

インテル® Advisor – フローグラフ・アナライザー

ヘテロジニアス・ハードウェアの 可視化とチューニング

インテル コーポレーション シニア・テクニカル・コンサルティング・エンジニア Kevin O'Leary

インテル コーポレーション テクニカル・コンサルティング・エンジニア Anoop Madhusoodhanan Prabha

The Intel logo is located in the bottom left corner of the slide. It consists of the word "intel" in a lowercase, sans-serif font, with a registered trademark symbol (®) to its upper right. The logo is white and stands out against the dark blue background. There are also several light blue squares of varying sizes arranged in a grid-like pattern to the left of the logo.

intel®

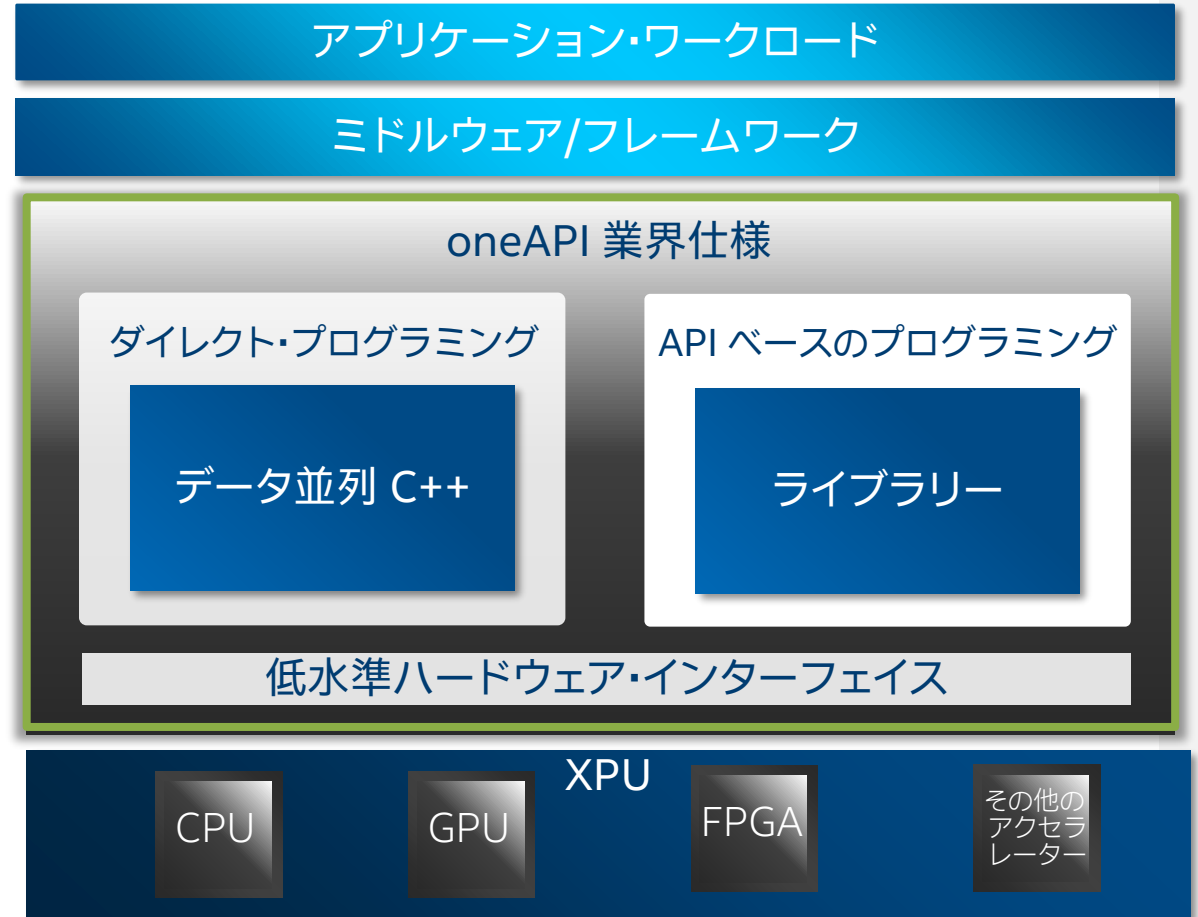
内容

- oneAPI の概要
- インテル® Advisor のフローグラフ・アナライザー機能を使用したデータの収集
- フローグラフ・アナライザーについて

oneAPI 業界イニシアチブ ベンダー固有でないソリューション

- C++ および SYCL* 標準ベースのクロスアーキテクチャ言語であるデータ並列 C++ (DPC++)
- 主要なドメイン固有の関数を高速化するように設計された強力な API
- ベンダーにハードウェア抽象化レイヤーを提供する低水準ハードウェア・インターフェイス
- コミュニティと業界のサポートを促進するオープンなスタンダード
- 複数のアーキテクチャとベンダーでコードを再利用可能

