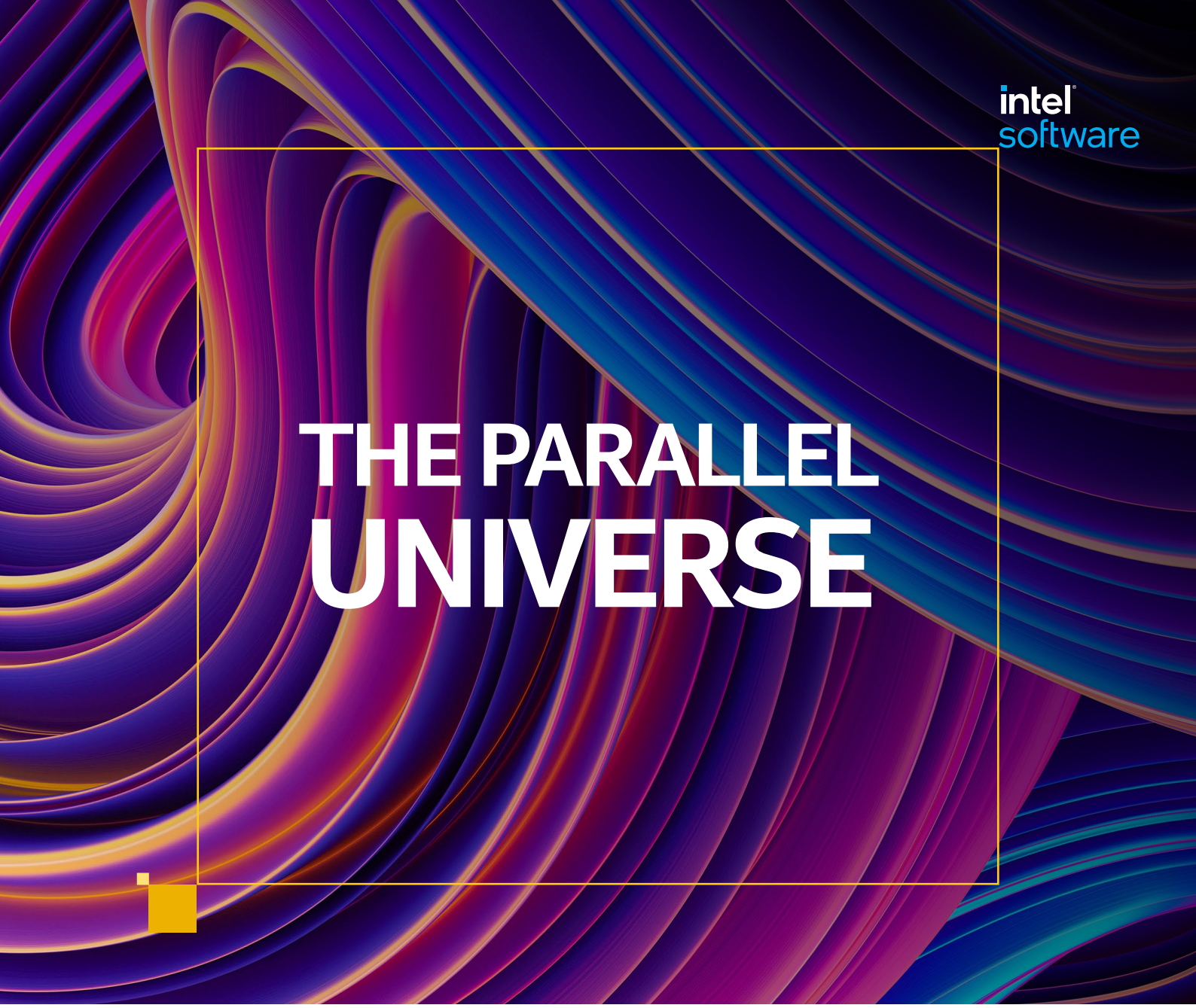


インテルの CPU と GPU 向けの LLVM と GCC のベクトル化

OpenMP* を使用した効率良いヘテロジニアス並列プログラミング

ArrayFire* と oneAPI、各種ライブラリー、OpenCL* の相互運用性

Issue
47
2022

目次

編集者からのメッセージ 3

注目記事

インテルの CPU と GPU 向けの LLVM と GCC のベクトル化 5
最新のコンパイラーで SIMD サポートが急速に進化

OpenMP* を使用した効率良いヘテロジニアス並列プログラミング 17
CPU と GPU を同時に動作させるベスト・プラクティス

ArrayFire* と oneAPI、各種ライブラリー、OpenCL* の相互運用性 22
oneAPI を活用してコードの書き換えを回避

oneAPI レベルゼロ・インターフェイスの使用 29
レベルゼロ API の概要紹介

Habana* Gaudi* 上での MLPerf* トレーニングの SigOpt を使用したハイパーパラメーター最適化 38
AI トレーニングでより高い精度とより高速な収束を実現

