocorre a inserção da fonte (DRIVER_InsertSource), a propagação (DRIVER_Propagate), troca os ponteiros (SwapArrays) e, dependendo das opções de compilação e passo de tempo, faz borda absortiva e/ou escreve no arquivo de saída e na tela. Ao final do laço imprime o desempenho da aceleração (em milhões de Samples por segundo) e finaliza com a DRIVER_Finalize.
As rotinas com prefixo DRIVER_ são implementadas por todas as acelerações, e como somente uma aceleração é compilada por vez essas funções fazem a conexão do código principal com o código acelerado.
A estrutura de arquivos está disposta de maneira similar à ideia dos drivers. Uma aceleração, que no desenvolvimento é neste documento é referenciada como "backend", e está descrita a seguir.

3 Layout de um backend

Cada backend possui um diretório próprio, com arquivo Makefile, arquivo com chaves de compilação próprias para aceleração e pelo menos quatro arquivos-fonte.
A conexão do código principal no backend se dá pelas funções implementadas no arquivo {backend}_driver.c ({backend} é o nome do backend, por exemplo OpenMP), a saber:

- DRIVER_Initialize;
- DRIVER_Finalize;
- DRIVER_Propagate;
- DRIVER_Update_Pointers;
- DRIVER_InsertSource.

Estas funções, se necessário, devem unicamente chamar a função com nome similar, porém com sufixo {backend}_. Não devem fazer nenhuma operação a mais. Estas funções estão implementadas nos arquivos descritos abaixo:

- {backend_stuff}.c/cu: possui as seguintes funções opcionais (ou seja, somente se o backend precisar devido a peculiaridade própria):
    - {backend}_Initialize: responsável por inicializar o backend (exemplos são alocação de memória do acelerador em CUDA ou pragma OpenACC para copiar dados para o acelerador);
    - {backend}_Finalize: responsável por finalizar o backend (exemplo liberação da memória do acelerador);
    - {backend}_Update_pointers: responsável por atualizar os ponteiros na memória central que serão lidos para a escrever no arquivo de saída.
- {backend}_propagate.c/cu: todos os backends possuem. Implementa a {backend}_Propagate, responsável pela propagação. Se necessário implementa outras funções ou kernels;
- {backend}_insertsource.c/cu: todos os backend possuem. Implementa a {backend}_InsertSource, responsável pela inserção da fonte no domínio.

Junto com os códigos-fonte cada backend existem mais dois arquivos importantes:

- Makefile para compilação dos códigos do backend;
- flags.mk: chaves de compilação de todo o programa, inclusive do código-fonte na raiz.

A seguir está descrita a configuração e forma de compilação.

4 Compilação do fletcher com os backends

De forma a respeitar a estrutura de backends, com peculiaridades e configurações próprias, cada um possui seu próprio Makefile e um arquivo com configurações de compilação denominado flags.mk. Este arquivo configura a compilação do programa inteiro.

Na raiz o usuário compila o programa com o comando "make". O controle do backend a ser utilizado é feito com o argumento "backend=BACKEND". Assim sendo, "make backend=CUDA" compila o Fletcher com o backend CUDA. Caso o argumento seja otimido então o backend OpenMP será utilizado.

O make começa compilando os arquivos da raiz. Utiliza para tanto duas configurações:

- config.mk: arquivo da raiz contendo configurações gerais e flags globais;
- backend/flags.mk: arquivo do backend contendo configurações específicas do backend;

O flags.mk é utilizado também para compilação dos códigos da raiz – main, model, boundary etc – porque é importante, para o bom funcionamento do programa, que as mesmas otimizações e principalmente o mesmo compilador seja utilizado para todos os códigos.

A compilação segue para o diretório do backend, onde é controlada pelo Makefile do mesmo.

Ao final ocorre a linkedição no diretório raiz, gerando o executável "ModelagemFletcher.exe". O compilador utilizado para toda a compilação – raiz e backend – é o mesmo para a linkedição.

No caso de código CUDA, compilado com o nvcc, este é utilizado somente para os arquivos com extensão ".cu".

5 Peculiaridades do código

O código possui algumas peculiaridades que merecem ser descritas neste documento. A primeira delas é a métrica de desempenho. A segunda é o reuso de código. A terceira é a troca das funções de derivadas por macros.

5.1 Métrica de desempenho samples/s

Esta versão inclui uma métrica de desempenho conhecida como samples/s ("samples por segundo"). Ela métrica corresponde a quantidade de trabalho realizada dividida pelo tempo gasto. Um trabalho, ou "um sample", é a aplicação do *stencil*, derivadas e qualquer outra operação em um ponto de grade.

Assim sendo a quantidade total de trabalho (quantidade de samples) do Fletcher é:

$$samples = NX * NY * NZ * nTimesteps$$

O samples/s é a divisão do número de samples pelo tempo gasto para esta operação.

Por mera comodidade se costuma mensurar esta métrica em “megaSamples” ou “gigaSamples” por segundo.