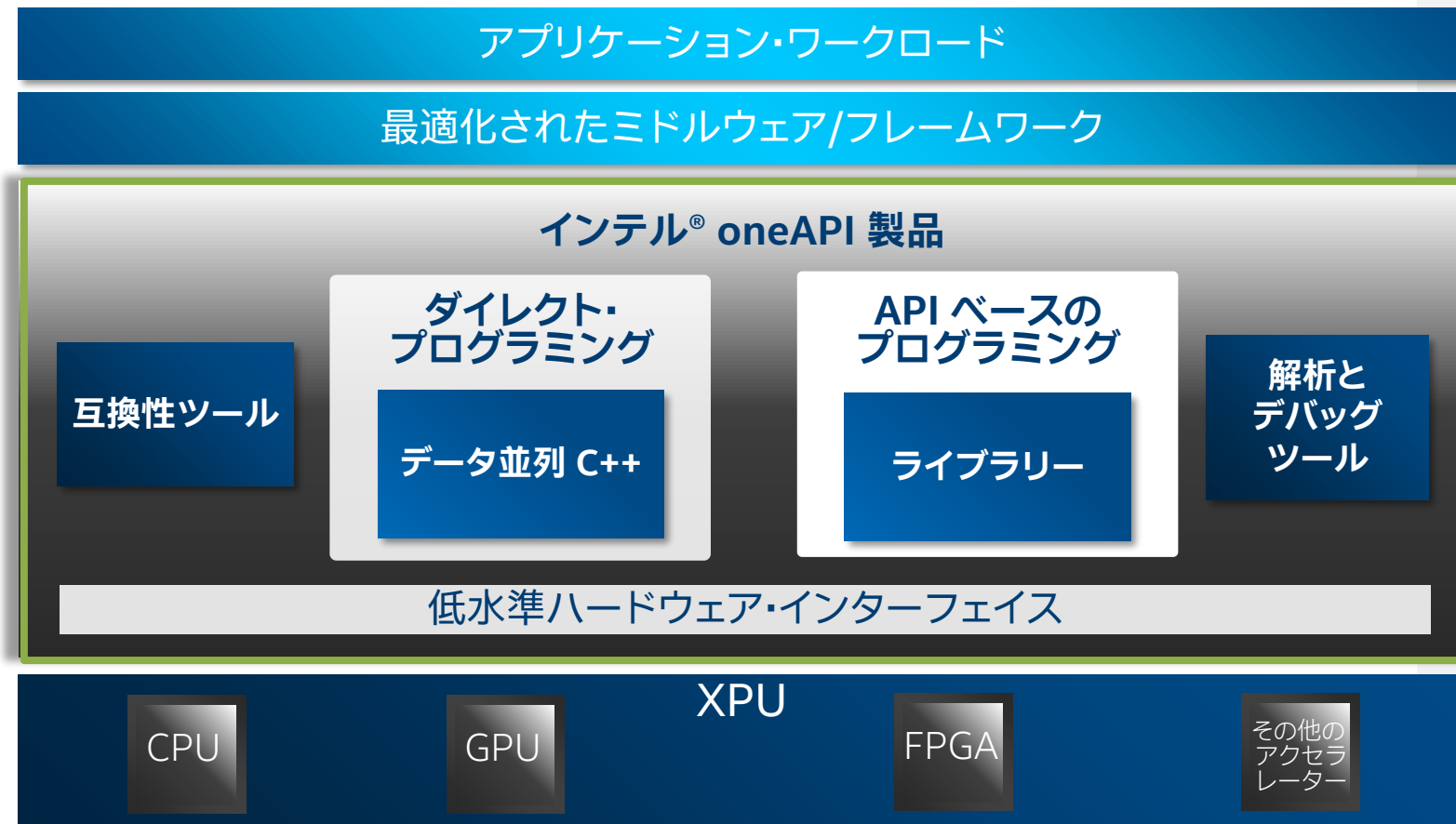
oneapi.com (英語)
皆様からのフィードバックを募集中



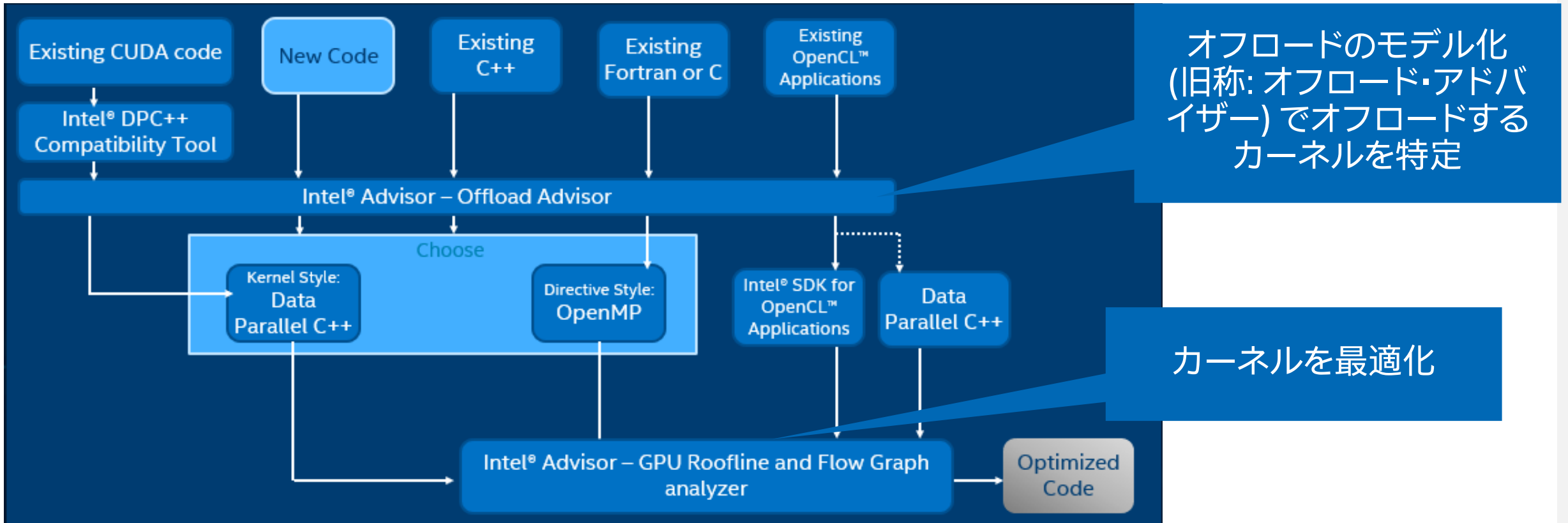
詳細は、oneapi.com (英語) を参照してください。

インテル® oneAPI 製品

- ベース・ツールキットとドメイン固有のアドオン・ツールキットとして配布
- 次のコンポーネントを含む:
 - CUDA* コードの移行向け
インテル® DPC++ 互換性ツール
 - 高度なパフォーマンス解析および
デバッグツール



インテル® Advisor を使用してパフォーマンスを向上



GPU 計算解析向けのインテルの解析ツール

インテル® Advisor

オフロードのモデル化

- 効果の大きいオフロード候補を特定
- ボトルネックと主な制限要因の検出
- パフォーマンス、ヘッドルーム、ボトルネックをモデル化してハードウェアを入手する前にコードを準備

ルーブリック解析

- ハードウェアの制限に対するパフォーマンスのヘッドルームを確認
- ボトルネックと最も大きな効果が得られる最適化を特定してパフォーマンスの最適化戦略を決定
- 最適化の進捗状況を視覚化

