Descrição Modelagem Fletcher 3.0

13 de março de 2019

Jairo Panetta, Pedro Pais Lopes, Rodrigo Lima Machado

1 Conteúdo

Descaradamente, seleciono trechos do Relatório Técnico II que contém alguma descrição das versões disseminadas no arquivo fletcher-3.0.tar.gz.

Espero ser útil.

2 Versão Unificada

Cada um dos diretórios presentes em fletcher-3.0.tar.gz implementa estrutura de software que reusa trechos de código de implementações C, OpenMP, OpenACC e CUDA da Modelagem Fletcher. Denominamos essa estrutura de software versão *unificada*, descrita a seguir.

Os arquivos fonte são distribuídos em uma árvore de diretórios. A raiz da árvore contém os arquivos comuns às quatro linguagens (denominados *front-ends*), escritos em C, e um conjunto de diretórios contendo os arquivos específicos de cada linguagem (denominados *back-ends*). Há três *back-ends* com as codificações em OpenMP, OpenACC e CUDA. A codificação em C é a obtida compilando a implementação em OpenMP com a respectiva chave de compilação desligada.

Arquivos *front-ends* implementam as funcionalidades comuns a todas as linguagens, como inicializar a computação, percorrer o laço que avança a propagação no tempo e finalizar a computação. Funções dos *back-ends* são invocadas onde necessário, por exemplo, para propagar o campo de onda um passo no tempo.

Arquivos *front-ends* são conectados aos arquivos *back-ends* invocando *drivers*. Um *driver* é uma função comum ao *front-end* e ao *back-end* com assinatura e semântica padronizadas. Cada *back-end* implementa os *drivers* na sua linguagem, gerando as semânticas desejadas. Bastam cinco *drivers* para gerar a versão unificada: *Initialize*, *Finalize*, *Propagate*, *UpdatePointers* e *InsertSource*.

Os *drivers Initialize*, *Finalize*, *UpdatePointers* e *InsertSource* implementam funcionalidades necessárias em uma arquitetura e inexistentes nas outras (por exemplo, alocar e transportar áreas de memória entre a CPU e a GPU), assim como funcionalidades que são implementadas de forma particular a cada linguagem (por exemplo, iniciar e terminar regiões paralelas).

O *driver Propagate* propaga o campo de ondas um passo no tempo em todos os pontos da grade. Os *back-ends* de *Propagate* em OpenMP e OpenACC iniciam-se com o ninho de laços que percorre a grade e as diretivas que implementam paralelismo no ninho. A codificação em CUDA invoca um *kernel* que atribui um ponto de grade a uma *thread*, gerando implicitamente o ninho de laços que percorre a grade. O interior do ninho de laços (e do *kernel* em CUDA) inclui o mesmo arquivo (denominado *sample.h*) que contém todos os comandos necessários para propagar a onda um passo no tempo em um ponto da grade. Dessa forma, qualquer alteração inserida em *sample.h* é automaticamente propagada para todas as implementações.

3 Versão Original

Para simplificar descrições posteriores, anexamos listagem do arquivo *sample.h* original, detalhada a seguir.