Modin で pandas* のワークフローをスケーリング 50
書き換え不要のスケーラブルなデータ・アナリティクス

Ray から Chronos へ 55
Ray 上に BigDL でエンドツーエンドの AI ユースケースを構築

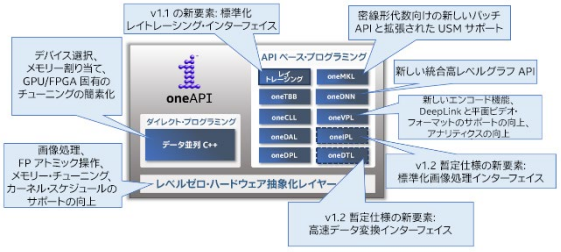
コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

編集者からのメッセージ

Henry A. Gabb インテル コーポレーション シニア主席エンジニア
HPC と並列コンピューティング分野で長年の経験があり、並列プログラミングに関する記事を多数出版しています。『Developing Multithreaded Applications : A Platform Consistent Approach』の編集者 / 共著者で、インテルと Microsoft による Universal Parallel Computing Research Centers のプログラム・マネージャーを務めました。

勢いを増す oneAPI

『The Parallel Universe』の編集者になった直後の「編集者からのメッセージ」では、来るべきヘテロジニアス並列コンピューティング時代に不安を感じていたことを述べました。あれから 5 年経った今は、oneAPI（英語）のお陰でヘテロジニアス並列コンピューティングの未来について楽観視しています。インテルコーポレーションの副社長兼ソフトウェア開発製品事業本部長の Sanjiv Shah は最近のブログ「oneAPI の進歩に感謝」（英語）において、同様に楽観的な意見を述べています。新しい oneAPI v1.1 仕様が公開され、v1.2 暫定仕様では多くの新機能が追加されています。2021 年は新たに 11 の oneAPI 研究拠点が世界各地で発足しました。そして、oneAPI はまたもや HPCwire Reader' s Choice Award に選ばれ、「2021 年ベスト HPC プログラミング・ツール / テクノロジー」（英語）を受賞しました。2022 年にはさらなる飛躍が期待されます。



本号の注目記事「インテルの CPU と GPU 向けの LLVM と GCC のベクトル化」では、最新のコンパイラーにおける SIMD（Single Instruction, Multiple Data）サポートの進化を解説し、自動ベクトル化、プログラマーによるベクトル化、およびデータ並列ライブラリー・アプローチの例を示します。

「**OpenMP* を使用した効率良いヘテロジニアス並列プログラミング**」では、CPU と GPU を同時に動作させるベスト・プラクティスを紹介します。標準の OpenMP* ディレクティブを使用して、非同期のヘテロジニアス並列処理を表現するためのアドバイスとコード例を提供します。

関心の分離（SoC）（英語）の観点から考えると、私自身はパフォーマンス・チューニング・エンジニアというよりも、ドメイン・サイエンティスト寄りで、ハードウェアの詳細を隠しつつパフォーマンスを発揮するプログラミングの抽象化に興味があります。最近は、ArrayFire*（英語）というヘテロジニアス並列ライブラリーを実験的に使っており、生産性とパフォーマンスの両面で良い結果が得られています。私の体験談は将来の号で紹介する予定ですが、本号では ArrayFire* のソフトウェア・エンジニアである Stefan Yurkevitch 氏が「**ArrayFire* と oneAPI、各種ライブラリー、OpenCL* との相互運用性**」について述べています。

「**oneAPI レベルゼロ・インターフェイスの使用**」では、ヘテロジニアス並列処理向けの高レベルのソフトウェア抽象化から、低レベルのハードウェア抽象化へと進みます。

本号の最後を締めくくるのはデータサイエンスに関する 3 つの記事です。1 つ目の記事は、モデルの精度を向上しつつ、「**Habana* Gaudi* 上での MLPerf* トレーニングの SigOpt を使用したハイパーパラメーター最適化**」を効率良く行う方法を紹介します。次の記事は、コードを書き換えずに「**Modin で pandas* のワークフローをスケーリング**」する方法を説明します。そして、最後の記事「**Ray から Chronos へ**」では、Ray 上に BigDL を使用してスケーラブルでエンドツーエンドの AI ワークフローを構築する方法を紹介します。

コードの現代化、ビジュアル・コンピューティング、データセンターとクラウド・コンピューティング、データサイエンス、システムと IoT 開発、oneAPI を利用したヘテロジニアス並列コンピューティング向けのインテル・ソリューションの詳細は、Tech.Decoded（英語）を参照してください。

Henry A. Gabb
2022 年 1 月

インテルの CPU と GPU 向けの LLVM と GCC のベクトル化

最新のコンパイラーで SIMD サポートが急速に進化

Xinmin Tian インテル コーポレーション シニア主席エンジニア
Hideki Saito インテル コーポレーション 主席エンジニア
Hongtao Liu インテル コーポレーション コンパイラー・エンジニア
James Reinders インテル コーポレーション oneAPI エバンジェリスト兼 The Parallel Universe 名誉編集長

近年の CPU や GPU のコアは、SIMD（Single Instruction, Multiple Data）実行ユニットを利用して、より高いパフォーマンスと電力効率を実現しています。SIMD ハードウェアは、インテル® SSE、インテル® AVX、インテル® AVX2、インテル® AVX-512、およびインテル® X^{e} アーキテクチャー（第 12 世代）ISA などの命令を介して利用されます。これらを直接使用することも可能ですが、低レベルのため移植性が著しく制限され、ほとんどのプロジェクトでは推奨されません。