フローグラフ・アナライザー

- CPU/GPU コードを視覚化して、CPU デバイス向けの推奨事項を取得

インテル® VTune™ プロファイラー

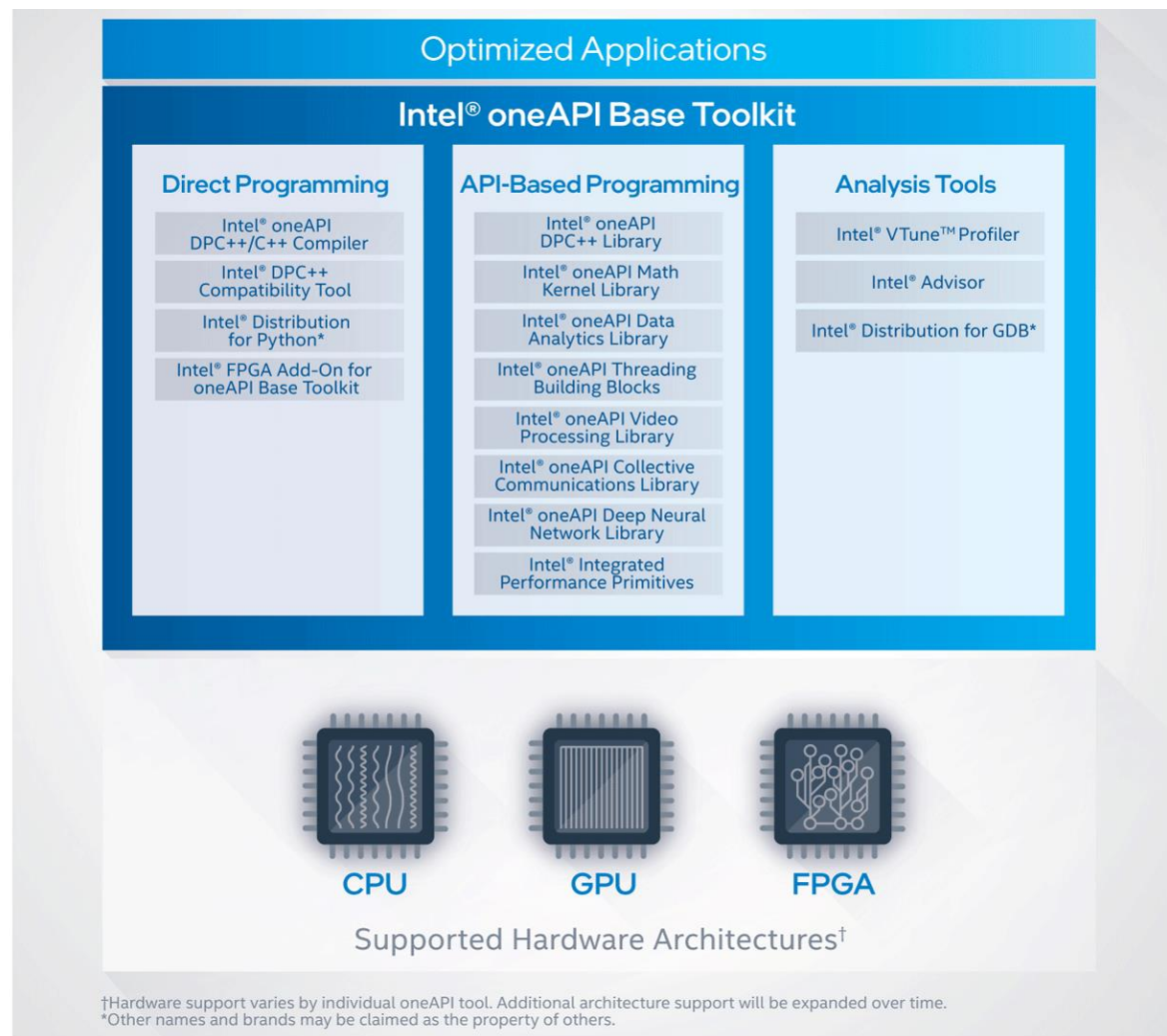
GPU 解析

- プラットフォームの各種 CPU と GPU コアでのコード実行を調査
- CPU と GPU アクティビティを関連付け
- アプリケーションが GPU 依存か、CPU 依存かを特定

GPU 計算/メディア・ホットスポット解析

- 最も時間のかかる GPU カーネルを解析し、GPU ハードウェア・メトリックを基に GPU 利用を特徴付け
- GPU コードのパフォーマンスをソース行レベルやカーネル・アセンブリー・レベルで解析

インテル® oneAPI ベース・ツールキット



インテル[®] Advisor

効率良いベクトル化、スレッド化、メモリー使用、アクセラレーターへのオフロード、フローグラフ・アルゴリズムを実現するコードを設計

ヘテロジニアス・ハードウェアを最大限に活用するツール インテル® Advisor



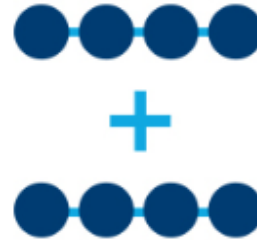
オフロードのモデル化

オフロード戦略を設計して
GPU 上でのパフォーマンスを
モデル化



ループライン解析

アプリケーションの
メモリーと計算を最適化



ベクトル化の最適化

より多くのベクトル並列処理を
行い、ベクトル化効率を向上



スレッドのプロトタイプ 生成

複数のスレッド設計を
モデル化、チューニング、テスト



ヘテロジニアス・ アルゴリズムの開発

データフローと依存関係
計算グラフを作成して解析

詳細: xlsoft.com/jp/products/intel/advisor/

インテル® Advisor

フローグラフ・アナライザー

フローグラフ・アナライザー ワークフロー

2 ステップのプロセス

1. フローグラフ・アナライザー・コレクター (FGT) でトレースを収集
 - FGT でアプリケーションを実行
 - `.graphml` ファイルと `.traceml` ファイルが生成される
2. フローグラフ・アナライザー GUI (FGA) でトレースを表示
 - `.graphml` ファイルを開く

フローグラフ・アナライザーの使用

コレクター

- fgtvars.sh を使用して必要なコレクターモジュールへのパスを設定

トレースの収集:

```
fgtrun.sh <application>
```

または

```
set FGT_ROOT to point to the <fga-install-dir>/fgt
```

```
$FGT_ROOT/linux/bin/fgtrun.sh <application>
```

生成されるファイル: <application>.graphml、<application>.traceml

コレクターを使用して
アプリケーションを実行し、
フローグラフを作成

フローグラフ・アナライザーについて

データ並列 C++ (DPC++) を使用したプログラミング

```
void vec_add(queue &q,
             const float A[],
             const float B[],
