oneAPI を使用して有限差分法を高速化

CUDA* ベースのステンシル計算を DPC++ へ移行

Clícia Pinto インテル コーポレーション テクニカルリーダー兼パフォーマンス・エンジニア
Lucas Batista SENAI CIMATEC Supercomputing Center HPC 開発者
Pedro de Santana SENAI CIMATEC Supercomputing Center HPC 開発者
Georgina González SENAI CIMATEC Supercomputing Center HPC 開発者

リバース・タイム・マイグレーション（RTM）では、有限差分法（FD）を使用して音響波動方程式の近似値を計算します。これは、RTM アプリケーションの計算上のボトルネックとなっており、炭化水素探査のリソースを割り当てる際に、タイムリーな結果と効率を保証するため最適化する必要があります。この記事では、インテル® DPC++ 互換ツールを使用して、CUDA* ベースの RTM コードをデータ並列 C++（DPC++）に移行した経験について説明します。

RTM の概要と入力データ

いくつかの地震探査法では、FD スキームにおいて、波動方程式の数値解としてステンシルが適用されています。石油ガス産業で地下構造の画像を生成するために広く使用されている RTM でもこの方法を採用しています。この方法には利点がありますが、2 つの大きな計算上のボトルネックがあります。伝播ステップで浮動小数点演算が多いことと、メインメモリーへの波動場の保存が困難なことです。これらのボトルネックの影響を軽減するため、エンジニアはタスクの並列性と計算リソースの最適化の両方を追求し、さまざまなアクセラレーターで実行可能なソリューションを設計しています。この方法の最適化は、井戸掘削におけるエラーの可能性を軽減させるため、探査地球物理学に大きな経済的利点をもたらします。Claerbout[1] が提案したように、RTM アルゴリズムは通常、前方時間伝播、後方伝播、および画像条件の相互相関を行います。RTM アルゴリズムのフローチャートでは、これらのステップをホストとデバイスの通信とともに示しています（**図 1**）。