ここでは、移植性が高く、使いやすいインターフェイスをプログラマーに提供するため、自動ベクトル化、言語構造やプラグマなどのヒントを使用した SIMD ベクトル化、および SIMD データ並列ライブラリー・アプローチの 3 つの手法を検討します。これらの手法の概要を説明し、コード例を使用して LLVM および GCC コンパイラーにおける SIMD ベクトル化の進化を紹介します。また、インテル® Xeon® プロセッサーおよびインテル® X^{e} アーキテクチャー GPU で最適なパフォーマンスを実現するため、LLVM および GCC コンパイラーにおける 2 つのベクトル化手法を検討します。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

LLVM と GCC の拡張

インテルは、LLVM と GCC コンパイラーの両方のベクトル化機能を強化することを目標としており、オープンソースへの貢献は重要な設計上の考慮事項でした。VPlan ベクトライザーと関連する VectorABI は、LLVM と GCC[13] の両方のオプティマイザーに統合できるように設計されています。

VPlan ベクトライザーのフレームワークは、LLVM トランクに統合される可能性があります。インテルの VectorABI[12] は公開されており、LLVM および GCC コミュニティーによって関数のベクトル化に利用されています。VPlan ベクトライザーは、インテル® Xeon® プロセッサー用のインテル® コンパイラーが以前提供していた結果を上回っています。

SIMD の利用

近年の CPU は SIMD 実行をサポートしています。SIMD とは、1 つの命令で複数のデータ要素を並列に実行するハードウェア機能です。制御フローが似ており（ベクトルの発散が少ない）、メモリーに依存しない複数のデータを一度に操作する場合に有効です。残念ながら、SIMD ISA を直接利用するプログラムを記述することは容易ではなく、移植性にも限界があります。この状況を改善するため、自動ベクトル化、ヒントや言語構造を使用したプログラマーによる SIMD ベクトル化、および C++ SIMD データ並列ライブラリーの利用という 3 つのアプローチを説明します。

自動ベクトル化

データと制御の依存関係解析を自動的に行い、組込みコストモデルに基づいてスカラープログラムをベクトル形式に変換することを自動ベクトル化[4][5] と呼びます。この方法は生産性や移植性に優れており、プログラマーにとって魅力的ですが、ループ境界やメモリー・アクセス・パターンなど、コンパイル時に未知な部分があるため、自動ベクトル化は必ずしも最適なコードを生成するわけではありません。

プログラマーによる SIMD ベクトル化

OpenMP*（バージョン 4.0 以降）には、ベクトルレベルの並列処理をサポートする SIMD 構文があります[7]。これらの構文は、標準化されたベクトル化構文のセットを提供するため、プログラマーは移植性のないベンダー固有の組込み関数やディレクティブを使用する必要がありません[6]。さらに、これらの構文はコンパイラーにコードの構造に関する追加のヒントを提供し、並列化とうまく融合した、より優れたベクトル化を可能にします[5]。

C++ SIMD データ並列ライブラリー

ISO C++ にはデータ並列ライブラリーに関する提案が提出されており[3]、その目的は、SIMD レジスターや命令、共通の命令デコーダーで実行される実行ユニットなどのデータ並列実行リソースによる高速化をサポートすることです。これらの実行リソースが利用できない場合、インターフェイスはシーケンシャル実行への透過的なフォールバックをサポートします。**図 1** に、C++ SIMD データ並列ライブラリーを使用した SIMD `memcpy` の例を示します。この例をコンパイルして、core-avx512 用の LLVM ベクトル中間表現（IR）とバイナリーを生成できます。

```
namespace stdsimd = std::experimental;

void simd_memcpy(
     stdsimd::native_simd<float> x,
     stdsimd::native_simd<float> y,
     void *p)
{
   auto cmp = x < y;
   memcpy(p, &cmp, cmp.size()*4);
}
```

```
define void @_Z11simd_memcpy_Pv(<16 x float> %x.coerce,
                                <16 x float> %y.coerce, i8*
                                nocapture %p)
{
entry:
  %0 = fcmp fast olt <16 x float> %x.coerce, %y.coerce
  %cmp.sroa.0.sroa.0.0.p.sroa_cast = bitcast i8* %p to <16 x i1>*
  store <16 x i1> %0, <16 x i1>* %cmp.sroa.0.sroa.0.0.p.sroa_cast
  ret void
}
```

図 1. C++ SIMD データ並列ライブラリーの例

SIMD ベクトル化は、どのベクトル化手法を使用して SIMD コードを生成するかにかかわらず、最新の CPU および GPU 上で計算集約型ワークロードの最適なパフォーマンスを実現するために非常に重要です。次のセクションでは、いくつかのコード例とともに、CPU と GPU に関する最近の LLVM SIMD ベクトル化の進歩を紹介します。