             float C[],
             const int size) {
    Timer s;
    // バッファを作成
    buffer<float,1> bufA(A, range<1>(VECTOR_SIZE));
    buffer<float,1> bufB(B, range<1>(VECTOR_SIZE));
    buffer<float,1> bufC(C, range<1>(VECTOR_SIZE));
    for(int i = 0; i < 5; ++i ) {
        q.submit([&](handler &cgh){
            float alpha = 2.0;
            auto Acc = bufA.get_access<access::mode::read>(cgh);
            auto Bcc = bufB.get_access<access::mode::read>(cgh);
            auto Ccc = bufC.get_access<access::mode::write>(cgh);
            cgh.parallel_for<class saxpy_kernel>(range<1>(size), [=](id<1> idx){
                Ccc[idx[0]] = alpha * Acc[idx[0]] + Bcc[idx[0]];
            });
        });
    }
    auto elapsed = s.Elapsed();
    std::cout << "Vec.Add took " << elapsed << "s\n";
    std::cout<<"Accessing host array outside SYCL scope:\n";
    for(int i = 0; i < VECTOR_SIZE; i+=1024) {
        std::cout << "C[" << i << "] = " << C[i] << "\n";
    }
}
```

```
int main(int argc, char **argv) {
    if (argc < 2) {
        std::cout << "Usage:- " << argv[0] << " [cpu, gpu]\n";
        return 1;
    }

    float A[VECTOR_SIZE], B[VECTOR_SIZE], C[VECTOR_SIZE];
    . . .
    if (std::string("cpu") == argv[1]) {
        cpu_selector device;
        queue q(device);
        vec_add(q, A, B, C, VECTOR_SIZE);
    } else if (std::string("gpu") == argv[1]) {
        gpu_selector device;
        queue q(device);
        vec_add(q, A, B, C, VECTOR_SIZE);
    }
    return 0;
}
```

