エディバラ大学学生フォーミュラ・チームの CUDA* から SYCL* への旅

この記事は、Codeplay Blogs に公開されている「Edinburgh Uni Formula Student's Journey from CUDA® to SYCL™」の日本語参考訳です。原文は更新される可能性があります、原文と翻訳文の内容が異なる場合原文を優先してください。

私は、2022 年に Edinburgh University Formula Student (EUFS) ドライバーレス・チームで計画と制御のリーダーを務めました。EUFS はエディンバラ大学の学生で構成されるチームで、学生フォーミュラ UK 大会に出場するため、自律走行レーシングカーを製作しています。これまで 5 年連続で優勝しており、この連勝記録を伸ばすことが目標です。Codeplay は、過去 3 年間、スポンサーとしてだけでなく、技術指導の提供により、EUFS チームをサポートしてきました。

大会では、未知のコースをレーシングカーで 10 周するという課題があります。コースにはコーンが設置されており、コースを最速で周回し、コーンにぶつかった回数が最も少ないチームが、最も多くのポイントを獲得します。この課題に効率的に取り組むため、EUFS では課題を 2 つのステージに分けました。最初のステージでは、コースを周回しながら、SLAM アルゴリズムを使ってすべてのコーンをゆっくりとマッピングします。コース全体のマッピングが完了したら、ラップタイムを最短にするため、高速走行ステージに切り替えます。

このプロジェクトの目標は、コースの全体像が判明したら高速走行を行うコントローラーを組み込むことでした。そのため、https://autorally.github.io/ (英語) をベースに、ハンドリングの限界で非常にアグレッシブな運転を可能にする MPPI (Model Path Predictive Integral) コントローラーを使用しました。このコントローラーは、何千もの潜在的な軌道をサンプリングし、それぞれの軌道にコストを割り当て、モンテカルロ法を使用して最適な軌道を見つけます。このプロセスでは、GPU の高度な並列性を利用して、車の非線形ダイナミクス・モデルを使用しながら、リアルタイムで何千もの軌道をサンプリングできます。

このプロジェクトは元々 ROS1 フレームワーク向けに CUDA* で記述されていたため、これを SYCL* に移行して現在のスタックと統合する方法を模索しました。SYCL* を選んだ理由は、CUDA* 互換のデバイスに縛られることなく自律走行車の GPU を自由に選択でき、さらに、カーネルコードを標準の ISO C++ で記述できるためです。

これは、EUFS sim (英語) シミュレーションで MPPI コントローラーが動作している短い動画です。オープンソースの MPPI コントローラー (英語) でコードを確認できます。

CUDA* から SYCL* への移行

CUDA* や SYCL* でプログラミングをしたことがないため、CUDA* から SYCL* にコードを移行し、同時にコントローラーが既存のシステムと互換性があることを確認できるかどうか、最初は少し懐疑的でした。幸いにも、CUDA* 実装はほとんど終わっていました。しかし、コードを理解する必要がありました。CUDA* のドキュメントはどれも非常に難しく感じられたので、まず SYCL* で並列プログラミングを学び、それから CUDA* のコードを調べることにしました。このアプローチは非効率的に見えますが、数週間で CUDA* と SYCL* の両方で並列プログラミング (少なくとも必要な範囲) を行えるようになりました。

API を習得後、コードを移行する方法を調査しました。すべてを手作業で移行するのは時間がかかりますが、オープンソースの SYCLomatic プロジェクト (英語) を使用することで、CUDA* から SYCL* への移行のほとんどを自動的に行うことができました。しかし、互換性ツールでは適切に移行できないものもあり、移行作業のほとんどはその解決に費やされました。例えば、同じ USM を使用するカーネル間でコンテキストが一貫していなかったり、SYCL* に CUDA* テクスチャーに相当するものがないことなどが挙げられます。また、sycl::queue の共有ポインターメンバーは仮想デストラクターを持つオブジェクトであり、暗黙的にメモリーを解放するため、SYCL* カーネルでは許可されていないことを忘れていて、対応が必要になりました。