5.2 Reuso de código

O código unificado do Fletcher reusa código entre os backends. Mas este reuso se dá somente no código interno ao propagate.
Todos os backends – OpenMP, OpenACC, OpenMPtarget e CUDA – utilizam o mesmo algoritmo para propagar uma onda. O seu código está implementado no arquivo sample.h e derivatives.h na raiz.
Como descrito em documento anterior a propagação se dá dentro de um ninho de laços que percorre as três dimensões do domínio. No caso do backend CUDA, devido ao paralelismo de blocos de threads e threads por blocos, a propagação ocorre dentro de um laço na direção Z. Em todos os backends o algoritmo é o mesmo.
A solução para reuso da mesma implementação do algoritmo foi de incluir um arquivo dentro do laço, conforme código a seguir (do backend OpenMP, bastante representativo):

```
void OPENMP_Propagate(int sx, int sy, int sz, int bord,
        float dx, float dy, float dz, float dt, int it,
        float * restrict ch1dxx, float * restrict ch1dyy, float * restrict ch1dzz,
        float * restrict ch1dxy, float * restrict ch1dyz, float * restrict ch1dxz,
        float * restrict v2px, float * restrict v2pz, float * restrict v2sz, float * restrict v2pn,
        float * restrict pp, float * restrict pc, float * restrict qp, float * restrict qc) {


#define SAMPLE_PRE_LOOP
#include "../sample.h"
#undef SAMPLE_PRE_LOOP


#pragma omp parallel
 { // start omp

   // solve both equations in all internal grid points,
   // including absortion zone


#pragma omp for
   for (int iz=bord; iz<sz-bord; iz++) {
     for (int iy=bord; iy<sy-bord; iy++) {
       for (int ix=bord; ix<sx-bord; ix++) {


#define SAMPLE_LOOP
#include "../sample.h"
#undef SAMPLE_LOOP


      }
    }
```

```
    }
  } // end omp
}
```

Notar as inclusões da "../sample.h" fora e dentro do ninho de laços. A chave de pré-processamento que circunda a inclusão controla de qual parte do arquivo sample.h o código é incluído.

5.3 Troca de funções por macros

O algoritmo de propagação realiza derivada de segunda ordem e cruzada. A implementação de tais derivadas se dá através das funções Der2 e DerCross, respectivamente.
Nesta versão ambas foram trocadas de função para macros, implementadas no arquivo derivatives.h. Este arquivo é incluído no início do backend_Propagate.c/cu.

6 Baseline de desempenho

As quatro acelerações/backends foram executadas com diversos tamanhos de grades cúbicas em dois equipamentos: o do projeto com GPU Nvidia Pascal P100 e 44 núcleos reais de CPU e outro, externo, com GPU Nvidia Kepler K80 e 20 núcleos reais de GPU. A métrica utilizada foi Msamples/s.
Foram feitas cinco compilações de três backends (o backend OpenMPtarget não foi executado devido a falta de compilador compatível com o Ubuntu 18.04):

- OpenMP CPU GCC: OpenMP em CPU com compilador GCC 6.4.0;
- OpenMP CPU PGI: OpenMP em CPU com compilador PGI 18.4;
- OpenACC CPU PGI: OpenACC em CPU com compilador PGI 18.4;
- OpenACC GPU PGI: OpenACC em acelerador GPU com compilador PGI 18.4;
- CUDA: CUDA em acelerador GPU com compilador NVCC 9.2.88 e GCC 6.4.0.

A tabela a seguir coloca o desempenho obtido no equipamento com a P100:

Tamanho (NX=NY=NZ)	OpenMP CPU GCC	OpenMP CPU PGI	OpenACC CPU PGI	OpenACC GPU PGI	CUDA
24	56	181	164	1007	343
88	30	283	358	1081	677
216	44	160	237	1011	842
344	57	152	230	934	812
472	46	143	191	768	797

Muito provavelmente devido a troca das derivadas de funções para macros o otimizador do compilador PGI melhorou o desempenho de forma substancial, principalmente porque o OpenMP CPU com PGI é mais de três vezes mais rápido que com GCC. Já o OpenACC é mais rápido que o OpenMP, talvez devido a maior liberdade do compilador com a diretiva kernels (utilizada no backend).
Ao se usar o acelerador o OpenACC possui maior desempenho que o CUDA, exceto na grade maior.

A execução do baseline no equipamento com a K80 gerou os resultados abaixo:

Tamanho (NX=NY=NZ)	OpenMP CPU GCC	OpenMP CPU PGI	OpenACC CPU PGI	OpenACC GPU PGI	CUDA
24	32	123	182	210	110
88	15	139	144	259	162
216	21	108	113	260	142
344	27	93	116	234	131
472	23	79	95	220	130

Assim como visto no outro equipamento o compilador PGI otimizou melhor o código (vide comparação de OpenMP com GCC e com PGI). OpenACC é ligeiramente melhor que o OpenMP com PGI. De forma consistente OpenACC na K80 é melhor que o CUDA, chegando a ser 80% melhor, e não apresenta queda de desempenho com tamanhos maiores de grade, como visto na Pascal.

7 Comentários finais

O desenvolvimento das otimizações do Fletcher segue em atividade. Suas otimizações muito provavelmente se darão somente dentro dos backends, no ninho e laços e no código executado por este ninho de laços. Assim sendo espera-se desenvolvimento dos arquivos {backend}_Propagate.c/cu, sample.h e derivatives.h.

O reuso do algoritmo da propagação permitiu uma base muito confiável de comparação entre os backends. Porém, dependendo da otimização poderá não ser interessante que todos os backends usem a mesma implementação. O código como está permite tal especificidade trocando as *flags* de compilação no arquivo flags.mk. Uma leitura atenta do sample.h e derivatives.h mostra quais *flags* são utilizadas.