Listagem 1: Arquivo *sample.h* original

```
// p derivatives, H1(p) and H2(p)

const float pxx= Der2(pc, i, strideX, dxxinv);
const float pyy= Der2(pc, i, strideY, dyyinv);
const float pzz= Der2(pc, i, strideZ, dzzinv);
const float pxy= DerCross(pc, i, strideX, strideY, dxyinv);
const float pyz= DerCross(pc, i, strideY, strideZ, dyzinv);
const float pxz= DerCross(pc, i, strideX, strideZ, dxzinv);

const float cpxx=ch1dxx[i]*pxx;
const float cpyy=ch1dyy[i]*pyy;
const float cpzz=ch1dzz[i]*pzz;
const float cpxy=ch1dxy[i]*pxy;
const float cpxz=ch1dxz[i]*pxz;
const float cpyz=ch1dyz[i]*pyz;
const float h1p=cpxx+cpyy+cpzz+cpxy+cpxz+cpyz;
const float h2p=pxx+pyy+pzz-h1p;

// q derivatives, H1(q) and H2(q)

const float qxx= Der2(qc, i, strideX, dxxinv);
const float qyy= Der2(qc, i, strideY, dyyinv);
const float qzz= Der2(qc, i, strideZ, dzzinv);
const float qxy= DerCross(qc, i, strideX,  strideY, dxyinv);
const float qyz= DerCross(qc, i, strideY,  strideZ, dyzinv);
const float qxz= DerCross(qc, i, strideX,  strideZ, dxzinv);

const float cqxx=ch1dxx[i]*qxx;
const float cqyy=ch1dyy[i]*qyy;
const float cqzz=ch1dzz[i]*qzz;
const float cqxy=ch1dxy[i]*qxy;
const float cqxz=ch1dxz[i]*qxz;
const float cqyz=ch1dyz[i]*qyz;
const float h1q=cqxx+cqyy+cqzz+cqxy+cqxz+cqyz;
const float h2q=qxx+qyy+qzz-h1q;

// p-q derivatives, H1(p-q) and H2(p-q)

const float h1pmq=h1p-h1q;
const float h2pmq=h2p-h2q;

// rhs of p and q equations

const float rhsp=v2px[i]*h2p + v2pz[i]*h1q + v2sz[i]*h1pmq;
const float rhsq=v2pn[i]*h2p + v2pz[i]*h1q - v2sz[i]*h2pmq;

// new p and q

pp[i]=2.0*pc[i] - pp[i] + rhsp*dt*dt;
qp[i]=2.0*qc[i] - qp[i] + rhsq*dt*dt;
```

O trecho de código avança a propagação da onda um passo de tempo no ponto da grade definido pela variável `i`. A computação é estruturada em cinco trechos apresentados em ordem lexicográfica e que calculam:

1. Derivadas do campo *p* no instante atual (sobre o campo `pc`) e seus operadores (`h1p` e `h2p`);
2. Derivadas do campo *q* no instante atual (sobre o campo `qc`) e seus operadores (`h1q` e `h2q`);
3. Operadores do campo *p-q* no instante atual (`h1pmq` e `h2pmq`);
4. Lado direito das equações que avançam os campos *p* e *q* (`rhsp` e `rhsq`);

5. Avanço no tempo, calculando os campos no instante futuro (`pp` e `qp`).

Os campos tridimensionais são mapeados em arrays unidimensionais na ordem (x, y, z), isto é, x é a direção que varia mais rapidamente na memória. Cada segunda derivada é calculada invocando o macro `Der2`; a direção da derivada é determinada pelo salto em memória (`strideX`, `strideY`, `strideZ`). Cada derivada cruzada é calculada invocando o macro `DerCross` com direções determinadas da mesma forma. O cálculo dos operadores *H1* e *H2* de Fletcher utiliza coeficientes trigonométricos (`ch1dxx` ... `ch1dyz`) calculados uma única vez no *front-end*.

O macro `Der2` calcula a segunda derivada por diferença finita unidimensional de 9 pontos centrada em `i`. O macro `DerCross` calcula a derivada cruzada por diferença finita bidimensional de 64 pontos centrada em `i`.

Passamos a denominar versão *Original* à esta versão unificada.

3.1 Avaliação de Desempenho da Versão Original

O paralelismo da versão Original é imediato, pois a propagação em um ponto da grade é independente da propagação em qualquer outro ponto da grade a cada passo no tempo. Logo, o ninho de laços que percorre a grade, contido em *Propagate*, é totalmente paralelizável.

Confeccionamos avaliação de desempenho que visa determinar o custo da portabilidade entre CPU e GPU, contrastando a velocidade da codificação portátil nas duas arquiteturas (OpenACC) com a velocidade das codificações não portáteis (OpenMP na CPU e CUDA na GPU), específicas para cada arquitetura. A portabilidade de OpenACC é de tal ordem que uma única compilação gera código para as duas arquiteturas. Variável de ambiente define, durante a execução, a arquitetura a utilizar.

