インテル® SHMEM: GPU と SYCL デバイス向けマルチノード・データ共有を高速化

この記事は、インテルのウェブサイトで公開されている「Intel® SHMEM: Accelerate Multi-Node Data Sharing for GPUs and SYCL* Devices」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は原文を優先してください。

インテル® oneAPI HPC ツールキット 2025.1 のリリースに伴い、インテル® SHMEM (英語) は、オープンソースの GitHub プロジェクトから、分散型マルチノード計算環境におけるリモート・データ・アクセスとメモリー共有向けの、OpenSHMEM 1.5 標準に準拠した区分化大域アドレス空間 (PGAS) プログラミング・モデルを実装した、検証済みの製品リリースへと進化しました。

インテル® SHMEM の特長は、OpenSHMEM 通信 API を拡張し、分散型インテル® Xeon® スケーラブル・プロセッサー構成だけでなく、インテル® GPU 上の SYCL ベースの計算デバイスカーネルにも対応していることです。これにより、インテル® SHMEM は、SYCL デバイス GPU を搭載したインテル® Xeon® 6 プロセッサー・ベースの計算クラスターや、Argonne Leadership Computing Facility (ALCF) (英語) の Aurora エクサスケール・スーパーコンピューター (英語) や Cambridge Open Zettascale Lab (英語) の Dawn スーパーコンピューター (英語) などのスーパーコンピューターで動作する高性能な大規模データセット・アプリケーションにとって理想的なコンパニオン・ライブラリーおよび API となります。

OpenSHMEM ベースのソリューションとの違いは、NVSHMEM や ROC SHMEM とは異なり、インテル® SHMEM の SYCL 拡張は、ユーザーを特定のベンダー固有のプログラミング環境に制限しないことです。SYCL は本質的にマルチアーキテクチャーおよびマルチベンダーに対応しており、これは OpenSHMEM のようなオープン API の精神により合致するとインテルは考えています。ポータブルなプログラミング環境と一貫性を持つように設計された SYCL は、OpenSHMEM の GPU 拡張に対する標準化の取り組みに貢献をすると感じています。

ソフトウェア・アーキテクチャー

インテル® SHMEM は、区分化大域アドレス空間 (PGAS) プログラミング・モデルを実装しており、現在の OpenSHMEM 標準におけるホスト主導型操作のほとんどを網羅しています。また、GPU カーネルから直接呼び出し可能な新しいデバイス主導型操作も含まれています。インテル® SHMEM が提供するコア機能セットは以下のとおりです。

1. OpenSHMEM 1.5 準拠のポイントツーポイント・リモート・メモリー・アクセス (RMA)、アトミックメモリー操作 (AMO)、シグナリング、メモリー順序付け、同期操作に対するデバイスおよびホスト API サポート。
2. OpenSHMEM 1.5 チーム API に準拠した集合操作に対するデバイスおよびホスト API サポート。
3. RMA、シグナリング、集合操作、メモリー順序付け、同期操作に対する SYCL ワークグループおよびサブグループ・レベル拡張に対するデバイス API サポート。
4. 現在の OpenSHMEM 仕様の C11 汎用ルーチンに代わる完全な C++ 関数テンプレート・セット。

これらの機能の実装詳細と、ランタイム・パフォーマンス向けにどのように最適化されているかについて説明します。

現在の OpenSHMEM メモリーモデルは、C/C++ アプリケーションが使用するホストメモリーの観点から仕様が定められています。これは、仕様の変更や修正を行わない限り、GPU やその他のアクセラレーター・メモリーを対称セグメントとして使用することを制限します。また、GPU メモリーが各プロセシング要素 (PE) によって対称ヒープとして使用される機能的なマッピングで、すべての PE が同時に実行できるように、実行モデルを適応させる必要があります。SYCL プログラミング・モデルでは、統合共有メモリー (USM) によって、ホスト、デバイス、および共有アドレス空間からのメモリーの使用が許可されます。これには、ランタイムのメモリータイプに応じて、異なる通信と完了セマンティクスが必要になる場合があります。

インテル® SHMEM は、SYCL API をサポートする GPU デバイスと密接に連携したマルチノード計算アーキテクチャー向けに、低レイテンシーと高スループットの通信を実現するように設計されています。