ヘテロジニアス・コードのデバッグ フローグラフ・アナライザーを使用

The screenshot displays the Flow Graph Analyzer interface. The main window is divided into several sections:

- Hierarchical View:** Shows a tree structure of the code, with `iso3dfd_kernel_cpu` and `iso_3dfd_kernel_cpu_2` visible.
- Analysis View:** Displays a flow graph with nodes such as `memory_allocation_node`, `command_group_node`, `command_group_node`, `memory_transfer_node`, and `memory_deallocation_node`. A red box highlights a `command_group_node`.
- Output / Source View:** Shows the source code with a play button and a `Trace Views` section.
- Execution Trace:** A Gantt chart showing the execution of threads (Thread 0, Thread 1, Thread 2) over time (0.3s to 2.1s). Operations like `ciLinkProgram`, `ciSetKernelArguments`, and `tbb_parallel_for` are visible.
- Properties Panel:** Shows the properties of the selected node, including `Graph`, `Node`, and `Edge` properties.

Four callout boxes with arrows point to specific features:

- 全体的な状態** (Overall State) points to the Hierarchical View.
- グラフポロジ** (Graph Topology) points to the Analysis View.
- 実行トレース** (Execution Trace) points to the Execution Trace Gantt chart.
- プロパティ** (Properties) points to the Properties Panel.

非同期実行を可視化

```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue Q;

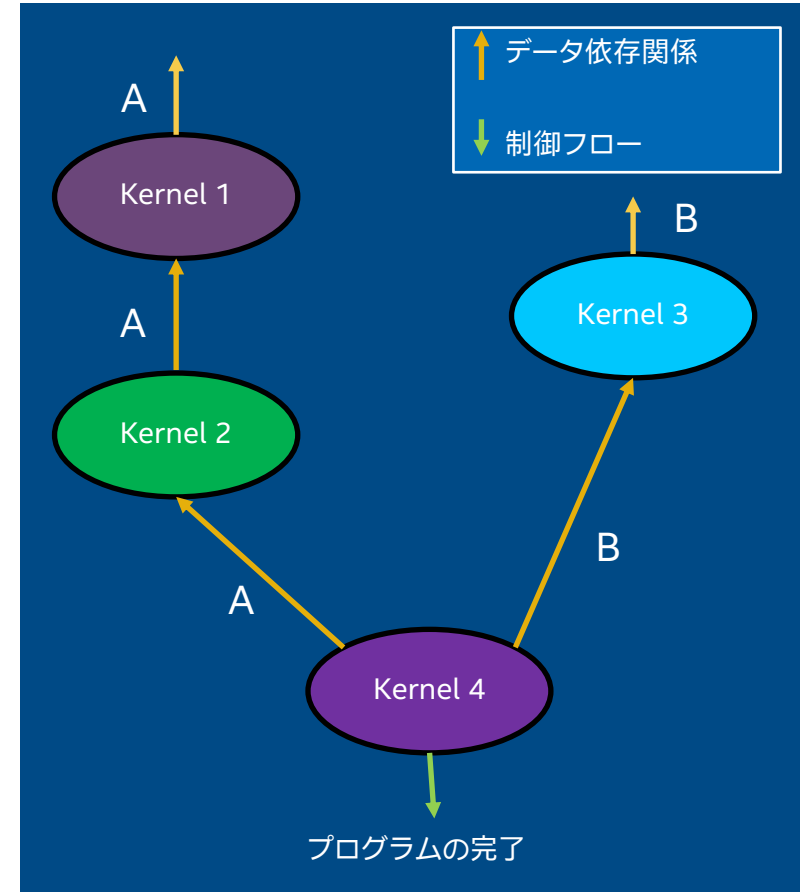
  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); }); } Kernel 1

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); }); } Kernel 2

  Q.submit([&](handler& h) {
    auto out = B.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); }); } Kernel 3

  Q.submit([&](handler& h) {
    auto in = A.get_access<access::mode::read>(h);
    auto inout =
      B.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      inout[idx] *= in[idx]; }); }); } Kernel 4
}
```