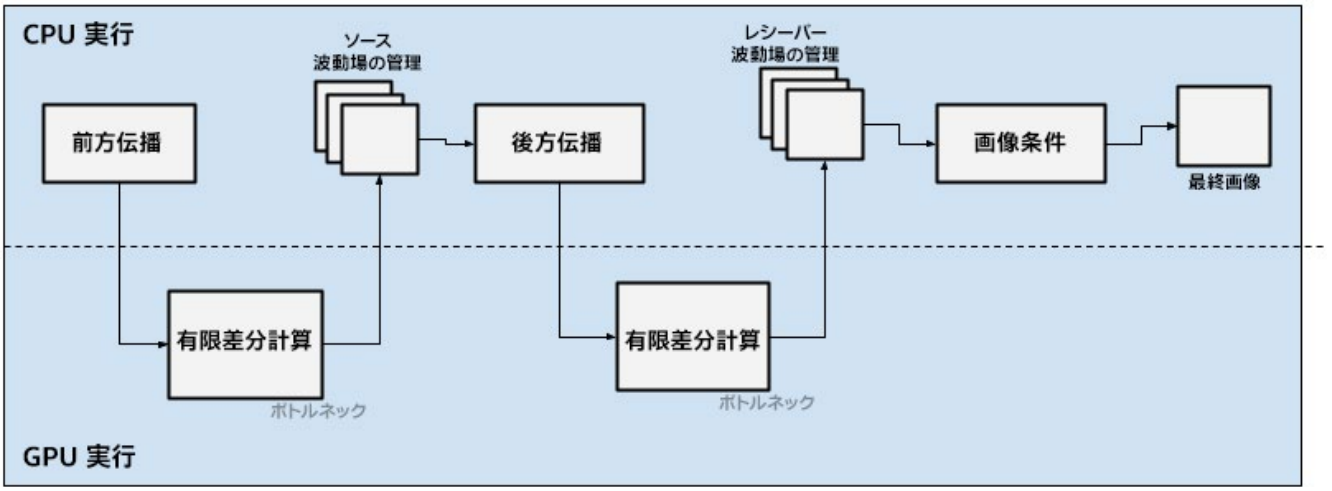

図 1. 簡易 2 次元 RTM フローチャート

図 2 は、2 次元の CUDA* ベースの実装におけるステンシル計算を示しています。**`order`** は FD スキームの次数、**`nx`** と **`nz`** は 2 次元音響速度モデルを表す入力行列のサイズ、**`p`** はソース / レシーバーの波動場、**`cx`** と **`cy`** はそれぞれ FD 係数の x 軸と y 軸、**`l`** は外挿された波動場を表しています。RTM アルゴリズム全体では、ベクトル **`p`** は異なる時間ステップにおける圧力ポイントの状態を格納しています。RTM は、計算量とデータ量が多いため、専用の処理ユニットによる高速化に適しています。

```
Input: order, nx, nz,*p, *l, *cx, *cy
Output: *l
Assignments: h_order ← order/2
i ← h_order + blockId.x * blockDim.x + threadId.x
j ← h_order + blockId.y * blockDim.y + threadId.y
mult ← i * nz
if < nx – h_order then
         if < nz - h_order then
                  for k = 0 to h_order do
                           aux =  k - h_order
                           aux += p[mult + j+aux] * cz[k]
                           aux += p[(i+aux)*nz + j] * cx[k]
                  end for
                  l[mult + j] = accz + accx
                  accz = 0
                  accx = 0
          end if
end if
```

図 2. RTM ステンシル計算アルゴリズム

リファレンス RTM から oneAPI への移行

インテル® DPC++ 互換性ツールは、CUDA* 言語カーネルやライブラリーの API 呼び出しを DPC++ に移行するのを支援します。通常、CUDA* ソースコードの 80 ~ 90% は自動的に移行されるため、このプロセスは「準備」、「移行」、「レビュー」の 3 つの段階で構成されます。準備段階では、ソースコードを移行ツールに適応させることを目指します。この段階では、すべての CUDA* ヘッダーファイルがデフォルトの場所、または **`--cuda-include-path=<path/to/cuda/include>`** オプションで指定したカスタムの場所にあることを確認する必要があります。移行段階では、インテル® DPC++ 互換性ツールがオリジナルのアプリケーションを入力として受け取り、注釈付きの DPC++ コードを生成します。レビュー段階では、自動変換されたコードを確認し、移行されなかったコードを手動で変換するための注釈をレビューし、コードの改善の可能性を探ります。

最初の移行では、**図 3** に示すように、インテル® DPC++ 互換性ツールによって、CUDA* のメモリーコピー API 呼び出しが **`sycl::queue.memcpy()`** に移行されることを確認しました。正常に動作し、エラーのない DPC++ ソースコードが得られても、明示的なメモリー管理では最高のパフォーマンスが得られない可能性があります。メモリー管理の改善について調査するため、移行後のソースコードを手動で変更し、各データ・オブジェクトで SYCL* のバッファーとアクセサーを使用するようにしました。

```
dpct::device_ext &dev_ct1 = dpct::get_current_device();
sycl::queue &q_ct1 = dev_ct1.default_queue();
q_ct1.memcpy(d_p, p[0], mtxBufferLength).wait();
             q_ct1.memcpy(d_pp, pp[0], mtxBufferLength).wait();
             q_ct1.memcpy(d_v2, v2[0], mtxBufferLength).wait();
             q_ct1.memcpy(d_coefs_x, coefs_x, coefsBufferLength).wait();
             q_ct1.memcpy(d_coefs_z, coefs_z, coefsBufferLength).wait();
             q_ct1.memcpy(d_taperx, taperx, brdBufferLength).wait();
             q_ct1.memcpy(d_taperz, taperz, brdBufferLength).wait();
```

図 3. 移行後のメモリー管理

```
void fd_forward(int order, float **p, float **pp, float **v2, int nz, int nx, int nt,
int is, int sz, int *sx, float *srce, int propag)
{
dpct::device_ext &dev_ct1 = dpct::get_current_device();
sycl::queue &q_ct1 = dev_ct1.default_queue();
sycl::range<3> dimGrid(1, gridz, gridx);
sycl::range<3> dimGridTaper(1, gridBorder_z, gridx);
sycl::range<3> dimGridSingle(1, 1, 1);
sycl::range<3> dimGridUpb(1, 1, gridx);
sycl::range<3> dimBlock(1, sizeblock, sizeblock);
{
sycl::buffer<float, 1> *b_p = new sycl::buffer<float, 1>(p[0],
sycl::range<1>(nxe*nze));
sycl::buffer<float, 1> *b_pp = new sycl::buffer<float, 1>(pp[0],
sycl::range<1>(nxe*nze));
sycl::buffer<float, 1> b_v2(v2[0], sycl::range<1>(nxe*nze));
sycl::buffer<float, 1> b_coefs_x(coefs_x, sycl::range<1>(order+1));
sycl::buffer<float, 1> b_coefs_z(coefs_z, sycl::range<1>(order+1));
sycl::buffer<float, 1> b_taperx(taper_x, sycl::range<1>(nxb));
sycl::buffer<float, 1> b_taperz(taper_z, sycl::range<1>(nxb));
sycl::buffer<float, 1> *b_swap;
for (int it = 0; it < nt; it++){
         b_swap  = b_pp;
         b_pp = b_p;
         b_p = b_swap;
         kernel_tapper()
         kernel_lap()
         kernel_time()
         kernel_scr()
}
}
}
```