Executamos as codificações OpenMP, OpenACC e CUDA em 12 grades de tamanhos crescentes, escolhidas de forma que o tamanho da grade em qualquer direção seja múltiplo de 32. Utilizamos os compiladores PGI versão 18.10.0 e NVCC versão 10.0.130. Executamos no computador de pesquisa I do projeto, composto por duas pastilhas Intel Xeon E5-2699, quatro GPUs NVIDIA P100 e 256GB de memória central. Cada pastilha Xeon opera à 2,2GHz e possui 22 núcleos de processamento.

A Figura 1 apresenta a velocidade em MSamples/s em função do tamanho do problema para as codificações OpenMP, OpenACC executando na CPU, OpenACC executando na GPU e CUDA. A velocidade apresentada é a média de cinco execuções, com 100 passos no tempo cada. A velocidade é medida pela divisão do número de pontos de grade atualizados a cada passo no tempo pelo tempo de execução do laço que avança o tempo, ambos acumulados ao longo dos passos no tempo. As execuções na CPU utilizaram todos os 44 núcleos e as na GPU utilizaram uma única P100.

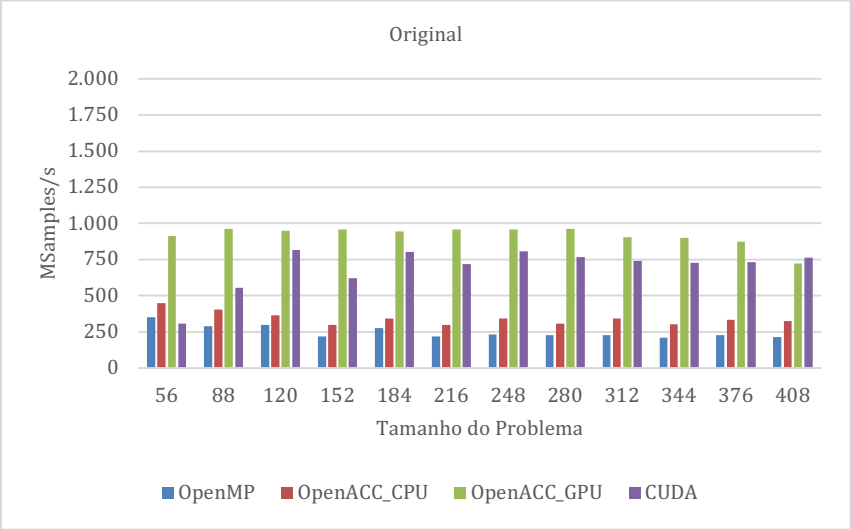

Figura 1: Velocidade da Versão Original

Surpreendentemente, a versão portátil é mais rápida do que as versões não portáteis. OpenACC é consistentemente mais rápido que OpenMP na CPU e que CUDA na GPU.

3.2 Direções de Otimização

Como buscamos avaliar o custo da portabilidade na eficiência, não é adequado escolher otimizações específicas para uma arquitetura. Empregamos otimizações independentes da arquitetura alvo. As otimizações visam reduzir a quantidade de contas e a quantidade de memória utilizada. Denominamos *genéricas* tais otimizações.

As otimizações genéricas foram implementadas em todas as linguagens, para contrastar o desempenho da codificação portátil (OpenACC) com o desempenho das não portáteis (OpenMP e CUDA).

Seguimos apresentando três versões com otimizações genéricas. A apresentação de cada versão inicia-se com sua descrição, seguida da avaliação de desempenho. Utilizamos a mesma avaliação de desempenho para todas as versões, permitindo contrastar a velocidade de execução entre quaisquer versões e codificações.

Todas as versões utilizam a estrutura de software unificada acima descrita.

4 Versão Der1Der1Orig

Constatamos que o cálculo das derivadas cruzadas demanda 76% das operações de ponto flutuante da versão original. Esta constatação foi obtida por PAPI na execução sequencial OpenMP de uma grade de 216 x 216 x 216 pontos internos. Constatação similar foi obtida com NVPROF na GPU na mesma grade.