コンパイラーはデータを解決して
依存関係を制御

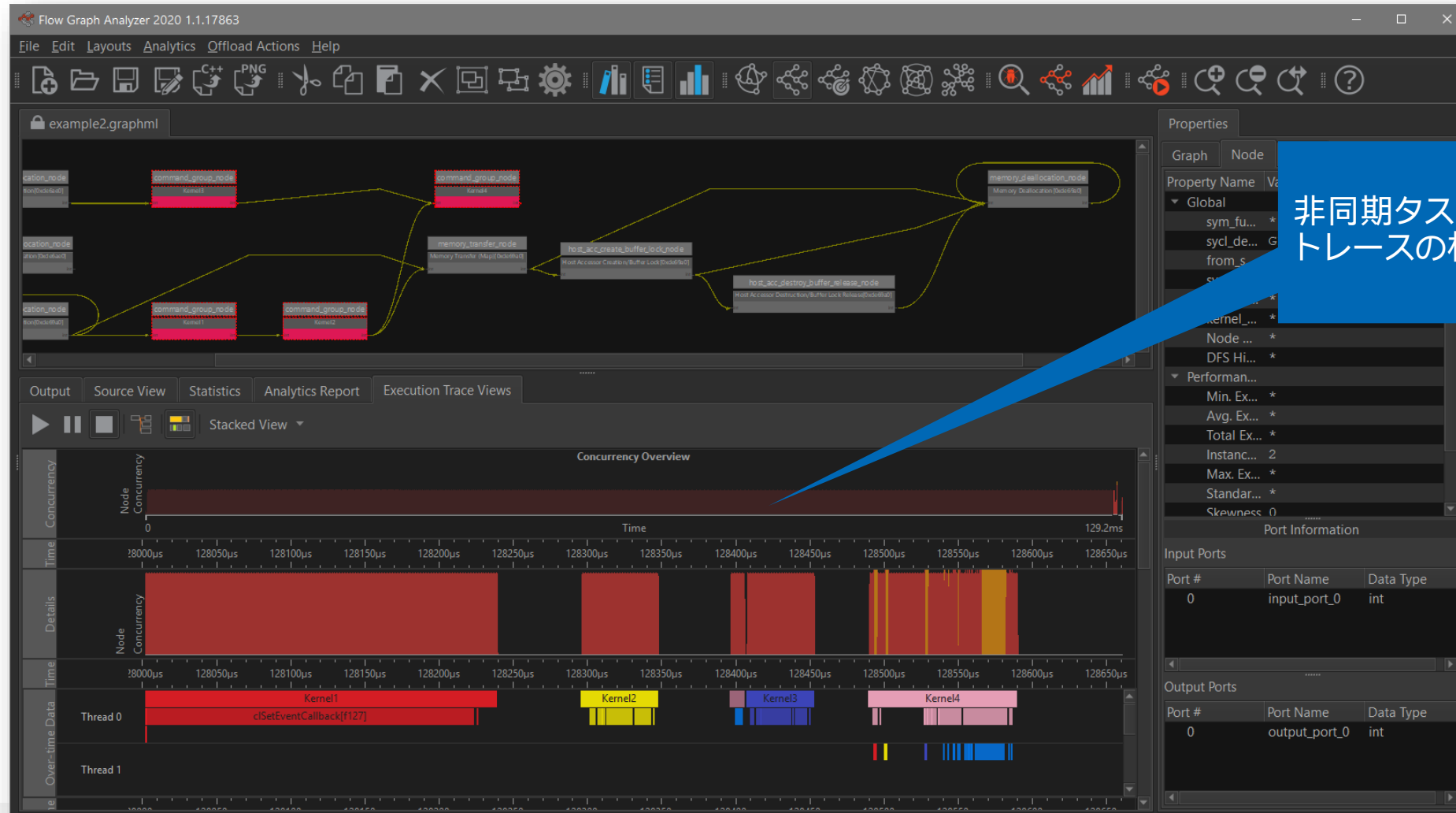


フローグラフ・アナライザー (FGA) を使用した解析

- バッファ生成時に使用されるホストポインターが `const` として宣言されている場合に余分なコピーを検出
- ループ内でキューが作成されているかどうかを検出 (毎回カーネルが再コンパイルされる)
- ループ内でホストアクセサーが使用されているかどうかを検出—ホストアクセサーが管理するデータ量によってはコストが大きくなる
- データ並列アルゴリズム (`parallel_for`) のスタートアップ・コストとインバランス・コスト。`parallel_for` がレンジを使用し、ツールがインバランスをレポートする場合、インバランスを解消する `nd_range` のブロックサイズを調査。必要に応じて `SYCL_BE=PI_OPENCL` を設定

フローグラフ・アナライザー

非依存のグラフノードが並列に実行しているか確認する



非同期タスクグラフと実行
トレースの相関性

フローグラフ・アナライザー ソースとの関連付け

The screenshot displays the Flow Graph Analyzer interface. The top window shows a flow graph with nodes such as 'command_group_node', 'memory_transfer_node', and 'to_it_acc_create_buffer_lock_node'. The bottom window shows the source code for a C++ program. The source code is as follows:

```
6 int main() {
7     auto R = range<1>{ num };
8     buffer<int> A{R}, B{R};
9
10    queue Q;
11    Q.submit([&](handler &h) {
12        auto out = A.get_access<access::mode::write>(h);
13        h.parallel_for<class Kernel1>(R, [=](id<1> idx) {
14            out[idx] = idx[0];
15        });
16    });
17
18    Q.submit([&](handler &h) {
19        auto out = A.get_access<access::mode::write>(h);
20        h.parallel_for<class Kernel2>(R, [=](id<1> idx) {
21            out[idx] = idx[0];
22        });
23    });
24
25    Q.submit([&](handler &h) {
26        auto out = B.get_access<access::mode::write>(h);
27        h.parallel_for<class Kernel3>(R, [=](id<1> idx) {
28            out[idx] = idx[0];
29        });
30    });
31 }
```