図 4. DPC++ 前方伝播 (簡易バージョン)

図 4 は、DPC++ に移行された前方時間伝播を行う関数です。ここで、**nt** は波動方程式のモデル化に必要な時間ステップの数です。このアルゴリズムでは、データ・オブジェクトが、デバイスのメモリーの制御と変更に使用されるバッファーとして定義されています。後方伝播も同じ構造になっています。前方関数と後方関数はどちらも、GPU 実行を処理するためカーネルを呼び出します。

```
q_ct1.submit([&](sycl::handler &cgh) {
auto acc_pr = b_pr->get_access<sycl::access::mode::read_write>(cgh);
auto d_laplace_ct4 = d_laplace;
auto acc_coefs_x = b_coefs_x.get_access<sycl::access::mode::read>(cgh);
auto acc_coefs_z = b_coefs_z.get_access<sycl::access::mode::read>(cgh);
cgh.parallel_for(
      sycl::nd_range<3>(dimGrid * dimBlock, dimBlock),
      [=](sycl::nd_item<3> item_ct1) {
              kernel_lap(order, nx, nz, acc_pr, d_laplace_ct4,
              acc_coefs_x, acc_coefs_z, item_ct1);
      });
});
```

図 5. DPC++ の kernel_lap 関数呼び出し

図 5 は、**kernel_lap** 呼び出しを使用した DPC++ カーネルの呼び出しを示しています。これは、ステンシル計算のメイン・プロシージャーです。移行したアプリケーションの各カーネルは、特定のデバイス用のキューに投入され、並列カーネルが起動する前にデータアクセス要件を満たす必要があります。

最終的な RTM 画像を使用して、オリジナルの CUDA* 実装と移行した DPC++ コードを比較します。地震画像の生成には、**表 1** に示す入力パラメーターと、**図 6** に示す速度モデルを使用します。CUDA* 実装と DPC++ 実装のそれぞれの出力画像を**図 7** に示します。DPC++ 実装の最終画像は、リファレンスと比較して十分な精度を達成しています。

パラメーター	値
z 軸のポイント	195
x 軸のポイント	375
タイムステップ	1700
z 軸のサンプル間隔	10
x 軸のサンプル間隔	10
タイムステップのサンプル間隔	0.001
周波数ピーク	20.0
ショット回数	6
オーダー	8