Reduzir a quantidade de operações para calcular as derivadas cruzadas foi sugestão apresentada por Carlos Cunha (Petrobras/GEOF/TG), na primeira reunião de avaliação. A sugestão foi calcular a derivada cruzada em `xy` como primeira derivada em `y` da primeira derivada em `x`. Essa sugestão não reduz a quantidade de contas no cálculo da derivada cruzada em um ponto, mas reduz a quantidade de contas no cálculo da derivada cruzada em `y` consecutivos, por reutilizar sete derivadas em `x` previamente calculadas dentre as oito necessárias. Como a mesma redução ocorre nas demais derivadas cruzadas, aplicamos essa sugestão para o cálculo de todas as derivadas cruzadas.

Inicialmente, calculamos e armazenamos as derivadas primeiras em `x` e em `y` de todos os pontos da grade. Utilizamos a primeira derivada em `x` para calcular as derivadas cruzadas em `xy` e em `xz` e utilizamos a primeira derivada em `y` para calcular a derivada cruzada em `yz`.

Para implementar essa otimização modificamos o arquivo *sample.h* de duas formas. Primeiro, inserindo preâmbulo ao ninho de laços que calcula e armazena as derivadas primeiras, invocando o novo macro `Der1`, que calcula a primeira derivada utilizando diferenças finitas de oito pontos. Segundo, substituindo as invocações a `DerCross` por invocações a `Der1`, recebendo como argumento de entrada a primeira derivada calculada no preâmbulo. Esses dois trechos de *sample.h* são orlados por *#ifdef*, permitindo que cada trecho seja inserido no local adequado do código em todas as codificações.

4.1 Avaliação de Desempenho da Versão Der1Der1Orig

Executamos a avaliação de desempenho padronizada na versão Der1Der1Orig. A Figura 2 apresenta a velocidade de execução desta versão com formato e nomenclatura idênticos aos da Figura 1.

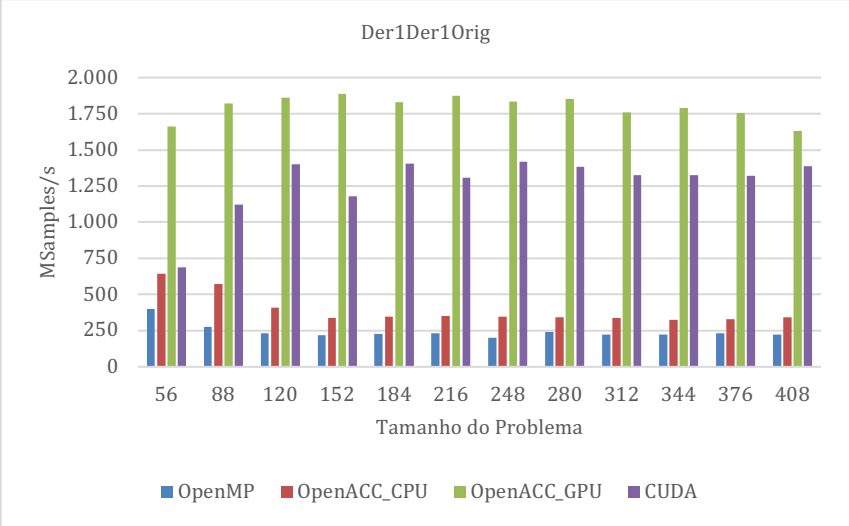

Figura 2: Velocidade da Versão Der1Der1Orig

As conclusões com relação ao custo da portabilidade obtidos na versão Original repetem-se nesta versão: a codificação portátil OpenACC é consistentemente mais rápida que as não portáteis, tanto na CPU quanto na GPU.

Para avaliação quantitativa, a Tabela 1 apresenta a média das velocidades sobre todas as grades (em MSample/s) para cada versão e codificação.

Tabela 1: Velocidade Média em cada Codificação e Versão

Versão	Velocidade Média (MSamp/s)			
	OpenMP	OpenACC_CPU	OpenACC_GPU	CUDA
Original	248,45	342,75	916,86	697,07
Der1Der1Orig	244,02	390,18	1.796,45	1.271,33

Na versão original, OpenACC é 38% mais rápido que OpenMP e 32% mais rápido que CUDA. Na versão Der1Der1Orig, OpenACC é 60% mais rápido que OpenMP e 42% mais rápido que CUDA. Logo, portabilidade não causa perda de desempenho – ao contrário, há ganho de desempenho.