LLVM VPlan ベクトル化

VPlan ベクトライザー

LLVM ベースのインテル® コンパイラーは、インテル® コンパイラー・クラシックに搭載されているベクトライザーと同等以上のパフォーマンスを目指して新たに設計されたループ・ベクトライザーを導入しています。この新しいベクトライザーは、LLVM コミュニティーのループ・ベクトライザー（通称 LV）と区別するため、主要な内部データ構造である VPlan（ベクトル化プラン）の名前から、しばしば VPlan ベクトライザーと呼ばれます。LORE と RAJAPerf の実験によると、LLVM ベースのインテル® コンパイラーは、HPC アプリケーションから抽出したさまざまな計算カーネルに対して、インテル® コンパイラー・クラシックと同等またはそれ以上のパフォーマンスのコードを生成できることが示されています[9]。この記事の執筆時点で、LLVM ベースのインテル® コンパイラーでは、`-O2` 以上の最適化と `-x`（Windows* では `/Qx`）ターゲットオプションを指定すると、VPlan ベクトライザーによる自動ベクトル化が有効になります。`-x` オプションを指定しないと、コミュニティーのループ・ベクトライザーが使用されます。VPlan ベクトライザーは、`-qopenmp`（Windows* では `/Qopenmp`）オプションを指定してインテルの OpenMP* 実装を有効にすると、OpenMP* SIMD について `-O0` 以上で有効になります。この記事の執筆時点では、よく使用される OpenMP* 4.5 の SIMD 機能の多くが機能し、パフォーマンスを発揮しています。インテルは、今後も最新の OpenMP* 5.2 の SIMD 機能をサポートするべく取り組んでいきます。

図 2 は、シンプルな外部ループ（左列）をインテル® コンパイラー・クラシック（`icc`、中央列）と LLVM ベースのインテル® コンパイラー（`icx`、右列）でベクトル化したものです。インテル® コンパイラー・クラシックで生成された ASM コードは全体的に簡潔で分かりやすいですが、LLVM ベースのインテル® コンパイラーのほうが、`%k1` マスクレジスターでの内部ループ実行条件の処理が優れているため、内側の `while` ループ（`icx` の基本ブロック `.LBB0_7` と `icc` の `..B1.7`）で明らかに優れた ASM コードを生成しています。

```
void foo(int N, float *a,
         float *b, float *c){
  #pragma omp simd
  for (int i=0;i<N;i++){
    float x = a[i];
    float y = b[i];
    while(x>y){
      x = x*x;
    }
    c[i] = x;
  }
}
```

```
icc -O2 -qopenmp-simd -xCORE-
AVX512 -c -S -unroll0

..B1.5:
 vmovups   (%rsi,%r8,4), %ymm1
 vmovups   (%rdx,%r8,4), %ymm0
 vcmpps    $14, %ymm0, %ymm1, %k1
 kortestw %k1, %k1
 je        ..B1.9
..B1.6:
 kmovw      %k1, %k0
..B1.7:
 kandw      %k0, %k1, %k2
 vmulps    %ymm1, %ymm1, %ymm1{%k2}
 vcmpps    $14, %ymm0, %ymm1, %k3
 kandw     %k3, %k2, %k4
 kandw     %k0, %k4, %k0
jne       ..B1.7
..B1.9:
 addl      $8, %r9d
 vmovups   %ymm1, (%rcx,%r8,4)
 addq      $8, %r8
 cmpl      %eax, %r9d
 jb        ..B1.5
```

```
icx -O2 -qopenmp-simd -xCORE-AVX512 -c
-S -unroll0

 jmp      .LBB0_4
.LBB0_5:
 vxorps   %xmm2, %xmm2, %xmm2
.LBB0_8:
 vcmpltps  %ymm0, %ymm1, %k1
 vmovaps %ymm2, %ymm0 {%k1}
 vmovups %ymm0, (%rcx,%rax,4)
 addq     $8, %rax
 cmpq     %rdi, %rax
 jae      .LBB0_9
.LBB0_4:
 vmovups (%rsi,%rax,4), %ymm0
 vmovups (%rdx,%rax,4), %ymm1
 vcmpltps  %ymm0, %ymm1, %k0
 kortestb  %k0, %k0
 je       .LBB0_5
# %bb.6:
 vmovaps %ymm0, %ymm3
 kmovq    %k0, %k1
.LBB0_7:
 vmulps   %ymm3, %ymm3, %ymm3
 vmovaps %ymm3, %ymm2 {%k1}
 vcmpltps  %ymm3, %ymm1, %k1 {%k1}
 ktestb   %k0, %k1
 jne      .LBB0_7
 jmp      .LBB0_8
```

図 2. VPlan ベクトライザーを使用した外部ループのベクトル化

カーネルと関数のベクトル化

LLVM ベースのインテル® コンパイラーは、VPlan ベクトライザー[5][10] を利用して、DPC++/OpenCL* カーネルのベクトル化と OpenMP* 関数のベクトル化を実装しています。これは、関数のベクトル化の問題をループのベクトル化の問題に置き換えることで実現されます。ループのベクトル化を最適化する実装のほとんどは、カーネル / 関数のベクトル化にも利用できることが期待できます。