表 1. ここで紹介したモデル固有の
パラメーターの説明

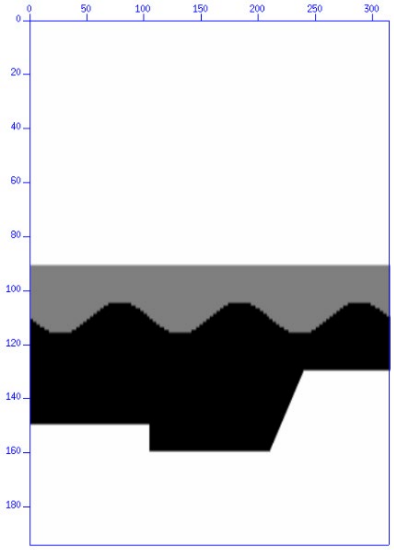

図 6. Koslov 速度モデル

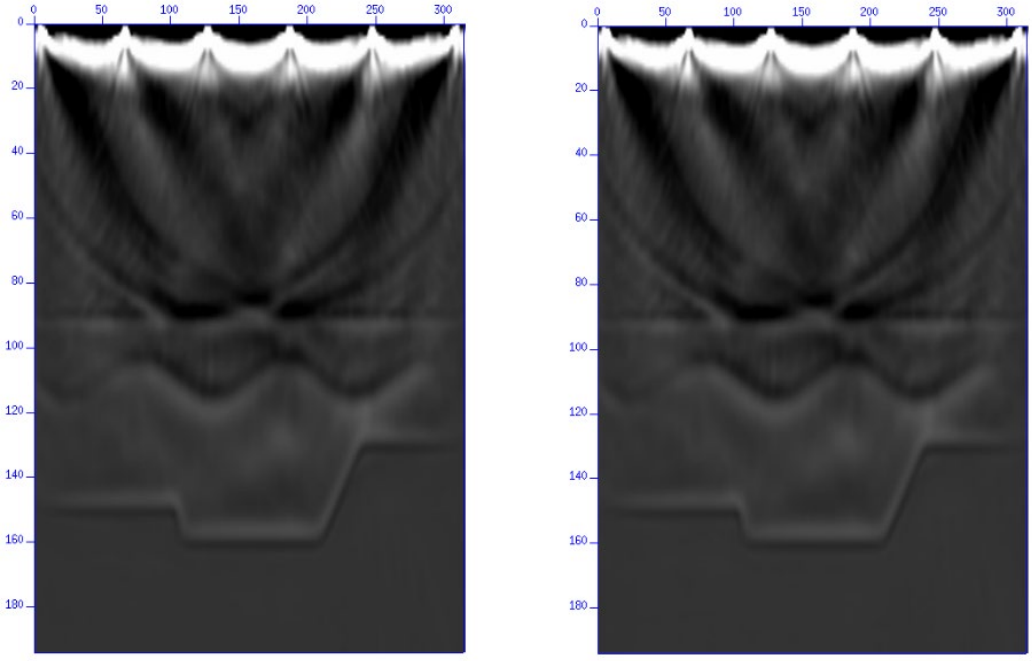

図 7. オリジナルの CUDA* ベースの RTM ソースコード（左）と
移行後の DPC++ コード（右）によって生成された地震画像

メモリー管理の改善

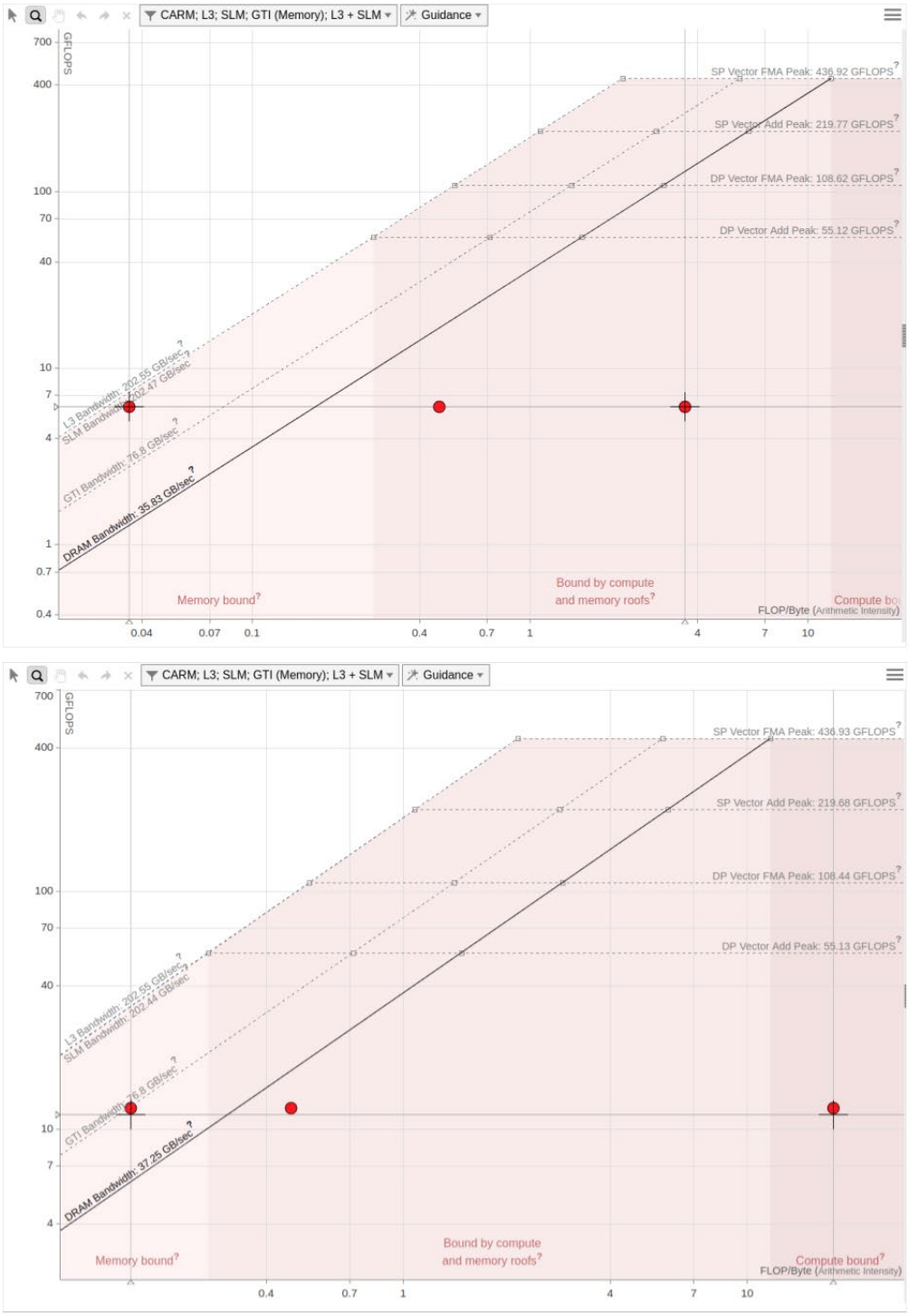

図 8. 第 9 世代インテル® HD グラフィックス NEO 上で明示的なメモリー割り当て（上）とバッファー / アクセサーによるメモリー管理（下）を使用して実行した DPC++ バージョンのルーフライン・ビュー

DPC++ のメモリー管理オプションを調査するため、移行したアプリケーションからバッファー / アクセサーベースのバージョンを開発します。これより、デバイス上で明示的にメモリーを割り当てたり、解放したりする必要がなくなります。また、異なる処理ユニットとの間のデータ転送を管理する必要もありません。これを実現するには、構造体ごとにバッファー / アクセサーを作成し、それらをデバイスに明示的にコピーします。バッファーは計算終了後に破棄され、データはホストにコピーバックされます（メモリー同期）。括弧は、バッファー定義の周囲にデータ・オブジェクトを共有できるスコープを作成します。実行がこのスコープ外になると、実行フロー間の同期が行われ、バッファーが破棄されます。オリジナルと変更した移行ソースコードを比較するため、インテル® Advisor でルーフライン解析を行い、ハードウェアの制限とシステム上の異なるメモリー階層間のデータ・トランザクションを評価することで、パフォーマンスを推定します。