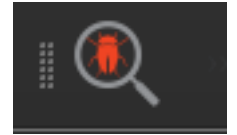
依存関係を
可視化

カーネルの
ソースコードを
表示

パフォーマンスの問題を検出

ルールチェッカーでパフォーマンスの問題を検出

- わずかなリファクタリングでもパフォーマンスが低下する可能性がある
- ルールチェック・エンジンは余分なコピーが発生するようなエラーをチェック



The screenshot displays a graph of memory operations. The graph shows a sequence of nodes: memory_allocation_node (0x7f3650), command_group_node (saxpy_kernel), memory_transfer_node (Memory Transfer (Copy) [0x7f3650]), and memory_deallocation_node (0x7f3650). Below this, there are three pairs of memory allocation and deallocation nodes. The middle pair, with addresses 0x7f3890, is highlighted in red, indicating a diagnostic issue. The diagnostic report below the graph shows a 'Buffer Copy Alert' for the memory allocation at 0x7f3890, with the information 'Buffer [0x7f3890]: Possibly copy of host pointer'. The report also shows a 'DPC++ Summary' for the graph 'g0' with performance metrics: Computation [1%], Compile/Build [98%], and Memory Operations [0%].

Severity	Diagnostic	Object	Name	Information
g0	Results(8)			
DPC++ Issues	Results(4)			
Summary	Results(2)			
●	Buffer Copy Alert	Node	Memory Allocation[0x7f3890]	Buffer [0x7f3890]: Possibly copy of host pointer
●	DPC++ Summary	Graph	g0	Computation [1%], Compile/Build[98%], Memory Operations[0%]
Schedul...	Results(0)			

並列処理の効率とは?

The screenshot displays the Flow Graph Analyzer interface. The top toolbar includes icons for file operations, analysis, and search. The main window shows a flow graph with nodes like 'Memory Allocation Node' and 'Command Group Node'. A search icon is highlighted with a red dashed box. The diagnostic report at the bottom is also highlighted with a red dashed box. A blue box on the right contains the text '並列アルゴリズムの効率' (Efficiency of Parallel Algorithms). The Intel logo is in the bottom right corner.

Diagnostic Report Data:

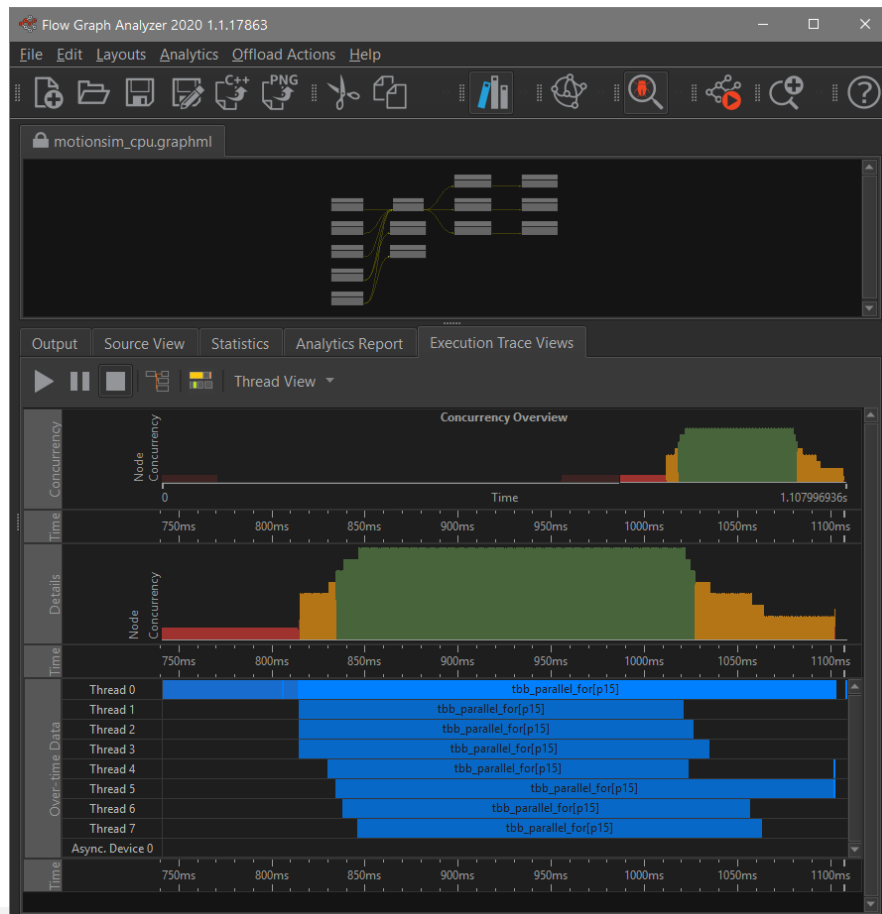
Severity	Diagnostic	Object	Name	Information
g0	Results(6)			
g0	DPC++ Issues Results(4)			
g0	Summary Results(2)			
g0	Host Pointer Access in ...	Node	kernel1	Host Access... Cycle [g0:n9]: Contains host pointer use in the cycle!
g0	DPC++ Summary	Graph	g0	Computation [56%], Compile/Build[43%], Memory Operations[0%]
g0	Schedule Results(0)			No data!
g0	Parallel ... Results(11)			
g0	Parallel Algorithm	Node	kernel1	[p51] DPC++ algorithm efficiency is ~95% with startup penalty ~2% and an imbalance penalty ~2%
g0	Parallel Algorithm	Node	kernel1	[p15] DPC++ algorithm efficiency is ~86% with startup penalty ~10% and an imbalance penalty ~2%
g0	Parallel Algorithm	Node	kernel1	[p27] DPC++ algorithm efficiency is ~90% with startup penalty ~4% and an imbalance penalty ~5%
g0	Parallel Algorithm	Node	kernel2	[p53] DPC++ algorithm efficiency is ~85% with startup penalty ~2% and an imbalance penalty ~12%
g0	Parallel Algorithm	Node	kernel2	[p17] DPC++ algorithm efficiency is ~86% with startup penalty ~2% and an imbalance penalty ~11%
g0	Parallel Algorithm	Node	kernel2	[p23] DPC++ algorithm efficiency is ~79% with startup penalty ~2% and an imbalance penalty ~17%
g0	Parallel Algorithm	Node	kernel2	[p29] DPC++ algorithm efficiency is ~15% with imbalance penalty ~84%
g0	Parallel Algorithm	Node	kernel2	[p35] DPC++ algorithm efficiency is ~80% with startup penalty ~2% and an imbalance penalty ~17%
g0	Parallel Algorithm	Node	kernel2	[p41] DPC++ algorithm efficiency is ~79% with startup penalty ~2% and an imbalance penalty ~17%
g0	Parallel Algorithm	Node	kernel3	[p19] DPC++ algorithm efficiency is ~100%
g0	Parallel Algorithm	Node	kernel3	[p67] DPC++ algorithm efficiency is ~100%
g0	Node St... Results(9)			
g0	Compute percentage	Node	kernel1	Instances[4], Computation [58%], Compile/Build[41%]
g0	Compute percentage	Node	Memory Tra...	Instances[8], Memory Operations [100%]
g0	Compute percentage	Node	Memory All...	Instances[1], Memory Operations [100%]
g0	Compute percentage	Node	kernel2	Instances[7], Computation [8%], Compile/Build[91%]
g0	Compute percentage	Node	Memory All...	Instances[2], Memory Operations [100%]