A Tabela 1 permite calcular o aumento de velocidade da versão Der1Der1Orig com relação à Original. Em OpenMP, a versão Der1Der1Orig é ligeiramente mais lenta que a Original (98% da velocidade da Original). Entretanto, em todas as outras codificações, a versão Der1Der1Orig é mais rápida que a Original (14% em OpenACC na CPU, 96% em OpenACC na GPU e 82% em CUDA). Embora esta otimização seja independente de máquina, concluímos que o ganho desta otimização é muito significativo na GPU e pouco significativo na CPU.

Em suma, OpenACC permite portabilidade entre CPU e GPU, com ganhos de velocidade com relação às codificações não portáteis. Já a substituição da derivada cruzada pela derivada primeira da derivada primeira é muito proveitosa na GPU e pouco proveitosa na CPU.

5 Versão Der1Der1HM

A versão Der1Der1Orig utiliza mais memória que a versão Original, por incluir quatro novos campos para armazenar as derivadas primeiras. A inclusão aumenta a quantidade de memória necessária em aproximadamente 20%, limitando o tamanho da grade simulada.

Como as derivadas primeiras são utilizadas apenas para calcular as derivadas cruzadas, os valores dessas derivadas em determinada profundidade (*z*) são necessários apenas em algumas profundidades acima (4) e abaixo (4) dessa profundidade. Logo, o campo completo de cada derivada primeira pode ser substituído por um buffer circular, que armazena os valores dessa derivada em poucas profundidades (9), reduzindo o aumento de memória. Ainda mais, como o buffer circular é muito menor que o campo completo, é possível melhorar o uso da hierarquia de memória, acelerando a computação.

Para implantar essa otimização, optamos por codificar passo intermediário, que continua armazenando as derivadas primeiras em campos completos, mas move sua computação para o interior do laço em profundidade no corpo de *Propagate*. Antes do laço em profundidade, a inicialização de *Propagate* calcula todas as derivadas primeiras necessárias para calcular as derivadas cruzadas na primeira profundidade, exceto na última (maior) profundidade. Cada iteração do laço calcula as derivadas primeiras na última profundidade necessária, completando os dados para calcular as derivadas cruzadas e aplicar a modelagem Fletcher.

Denominamos esta versão *Der1Der1HM (High Memory)*, implementada sobre *Der1Der1Orig* alterando apenas a estrutura de laços em *Propagate* de cada *back-end*.

É relevante citar os impactos dessa modificação no paralelismo. As derivadas primeiras utilizadas em uma profundidade são calculadas nas poucas profundidades anteriores e nessa profundidade. Consequentemente, o laço nas profundidades em *Propagate* não pode ser paralelizado, por conter dependências entre iterações consecutivas.

A inicialização de *Propagate* é totalmente paralelizada, pois o cálculo das derivadas primeiras em qualquer ponto das profundidades iniciais é independente do mesmo cálculo nos demais pontos. O laço central de *Propagate* (em `z`) é obrigatoriamente sequencial, mas é composto por duas regiões paralelas. A primeira região paralela calcula a derivada primeira em todos os pontos do plano `xy` da última profundidade necessária. A segunda região paralela aplica a modelagem Fletcher em todos os pontos do plano `xy` da profundidade atual. Obrigatoriamente há uma barreira para sincronizar a execução das

duas regiões paralelas, pois as derivadas calculadas na primeira região são utilizadas na segunda região.

5.1 Avaliação de Desempenho da Versão Der1Der1HM

Repetimos a avaliação de desempenho padronizada. A Figura 3 apresenta a velocidade de execução desta versão, utilizando os mesmos formato e nomenclatura das figuras correspondentes nas versões anteriores.