図 8 は、シングル・タイム・ステップのみの単純化された RTM 実行のルーフライン・グラフです。浮動小数点演算の数が減っているので、低パフォーマンス・メトリックが期待できます。上のグラフは、明示的なデータ管理を使用する移行後のソースコードのルーフラインを示しています。6.052 GFLOPS のパフォーマンスと 3.617 FLOP/ バイトの演算強度を達成しています。演算強度とは、データ量移動量（メモリー・トラフィック）に対する浮動小数点演算の総数の割合です。下のグラフは、バッファー / アクセサーバージョンのルーフラインを示しており、パフォーマンスは 2 倍の 12.246 GFLOPS、演算強度は 17.896 FLOP/ バイトを達成しています。

まとめ

この記事では、インテル ® DPC++ 互換性ツールを使用して RTM コードを CUDA* から DPC++ に移行し、インテル® Advisor でチューニングして、oneAPI の概念を実証しました。移行後のソースコードは、アプリケーションのアルゴリズム実行フローが独自の構造に統一されているため、より読みやすく、保守も容易になりました。また、移行後にバッファーやアクセサーを適用してメモリー管理を行うことで、パフォーマンスと演算強度をさらに向上できました。この記事で紹介したソースコードと手順[2] は公開リポジトリーにあります。

関連情報

1. Claerbout, JF. Toward a unified theory of reflector mapping. Geophysics, v. 36, n. 3, p. 467-481, 1971.
2. **https://github.com/cs2isenaicimatec/finite-difference-computation/**（英語）