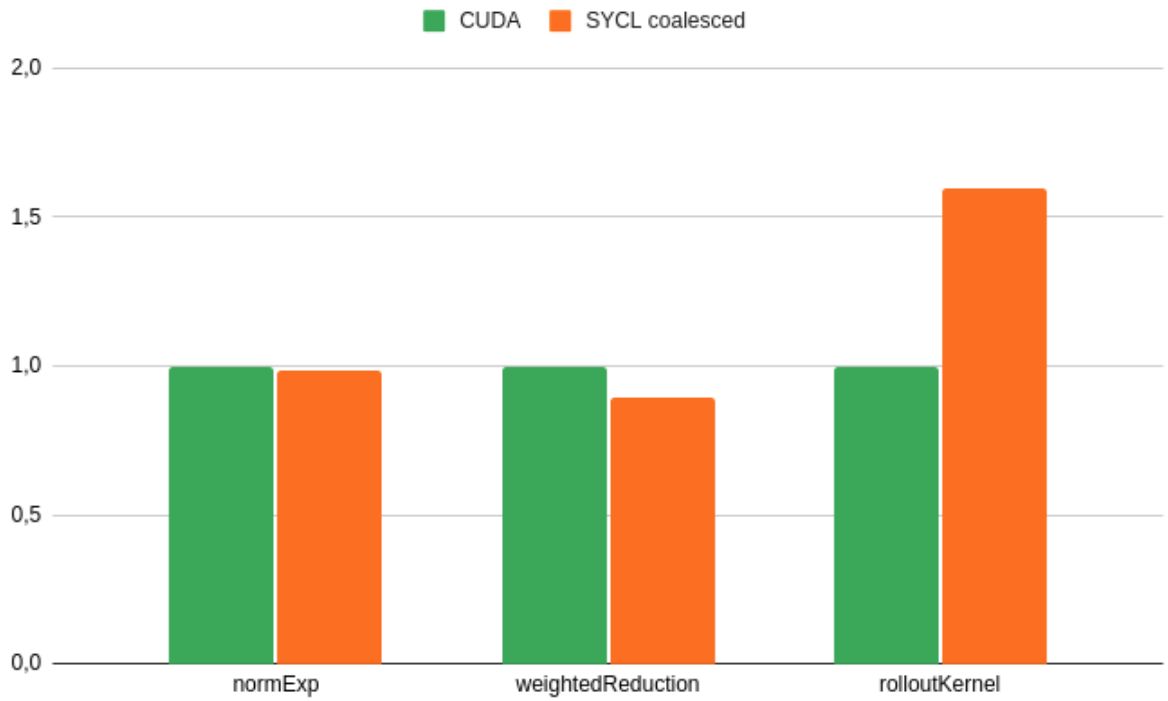
すべてのカーネルが同じコンテキストを使用するようにするのは簡単で、ほんの少しコードを追加するだけで済みました。CUDA* テクスチャーから画像のリスト表現への移行には、もう少し作業が必要でした。これは、低コストの軌道を定義するためコントローラーに渡されるコストマップに必要でした。CUDA* テクスチャーは正規化されているので、画像のリスト表現からコストを計算するため、線形代数を追加する必要がありました。sycl::image を使用することもできますが、マルチチャネルは必要ありませんでしたし、リスト表現のほうが使いやすいと感じました。しかし、今後コストマップでほかのチャネルを使用する必要性が生じたら、変更する可能性があります。最後の sycl::queue の共有ポインターメンバーの問題は、コードで仮想メソッドを使用していなかったので、仮想デストラクターがどこから来ているのかを理解するのに時間がかかりました。しかし、特定した後は、カーネル関数と GPU メモリーへのポインターを含み、sycl::queue のメンバーを含まないカーネル・セーフ・クラスを作成することで簡単に回避できました。今にして思えば、これらの問題はすべて、SYCL* フレームワークに関する経験と知識がもっとあれば、短時間で簡単に解決できたでしょう。

パフォーマンスを達成

コントローラーは SYCL* と CUDA* の両方で実装されているので、CUDA* から SYCL* に移行したことでパフォーマンスが低下していないかどうかを評価しました。NVIDIA Nsight* Systems プロファイラーを使用し、3 つのカーネルの実行時間を比較しました。軌跡サンプルの作成と続く軌跡コストの割り当てを行う rolloutKernel、コストを正規化する normExp カーネル、そして多数のサンプルから制御を絞り込む weightedReduction カーネルです。

驚いたことに、当初、SYCL* カーネルはすべて、対応する CUDA* カーネルよりもわずかに遅かったのです。

しかし、これらのカーネルの次元に関して、ささいなミスを犯していることが判明しました。スレッド <0,5,7> と <1,5,7> は、CUDA* では同じラップ内で隣接しており、グローバルメモリーの隣接部分にアクセスしますが、SYCL* では同じラップ内にない場合もあります。つまり、隣接するスレッドがメモリーの隣接部分にアクセスしないので、メモリーアクセスが結合されず遅くなります。幸いにも、この問題は修正するのがそれほど難しくありません。



40 分後、すべてのカーネルが高速化され、normExp カーネルと weightedReduction カーネルは、SYCL* のほうが CUDA* よりも速くなりました。rolloutKernel カーネルも大幅にスピードアップしましたが (約 40%)、CUDA* 実装と比較するとまだ遅いです。

NVIDIA Nsight* Compute を使用したところ、SYCL* 実装はメモリー上でより多くのデータを動かしており、ローカルメモリーを使用していることから、レジスタースピルが発生し、パフォーマンスの低下を招いている可能性があることが分かりました。

学生フォーミュラ・チームにとって、このパフォーマンスの差 (SYCL* で 4.5ms、CUDA* で 2.8ms) は気にならないものでした。今でも、何が原因なのか興味はありますが、優先順位の高いプロジェクトがあったため詳しく調査できませんでした。将来、もっと時間をかけて調査できればと考えています。

EUFS の CUDA* から SYCL* への旅

CUDA* や SYCL* の経験なく、並列計算もしたことがなかったため、このプロジェクトは非常に楽しく、かつ挑戦的で、素晴らしい学習体験となりました。最初は、プロジェクトの規模と複雑さに圧倒されましたが、予想していたよりも困難ではありませんでした。すぐにこのコントローラーで実世界のテストを行えるようになると期待しています。EUFS チームが、この作業を基に、今後さらに優れた、より高速なコードを構築してくれることを願っています。

最後に、プロジェクトを通して多大なるサポートを提供してくださった Codeplay の Joe Todd 氏と Rod Burns 氏に謝意を表したいと思います。

EUFS のオープンソース・プロジェクト (英語) と EUFS のウェブサイト (英語) もぜひご覧ください。