図 3 は、OpenMP* の `declare simd` ディレクティブ形式で表現された等価なベクトル化されたコードです[7]。8 ウェイのマスクを使用しないでベクトル化されたインテル® AVX-512 ベクトルバリアント関数（`_ZGVcN8luuu_bar`）を示しています。基本ブロックのレイアウトが異なり、コンパイラーは関数 `bar` の「8 つのインスタンス」をベクトル化していることを知っているため、外部ループ制御フローは存在しませんが、ASM コードの残りの部分は、ループベクトル化の例で `icx` が生成した ASM コード（**図 2**）と驚くほど似ています。これは、コンパイラーが関数本体の周りに 8 反復のループを挿入して、同じ VPlan ベクトライザーでベクトル化されているためです。

```
#pragma omp declare simd \
        linear(i) uniform(a,b,c)
void bar(int i, float *a, float *b, float *c){
  float x = a[i];
  float y = b[i];
  while(x>y){
    x = x*x;
  }
  c[i] = x;
}
```

```
icx -O2 -qopenmp-simd -xCORE-AVX512 -c -S -unroll0

_ZGVcN8luuu_bar:
 movslq   %edi, %rax
 vmovups  (%rsi,%rax,4), %ymm0
 vmovups  (%rdx,%rax,4), %ymm1
 vcmpltps %ymm0, %ymm1, %k1
 kortestb %k1, %k1
 je       .LBB3_1
# %bb.2:
 vcmpltps %ymm0, %ymm1, %k0
 vmovaps  %ymm0, %ymm3
.LBB3_3:
 vmulps   %ymm3, %ymm3, %ymm3
 vmovaps  %ymm3, %ymm2 {%k1}
 vcmpltps %ymm3, %ymm1, %k1 {%k1}
 ktestb   %k0, %k1
 jne      .LBB3_3
 jmp      .LBB3_4
.LBB3_1:
 vxorps   %xmm2, %xmm2, %xmm2
.LBB3_4:
 vcmpltps %ymm0, %ymm1, %k1
 vmovaps  %ymm2, %ymm0 {%k1}
 vmovups  %ymm0, (%rcx,%rax,4)
 vzeroupper
 retq
```

図 3. VPlan ベクトライザーを使用した関数のベクトル化の例

新しい ISA サポート

LLVM コンパイラー・フレームワーク上にベクトライザーを実装するメリットの 1 つは、優れたベクトルデータ型サポートを利用できることです。インテル® AVX-512 の FP16[11] が登場した際、ASM/OBJ コード生成サポートが追加されると同時に、ベクトライザーはその利点を生かすことができ、ベクトライザー開発者に嬉しい驚きを与えてくれました。**図 4** は、単純な FP16 のベクトル化の例です。

```
void foo(int N, __fp16 *a, __fp16 *b, __fp16 *c)
{
  #pragma omp simd
  for (int i=0;i<N;i++)
  {
    c[i] = a[i]+b[i];
  }
}
```

```
icx -qopenmp-simd -O2 -xsapphirerapids -c -S -unroll0

.LBB0_3:
 vmovups (%rdx,%rax,2), %ymm0
 vaddph  (%rsi,%rax,2), %ymm0, %ymm0
 vmovups %ymm0, (%rcx,%rax,2)
 addq    $16, %rax
 cmpq    %rdi, %rax
 jb      .LBB0_3
```

図 4. VPlan ベクトライザーを使用した FP16 のベクトル化の例（I）

新しく導入された命令セットに対して、すべてのオプティマイザーがすぐにうまく機能するわけではないことに注意してください。**図 5** は、**図 2** と同じ例ですが、FP16 データ型を使用しています。`vmulph` 命令を使用する最内ループは、現在、**図 2** や**図 3** ほどうまく最適化されていません。今後のリリースでは、このような問題の掘り起こしと改善を進めていく予定です。

```
void foo(int N, __fp16 *a, __fp16 *b, __fp16 *c)
{
  #pragma omp simd
  for (int i=0;i<N;i++)
  {
    __fp16 x = a[i];
    __fp16 y = b[i];
    while(x>y)
    {
      x = x*x;
    }
    c[i] = x;
  }
}
```

```
icx -qopenmp-simd -O2 -xsapphirerapids -c -S -unroll0

 jmp     .LBB0_4
.LBB0_5:
 vpxor   %xmm2, %xmm2, %xmm2
.LBB0_12:
 vcmpltph        %ymm0, %ymm1, %k1
 vmovdqu16       %ymm2, %ymm0 {%k1}
 vmovdqu %ymm0, (%rcx,%rax,2)
 addq    $16, %rax
 cmpq    %rdi, %rax
 jae     .LBB0_13
.LBB0_4:
 vmovups (%rsi,%rax,2), %ymm0
 vmovups (%rdx,%rax,2), %ymm1
 vcmpltph        %ymm0, %ymm1, %k0
 kortestw        %k0, %k0
 je      .LBB0_5
# %bb.6:
 vmovaps %ymm0, %ymm3
 kmovq   %k0, %k1
 jmp     .LBB0_7
.LBB0_11:
 vmovdqu16       %ymm3, %ymm2 {%k1}
 kandw   %k1, %k2, %k1
 ktestw  %k0, %k1
 je      .LBB0_12
.LBB0_7:
 ktestw  %k1, %k0
 vmulph  %ymm3, %ymm3, %ymm4
 vxorps  %xmm3, %xmm3, %xmm3
 je      .LBB0_9
# %bb.8:
 vmovaps %ymm4, %ymm3
.LBB0_9:
 kxorw   %k0, %k0, %k2
 je      .LBB0_11
# %bb.10:
 vcmpltph %ymm4, %ymm1, %k2
 jmp     .LBB0_11
```