このような密接に連携した GPU システムの分散プログラミングにおける重要な課題は、GPU 間の高速ユニファイド・ファブリックを活用しながら、同時にノード外のデータ通信もサポートすることです。さらに、SHMEM アプリケーションはさまざまなサイズのメッセージを転送するため、ライブラリーはメッセージとネットワークのサイズを考慮して、GPU-GPU ファブリック、共有ホストメモリー、またはネットワークのどのパスが最適であるか迅速かつ効率的に判断する必要があります。

また、転送するデータが GPU メモリーにある場合、データをホストメモリーにコピーし、データ到着時に GPU とホストメモリーを同期するよりも、ゼロコピー設計を利用するほうが有利です。GPU 主導の通信は、GPU リモート・ダイレクト・メモリー・アクセス (RDMA) 機能の登場により特に重要になっています。この機能によって、NIC によるゼロコピー転送用に GPU デバイスメモリーを登録できますが、この機能が利用できない場合、ソフトウェア・エンジニアリング上の課題が生じます。

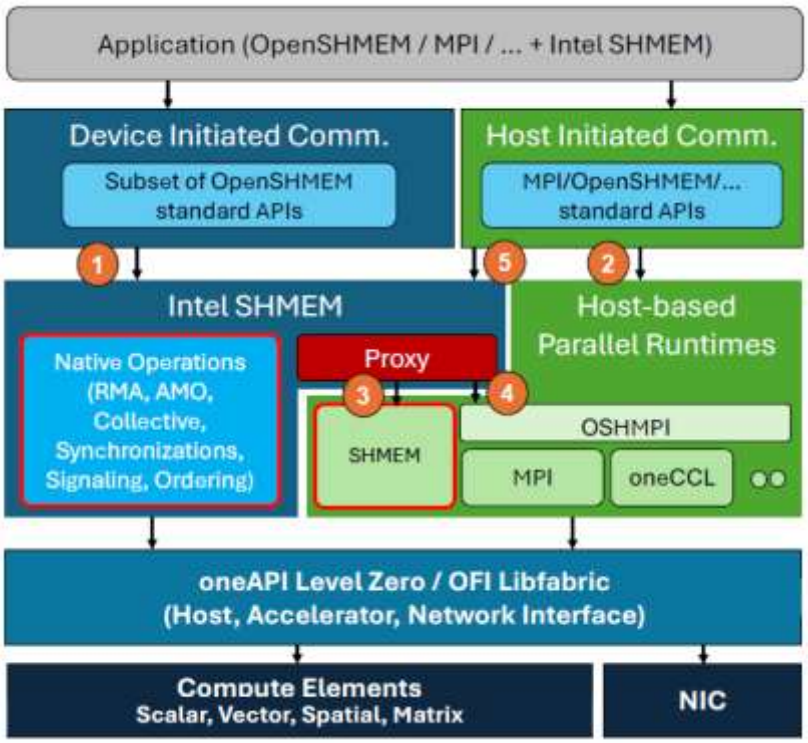

図 1. インテル® SHMEM ソフトウェア・アーキテクチャーの概要

これらの考慮事項を念頭に置いて、インテル® SHMEM ソフトウェア・アーキテクチャーを見てみましょう。図 1 の上部には、OpenSHMEM プログラム内のインテル® SHMEM 呼び出し (または、ホスト・プロキシー・バックエンドとの相互運用性がサポートされている場合は MPI プログラム) から成るアプリケーション層が示されています。

インテル® SHMEM API は、大きく分けて 2 つのカテゴリーに分類されます。

[1] デバイス主導型
[2] ホスト主導型

ノード間 (スケールアウト) 通信では、ホスト側のプロキシースレッドが標準の OpenSHMEM ライブラリーを利用して、特定の GPU 主導型操作をハンドオフします ([3] の円で示されています)。このプロキシー・バックエンドは、純粋な OpenSHMEM として実装する必要はありません。MPI との互換性には、[4] の OSHMPI ソリューションで十分であり、競争力のあるパフォーマンスを提供します。すべての [5] ホスト主導型 OpenSHMEM ルーチンはインテル® SHMEM で利用可能ですが、唯一の注意点は、ルーチンの先頭に shmem ではなく ishmem が付くことです。この命名規則は必須ではありませんが、GPU 抽象化をサポートするさまざまな OpenSHMEM ベースのランタイムを区別できるという利点があります。