並列アルゴリズムの
効率

パフォーマンスの問題

スタートアップ・ペナルティとインバランス

Parallel ... Results(1)
Parallel Algo... Node Simulation [p15] DPC++ algorithm efficiency is ~79% with startup penalty ~3% and an imbalance penalty ~16%



このアルゴリズムのスタートアップ・ペナルティは 79%!!

インバランス・ペナルティは 16%!

まとめ

- データセントリックな計算の多様なワークロードにより、CPU、GPU、FPGA、および AI アクセラレーターを含むさまざまな計算アーキテクチャーに対するニーズが高まっている
- oneAPI はインテルの CPU とアクセラレーターのプログラミングを統合して簡素化し、開発者に生産性と完全なネイティブ言語のパフォーマンスを提供
- インテル® Advisor の GPU をサポートする新機能:
 - オフロードのモデル化
 - GPU ルーフライン
 - フローグラフ・アナライザー

関連情報

- [インテル® oneAPI](#)
- [インテル® Advisor](#)
- [インテル® Advisor クックブック](#)

法務上の注意書きと最適化に関する注意事項

インテルのテクノロジーを使用するには、対応したハードウェア、ソフトウェア、またはサービスの有効化が必要となる場合があります。詳細については、OEM または販売店にお問い合わせいただくか、<http://www.intel.co.jp/> を参照してください。

実際の費用と結果は異なる場合があります。

インテルは、サードパーティーのデータについて管理や監査を行っていません。ほかの情報も参考にして、正確かどうかを評価してください。

最適化に関する注意事項: インテル® コンパイラーでは、インテル® マイクロプロセッサに限定されない最適化に関して、他社製マイクロプロセッサ用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサ依存の最適化は、インテル® マイクロプロセッサでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサ用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。注意事項の改訂 #20110804

<https://software.intel.com/content/www/us/en/develop/articles/optimization-notice.html#opt-jp>

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサ用に最適化されていることがあります。

SYSMARK* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。構成の詳細は、補足資料を参照してください。性能やベンチマーク結果について、さらに詳しい情報をお知りになりたい場合は、<http://www.intel.com/benchmarks> (英語) を参照してください。

性能の測定結果はシステム構成の日付時点のテストに基づいています。また、現在公開中のすべてのセキュリティー・アップデートが適用されているとは限りません。詳細は、システム構成を参照してください。絶対的なセキュリティーを提供できる製品またはコンポーネントはありません。

本資料は、(明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず) いかなる知的財産権のライセンスも許諾するものではありません。

インテルは、明示されているか否かにかかわらず、いかなる保証もいたしません。ここにいう保証には、商品適格性、特定目的への適合性、および非侵害性の黙示の保証、ならびに履行の過程、取引の過程、または取引での使用から生じるあらゆる保証を含みますが、これらに限定されるわけではありません。

© Intel Corporation. Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

intel®