図 5. VPlan ベクトライザーを使用した FP16 のベクトル化の例（II）

GCC12 の自動ベクトル化の強化

このセクションでは、インテル® Xeon Phi™ プロセッサー用に開発された GCC ベクトル化フレームワークを基に、GCC12 コンパイラーのインテル® AVX-512/ インテル® AVX-512 VNNI サポート用に最近開発したいくつかの自動ベクトル化の強化について説明します。

- GCC12 の自動ベクトル化は、「cheap」コストモデルを使用する -O2 でデフォルトで有効になります。このモデルは、ベクトル化可能なスカラーループのトリップカウントがハードウェア・ベクトル長の倍数であり、コードサイズの増加が観測されなければループをベクトル化します。例えば、**図 6** は、インテル® SSE4.2 を使用した GCC -O2 自動ベクトル化の例です。一方、-O3 でのループベクトル化のデフォルトのコストモデルは、ベクトル化されたコードパスがパフォーマンス向上を達成するかどうかを判断するチェックポイントを増やす、「dynamic」モデルを採用しています。

```
void ArrayAdd(int* __restrict a, int* b)
{
   for (int i = 0; i != 32; i++)
      a[i] += b[i];
}
```

```
ArrayAdd:
  xorl    %eax, %eax
.L2:
  movdqu  (%rdi,%rax), %xmm0
  movdqu  (%rsi,%rax), %xmm1
  paddd   %xmm1, %xmm0
  movups  %xmm0, (%rdi,%rax)
  addq    $16, %rax
  cmpq    $128, %rax
  jne     .L2
  ret
```

図 6. GCC（-O2）の自動ベクトル化の例

- GCC の `_Float16` 型のベクトル化を有効にして、対応するインテル® AVX-512 の `FP16` 命令を生成します。`float/double` 型と同様の SIMD 命令に加え、ベクトライザーは複素数の `_Float16` 型のベクトル化もサポートしています。**図 7** は、3 つの配列に対して共役複素数乗算と累算を行う例で、ベクトル化可能なループは `vfcmaddcph` 命令を生成するように最適化できます。

```
#include<complex.h>

void fmaconj (_Complex _Float16 a[restrict 16],
              _Complex _Float16 b[restrict 16],
              _Complex _Float16 c[restrict 16])
{
   for (int i = 0; i < 16; i++)
      c[i] += a[i] * ~b[i];
}
```

```
fmaconj:
  vmovdqu16   (%rdx), %zmm1
  vmovdqu16   (%rsi), %zmm0
  vfcmaddcph  (%rdi), %zmm1, %zmm0
  vmovdqu16   %zmm0, (%rdx)
  vzeroupper
  ret
```

図 7. インテル® AVX-512 の FP16 vfcmaddcph 命令を使用した GCC 自動ベクトル化

- GCC の自動ベクトル化は、インテル® AVX/ インテル® AVX-512 VNNI 命令の生成をトリガーするドットプラス・イディオムなどのイディオム認識を行うように強化されています。**図 8** は、コンパイラーが `vpdpbusd` 命令に加えて、総和リダクションを生成している様子を示しています。

```
int usdot_prod_qi (unsigned char * restrict a,
                   char *restrict b, int c, int n)
{
  for (int i = 0; i < 32; i++)
  {
    c += ((int) a[i] * (int) b[i]);
  }
  return c;
}
```

```
usdot_prod_qi:
  vmovdqu  (%rdi), %ymm0
  vpxor    %xmm1, %xmm1, %xmm1
  vpdpbusd (%rsi), %ymm0, %ymm1
  vextracti128 $0x1, %ymm1, %xmm0
  vpaddd   %xmm1, %xmm0, %xmm0
  vpsrldq  $8, %xmm0, %xmm1
  vpaddd   %xmm1, %xmm0, %xmm0
  vpsrldq  $4, %xmm0, %xmm1
  vpaddd   %xmm1, %xmm0, %xmm0
  vmovd    %xmm0, %eax
  addl     %edx, %eax
  vzeroupper
  ret
```

図 8. GCC 自動ベクトル化のインテル® AVX-512 VNNI イディオム認識

前述の 3 つの GCC 自動ベクトル化の強化に加え、冗長なゼロ拡張と切り捨てがベクトライザーでも認識される場合、`vpopcnt[b,w,d,q]` 命令を利用するように GCC を改善しました。これらの改善により、インテル® Xeon® スケーラブル・プロセッサー向けの GCC 自動ベクトル化機能が大幅に拡張されています。

インテル® GPU 向けの SIMD ベクトル化

設計理由

インテル® X^e アーキテクチャーを採用したインテル® GPU は、OpenCL* SIMT（Single Thread Multiple Data）と SIMD の両方をサポートするように設計されています。このセクションでは、LLVM VPlan ベクトライザーを有効にして、X^e GPU の SIMD ISA を利用して OpenMP* SIMD ループを SIMD コードに変換する方法を説明します。この設計と実装の背景には、2 つの理由があります。

- CPU 向けに OpenMP* SIMD 構文を使用して記述された既存の C++ および Fortran アプリケーションを、OpenMP* オフロードと SIMD を利用する X^e GPU 向けに比較的スムーズに移行する方法を提供します。
- X^e GPU 向けの SIMD ISA をフル活用するため、OpenMP* オフロード領域でさまざまな明示的 SIMD スキームを使用して、柔軟に SIMD ループをベクトル化します。