GPU のローカルグループ内では、Xe リンクにより、個々の GPU スレッドが他の GPU にあるメモリーに対してロード、ストア、およびアトミック操作を発行できます。個々のロードおよびストアは非常に低いレイテンシーを実現します。多数のスレッドが同時にロードとストアを実行すると、帯域幅は大幅に増加しますが、通信に計算リソース (スレッド) を消費します。

計算と通信をオーバーラップさせるには、利用可能なハードウェア・コピー・エンジンを使用します。これにより、GPU 計算コアが計算でビジー状態の間、Xe リンクをフルスピードで実行できますが、起動レイテンシーが発生します。

インテル® SHMEM は、さまざまな動作モードで、ロード-ストアを直接実行し、GPU スレッド間で負荷を分散し、大規模な転送や非ブロッキング操作にハードウェア・コピー・エンジンを使用するカットオーバー戦略を実行して、これらすべての手法によりパフォーマンスを最適化します。

⇒ 詳細: **Alex Brooks et al. (2024), Intel® SHMEM: GPU-initiated OpenSHMEM using SYCL, arXiv:2409.20476 [cs.DC]** (英語)

インテル® SHMEM コード実行フロー

1. 以下のヘッダーファイルをインクルードします。

   ```
   #include <ishmem.h>
   ```

2. OpenSHMEM ランタイムで ishmem ライブラリーを初期化 (英語) します。

   ```
   ishmem_init();
   ```

3. PE 識別子と PE の総数を照会します。

```
int my_pe = ishmem_my_pe();
int npes = ishmem_n_pes();
std::cout << "Hello from PE " << my_pe << std::endl;
```

4. いくつかの対称オブジェクトを割り当てます。

```
int *src = (int *) ishmem_malloc(array_size * sizeof(int));
int *dst = (int *) ishmem_calloc(array_size, sizeof(int));
```

5. 並列 SYCL カーネル内からソースバッファーとデスティネーション・バッファーを割り当てます。

```
auto e_init = q.submit([&](sycl::handler &h) {
    h.parallel_for(sycl::nd_range<1>{array_size, array_size},
[=](sycl::nd_item<1> idx) {
        int i = idx.get_global_id()[0];
        src[i] = (my_pe << 16) + i;
        dst[i] = (my_pe << 16) + 0xface;
    });
});
e_init.wait_and_throw();
```

6. 通信を行う前に、すべてのソースデータが初期化されていることを保証するため、バリア操作を実行します。

```
ishmem_barrier_all();
```

7. 各 PE がそのソースデータを次の PE に送信する (識別子の値が最大の PE が PE 0 に送信する)、単純なリングスタイルの通信パターンを実行します。

```
/* Perform put operation */
auto e1 = q.submit([&](sycl::handler &h) {
    h.single_task([=]() {
        int my_dev_pe = ishmem_my_pe();
        int my_dev_npes = ishmem_n_pes();

        ishmem_int_put(dst, src, array_size, (my_dev_pe + 1) %
my_dev_npes);
    });
});
e1.wait_and_throw();
```

8. 完了後、すべての通信が完了したことを確認するため別のバリア操作を実行します。

```
ishmem_barrier_all();

int *errors = (int *) sycl::malloc_host<int>(1, q);
*errors = 0;

/* Verify data */
auto e_verify = q.submit([&](sycl::handler &h) {
    h.single_task([=]() {
        for (int i = 0; i < array_size; ++i) {
            if (dst[i] != (((my_pe + 1) % npes) << 16) + i) {
                *errors = *errors + 1;
```

```
            }
        }
    });
});
e_verify.wait_and_throw();

if (*errors > 0) {
    std::cerr << "[ERROR] Validation check(s) failed: " << *errors <<
std::endl;
}
```

9. 割り当てられたすべてのメモリーを解放し、ライブラリーを終了します。対称 ishmem オブジェクトの場合は ishmem_free を呼び出す必要があります。

```
ishmem_free(source);
ishmem_free(target);
sycl::free(errors, q);

ishmem_finalize();
```

デバイス側の実装

インテル® SHMEM では、PE と SYCL デバイスを 1 対 1 でマッピングする必要があります。PE と GPU タイルを 1 対 1 でマッピングすることで、対応する PE の対称ヒープとして独立した GPU メモリー空間が確保されます。プロキシースレッドが GPU メモリー上の対称ヒープに対するホスト側の OpenSHMEM 操作をサポートするには、`FI_MR_HMEM` モードビットをセットしてバッファーを登録する必要があります。インテル® SHMEM が登録する領域を指定するため、GPU メモリーは、ホストメモリー上の標準 OpenSHMEM 対称ヒープと並行してサポートされる外部対称ヒープ上のデバイスメモリー領域として登録されます。