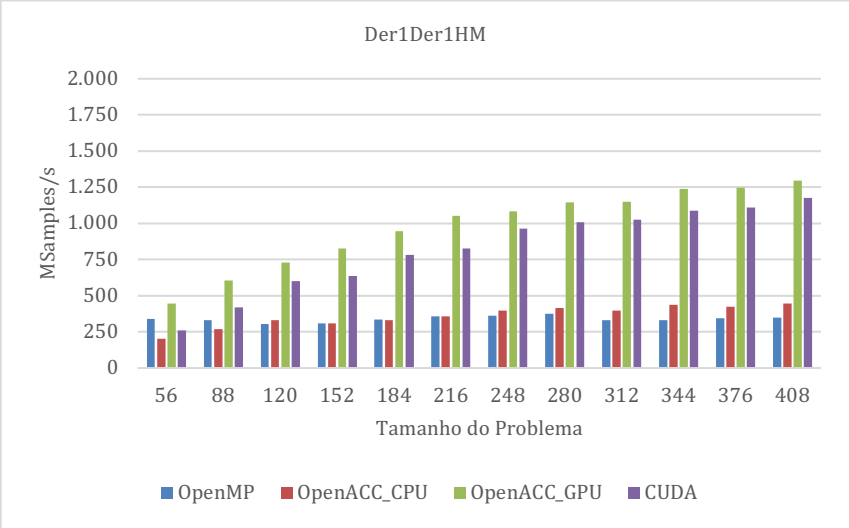

Figura 3: Velocidade da versão Der1Der1HM

A codificação portátil continua mais rápida que a codificação não portátil tanto na CPU quanto na GPU. Observamos que excepcionalmente nos dois problemas menores a codificação OpenMP é mais rápida que a codificação portátil, mas essa vantagem é invertida nos demais dez problemas.

É notável observar que a velocidade de execução cresce com o tamanho do problema, o que não ocorre nas versões anteriores. Cogitamos que o aumento de velocidade com o tamanho do problema se deva à redução do custo relativo da sincronização: enquanto o custo das regiões paralelas cresce quadraticamente com o tamanho do problema (pois em `xy`), o custo do sincronismo cresce linearmente (pois em `z`).

Para avaliação quantitativa, apresentamos a Tabela 2, que contém a média das velocidades sobre todos os problemas (em MSample/s) para as três versões e para todas as codificações. As duas primeiras linhas são idênticas às da Tabela 1.

Tabela 2: Velocidade Média em cada codificação e versão

Versão	Velocidade Média (MSamp/s)			
	OpenMP	OpenACC_CPU	OpenACC_GPU	CUDA
Original	248,45	342,75	916,86	697,07
Der1Der1Orig	244,02	390,18	1.796,45	1.271,33
Der1Der1HM	339,27	359,89	980,44	824,82

Na versão Der1Der1HM, a codificação OpenACC é 6% mais rápida que a versão OpenMP na CPU e 19% mais rápida que a versão CUDA, demonstrando numericamente que a codificação portátil é mais rápida que as não portáteis.

Houve ganho de desempenho desta versão com relação à Original em todas as codificações (37% em OpenMP, 5% em OpenACC na CPU, 7% em OpenACC na GPU e 18% em CUDA). Mas houve perda de desempenho desta versão com relação à versão Der1Der1Orig em todas as codificações exceto na OpenMP, que é 36% mais rápida nesta versão que em Der1Der1Orig. Atribuímos a perda de desempenho à inserção de sincronização.

6 Versão Der1Der1LM

O objetivo desta versão é reduzir o aumento de memória das duas versões anteriores com relação à versão Original. Alteramos a versão Der1Der1HM, armazenando a derivada primeira em um buffer circular. O buffer é constituído por nove planos `xy` da grade completa, contendo as derivadas primeiras nas nove profundidades utilizadas pela modelagem Fletcher. A cada nova profundidade apenas um plano é substituído: o plano da menor profundidade é substituído pelo plano da maior profundidade.

Esta versão foi construída alterando Der1Der1HM em diversos trechos de código, incluindo *sample.h*. As alterações mais relevantes concentram-se em *Propagate* e são descritas a seguir.

A inicialização de *Propagate* é idêntica à da versão anterior, mas armazenando as derivadas no buffer circular. O laço em profundidades em *Propagate* inicia atualizando os ponteiros do buffer circular, determinando o plano a substituir. Segue calculando a derivada restante, armazenada no plano substituído. Termina aplicando a modelagem Fletcher, utilizando as derivadas primeiras armazenadas no buffer circular.