インテル® GPU 向けの OpenMP* を使用するインテル® C++/Fortran コンパイラーの SIMD ベクトル化は、ハードウェアの機能を活用するように設計されており、きめ細かいレジスター管理、SIMD サイズ制御、クロスレーン・データ共有を可能にします。

高レベルの SIMD ベクトル化フレームワーク

図 9 は、インテル® GPU のデバイス・コンパイル・パスに実装された SIMD ベクトル化フレームワークの概要で、oneAPI コンパイラーに搭載されている CPU 向けの LLVM VPlan ベクトライザー[4][5] をフル活用しています。VPlan ベクトライザー（ボックス 4）は、言語のフロントエンド（ボックス 1）とミドルエンドの最適化（ボックス 2 と 3）から LLVM スカラー IR を受け取り、X^e GPU 操作用に定義された GPU ターゲット組込み関数への下位変換（ボックス 5）と LLVM ベクトル IR 生成を実行します。その後、インターフェイス IR として SPIR-V* を使用して、GPU ベクトル・バックエンド・コンパイラーに GPU 用の LLVM ベクトル IR を渡します（ボックス 6 と 7）[8]。

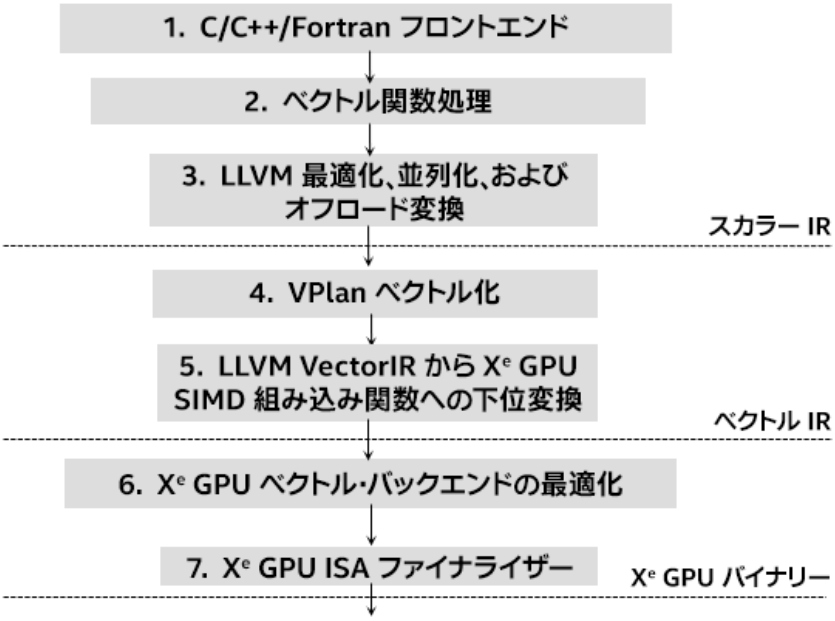


図 9. デバイスコンパイルの SIMD ベクトル化フレームワーク

これらの GPU 固有の組込み関数を中心に、一連の明示的な SIMD 固有の最適化および変換（ボックス 6）が開発されています。プログラマーは OpenMP* プログラミング API を介してループベクトル化やベクトル長の選択を制御し、コンパイラーのベクトル・バックエンド（ボックス 6 と 7）は、プログラマーの注釈に基づいてさまざまなコンパイラーによる最適化のトレードオフを達成するよう努めます。コンパイラー・ミドルエンドで生成される OpenMP* の明示的な SIMD カーネルは、インテル ® GPU の OpenCL* ランタイム[1] や oneAPI レベルゼロ[2] と完全に互換性があり、OpenCL* で記述されているかのように直接起動できます。

インテル® Xe アーキテクチャー・ベースの GPU 向け SIMD コードの生成例

図 10 は、OpenMP* オフロードの例です。ターゲット領域には 2 つの SIMD ループがあり、1 つは `simdlen(16)` で単精度の FMA（積和）を、もう 1 つは `simdlen(8)` で倍精度の FMA を実行します。そのため、コンパイラーは両方のループに対して 512 ビットの SIMD ベクトル化を行うことができます。

```
Float a[N][M]; double b[N][M];
... ...
#pragma omp target teams distribute parallel for map(tofrom:a[0:N][0:M]) map(tofrom:b[0:N][0:M])
    for (int k = 0; k < N; ++k) {
      float  x = k * 1.0f;
      double y = k * 1.0;
#pragma omp simd simdlen(16)
      for (int j = 0; j < M; ++j) {
        a[k][j] = a[k][j] + x*a[k][j];
      }

#pragma omp simd simdlen(8)
      for (int j = 0; j < M; ++j) {
        b[k][j] = b[k][j] + y*b[k][j];
      }
}
... ...
```