インターフェイスは以下のとおりです。

- `shmemx_heap_create(base_ptr, size, ...)`
- `shmemx_heap_preinit()`
- `shmemx_heap_preinit_thread(requested, &provided)`
- `shmemx_heap_postinit()`

インテル® SHMEM は、すべての OpenSHMEM API をサポートするよう努めています。ただし、初期化と終了 API、すべてのメモリー管理 API など、一部のインターフェイスは、ホストからのみ呼び出す必要があります。これは、CPU/GPU プロキシーの相互作用を処理するデータ構造の設定や、デバイスメモリーの動的割り当ての実行にホスト環境が最適であるためです。

一方、RMA、AMO、シグナリング、集合、メモリー順序付け、ポイントツーポイント同期ルーチンなど、ほとんどの OpenSHMEM 通信インターフェイスは、ホストとデバイスの両方から同一のセマンティクスで呼び出すことができます。

新機能

新しいインテル® SHMEM リリースには、プログラミング・モデルとサポートされる API 呼び出しの詳細を記載した完全なインテル® SHMEM 仕様、サンプルプログラム、ビルドおよび実行手順などが含まれています。

ポイントツーポイントのリモート・メモリー・アクセス (RMA)、アトミックメモリー操作 (AMO)、シグナリング、メモリー順序付け、チーム、集合、同期操作、ストライド RMA 操作を含む、OpenSHMEM 1.5 および 1.6 の機能を使用して、デバイスとホストの両方をターゲットにできます。

インテル® SHMEM のデバイスの API サポートを RMA、シグナリング、集合、メモリー順序付け、同期操作の SYCL ワークグループおよびサブグループ・レベルの拡張に、ホストの API サポートを SYCL キュー順序付け RMA、集合、シグナリング、同期操作に利用できます。

最近追加された主な機能一覧

- `on_queue` API 拡張のサポートにより、OpenSHMEM 操作をホストから SYCL デバイスにキューイングできるようになりました。これらの API では、ユーザーが SYCL イベントのリストを依存関係ベクトルとして提供することもできます。
- OSHMPI (英語) のサポート。適切な MPI バックエンドを使用することで、インテル® SHMEM を OSHMPI 上で実行するように構成できるようになりました。詳細については、「インテル® SHMEM のビルド」(英語) を参照してください。
- インテル® Tiber™ AI クラウド上でのインテル® SHMEM のサポート。こちらの手順 (英語) に従ってください。
- OpenSHMEM スレッドモデルの限定サポート。スレッドの初期化およびクエリールーチンに対するホスト API のサポート。
- ベクトルのポイントツーポイント同期操作に対するデバイスおよびホスト API のサポート。
- インテル® MPI ライブラリーを介した OFI Libfabric (英語) MLX プロバイダー対応ネットワークのサポート。
- 新しい機能の説明と API を追加して仕様 (英語) を更新しました。
- 新しい API の機能をカバーするユニットテスト (英語) のセットを拡大し、改善しました。

⇒ インテル® SHMEM のサポートされている機能/サポートされていない機能 (英語)

今すぐダウンロード

インテル® SHMEM は、スタンドアロン (英語) またはインテル® HPC ツールキットの一部として利用できます。GitHub リポジトリー (英語) からソースを入手することもできます。

インテル® SHMEM を使用して、マルチアーキテクチャー、GPU 主導の分散マルチノード・ネットワークにおけるリモート・データ・アクセスとメモリー共有を、新たなレベルへと引き上げてください。

関連情報

記事

- インテル® SHMEM の紹介
- インテル® SHMEM: SYCL を使用した GPU 主導の OpenSHMEM (英語)

ドキュメント

- GitHub で公開されているドキュメント (英語)
- リリースノート (英語)
- 動作環境 (英語)
- はじめに
    - プログラムの記述 (英語)
    - プログラムのコンパイルと実行 (英語)

サンプルコード

- すべてのサンプルコードを見る (英語)

製品および性能に関する情報

[1] 性能は、使用状況、構成、その他の要因によって異なります。詳細については、http://www.intel.com/PerformanceIndex/ (英語) を参照してください。