O paralelismo é similar ao da versão anterior, com uma região paralela contendo a inicialização, seguida pelo laço sequencial em profundidades, contendo duas regiões paralelas, a primeira para calcular a derivada restante e a segunda para aplicar Fletcher. A diferença está na indexação do buffer circular e na atualização dos ponteiros que implementam o buffer circular. Tal atualização ocorre no início das duas regiões paralelas e no interior do laço em profundidades.

Denominamos esta versão *Der1Der1LM* (*Low Memory*).

6.1 Avaliação de Desempenho de Der1Der1LM

A Figura 4 apresenta a velocidade de execução desta versão, mantendo formato e nomenclatura das figuras similares nas versões anteriores.

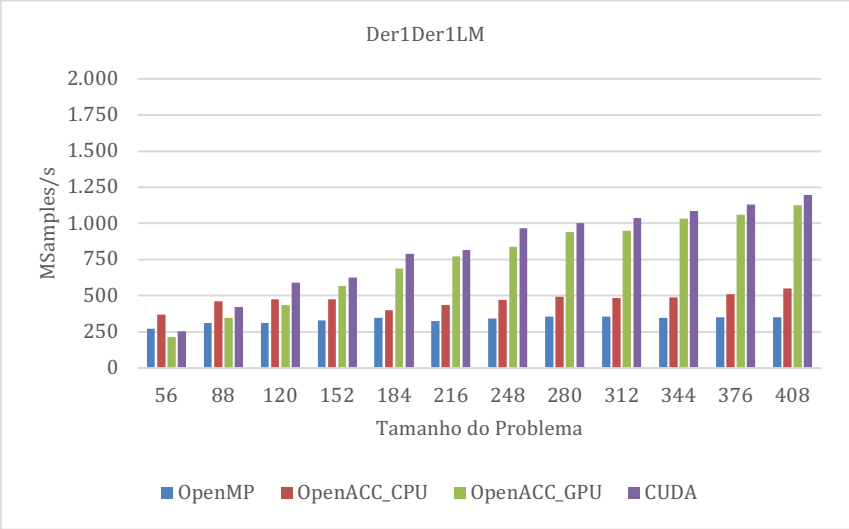

Figura 4: Velocidade da versão Der1Der1LM

A Figura 4 demonstra que a codificação portátil OpenACC continua mais rápida que a codificação OpenMP na CPU. Entretanto, a codificação portátil OpenACC é ligeiramente mais lenta que a codificação CUDA na GPU.

O aumento da velocidade com o tamanho do problema detectado na versão Der1Der1HM continua a existir, pelos mesmos motivos.

A finalidade desta versão era reduzir o uso de memória, o que foi atingido. Enquanto as versões Der1Der1Orig e Der1Der1HM requeriam aproximadamente 20% a mais de memória que a versão original, a versão Der1Der1LM requer apenas 0,4% a mais de memória que a versão original.

Novamente, apresentamos na Tabela 3 a média das velocidades sobre todos os problemas, para análise quantitativa. Por comodidade, repetimos a Tabela 2, acrescida das velocidades da nova versão.

Tabela 3: Velocidade Média em cada Codificação e Versão

Versão	Velocidade Média (MSamp/s)			
	OpenMP	OpenACC_CPU	OpenACC_GPU	CUDA
Original	248,45	342,75	916,86	697,07
Der1Der1Orig	244,02	390,18	1.796,45	1.271,33
Der1Der1HM	339,27	359,89	980,44	824,82
Der1Der1LM	333,56	468,03	747,80	826,93

Na versão Der1Der1LM a codificação OpenACC é 40% mais rápida que a codificação OpenMP na CPU, mas é 10% mais lenta que a codificação CUDA na GPU.

Esta é a versão mais rápida em OpenACC na CPU dentre todas as versões.

Esta versão é a segunda mais rápida em OpenMP na CPU, ligeiramente mais lenta (1,6%) que a versão mais rápida (Der1Der1HM), mas usando menos memória.

Entretanto, esta versão não é a mais rápida na GPU, provavelmente pelo custo do sincronismo. Para execução na GPU, recomendamos OpenACC na versão Der1Der1Orig, caso os requisitos de memória dessa versão não sejam limitantes. Caso a memória seja um fator limitante, recomendamos OpenACC na versão Original.