図 10. OpenMP* ターゲット領域の異なる SIMD 幅の例

SIMD ループのベクトル化では、コンパイル時にループのトリップカウントが判明していれば、コンパイラーはループのアンロールを決定できます。このプログラム例では、**図 11** に示すように、トリップカウント `M=32` に対して、最初の SIMD ループを `SIMD16` でベクトル化して `2` でアンロールし、2 番目の SIMD ループを `SIMD8` でベクトル化して `4` でアンロールしています。コンパイラーに共通する問題は、コンパイル時にループのトリップカウントが不明であることです。しかし、アプリケーション・プログラマーがトリップカウントを予測し、`#pragma loop count` を使用してコンパイラーにヒントを与えることができれば、コンパイラーは計算依存ループ (つまり、メモリーアクセスよりも計算に多くの時間を要する) をアンロールできるようになります。

```
        ......
        mad (16|M0)         r7.0<1>:f     r5.0<1;0>:f       r5.0<1;0>:f   r1.6<0>:f  {Compacted,$8.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r33     r7      0x80   0x020D43FF    {$3}
(W&f1.0.any16h) send.dc1 (16|M0)   r9     r34     null    0x0    0x022D0BFF    {$9}
        mad (16|M0)         r11.0<1>:f    r9.0<1;0>:f       r9.0<1;0>:f   r1.6<0>:f  {Compacted,$9.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r35     r11     0x80   0x020D43FF    {A@1,$6}
(W&f1.0.any16h) send.dc1 (16|M0)   r13    r36     null    0x0    0x022D0BFF    {$10}
        mad (8|M0)               r15.0<1>:df   r13.0<1;0>:df     r13.0<1;0>:df r4.2<0>:df {$10.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r37     r15     0x80   0x020D43FF
(W&f1.0.any16h) send.dc1 (16|M0)   r17    r38     null    0x0    0x022D0BFF
        mad (8|M0)               r19.0<1>:df   r17.0<1;0>:df     r17.0<1;0>:df r4.2<0>:df {$11.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r39     r19     0x80   0x020D43FF    {A@1,$4}
(W&f1.0.any16h) send.dc1 (16|M0)   r22    r40     null    0x0    0x022D0BFF    {$12}
        mad (8|M0)               r24.0<1>:df   r22.0<1;0>:df     r22.0<1;0>:df r4.2<0>:df {$12.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r41     r24     0x80   0x020D43FF    {A@1,$7}
(W&f1.0.any16h) send.dc1 (16|M0)   r26    r42     null    0x0    0x022D0BFF    {$13}
        mad (8|M0)               r28.0<1>:df   r26.0<1;0>:df     r26.0<1;0>:df r4.2<0>:df {$13.dst}
(W&f1.0.any16h) send.dc1 (16|M0)   null   r43     r28     0x80   0x020D43FF    {A@1,$5}
        ... ...
```

図 11. データ型を基にアンロールして生成されたインテル® GPU 用 SIMD コード

まとめ

この記事では、インテルの CPU と GPU の ISA 向けの、LLVM と GCC コンパイラーにおける SIMD ベクトル化テクノロジーの最近の進化を紹介しました。ハードウェア機能を利用して SIMD 並列処理を行うため、いくつかのベクトル化機能を説明しました。インテル® GPU では、SIMD ベクトル化は既存の一般的な SPMD モデルを補完します。継続的な取り組みとして、インテル® CPU のインテル® AVX-512 およびインテル® AVX-512-FP16/VNNI ISA と第 12 世代インテル® GPU ISA 向けに、LLVM ベースのインテル® oneAPI コンパイラーと GCC コンパイラーにさらなるパフォーマンス・チューニングと最適化が追加される予定です。

関連情報（英語）

[1] Intel, Intel® Graphics Compute Runtime for oneAPI Level Zero and OpenCL Driver, https://github.com/intel/compute-runtime, 2020.

[2] Intel, oneAPI Level Zero Specification, 2020. https://spec.oneapi.com/level-zero/latest/index.html

[3] C++ Standards Committee, Data-parallel vector library, 2020. https://en.cppreference.com/w/cpp/experimental/simd

[4] H. Saito, S. Preis, N. Panchenko, and X. Tian. Reducing the Functionality Gap between Auto-Vectorization and Explicit Vectorization. In Proceedings of the International Workshop on OpenMP (IWOMP), LNCS9903, pp. 173-186, Springer, 2016.

[5] X. Tian, H. Saito, E. Su, J. Lin, et.al. LLVM Compiler Implementation for Explicit Parallelization and SIMD Vectorization. LLVM-HPC@SC 2017: 4:1-4:11

[6] X. Tian, R. Geva, B. Valentine. Unleash the Power of AVX-512 through Architecture, Compiler and Code Modernization, ACM Parallel Architecture and Compiler Technology, September 11-15, 2016, Haifa, Israel.

[7] X. Tian, Bronis R. de Supinski: Explicit Vector Programming with OpenMP* 4.0 SIMD Extensions, HPC Today America, Nov 19. 2014. http://www.hpctoday.com/hpc-labs/explicit-vector-programming-with-openmp-4-0-simd-extensions/

[8] Guei-Yuan Lueh, Kaiyu Chen, Gang Chen, Joel Fuentes, Wei-Yu Chen, Fangwen Fu, Hong Jiang, Hongzheng Li, and Daniel Rhee, C-for-Metal: High Performance SIMD Programming on Intel GPUs. CGO 2021, 289-300.

[9] “Intel C/C++ compilers complete adoption of LLVM” https://www.intel.com/content/www/us/en/developer/articles/technical/adoption-of-llvm-complete-icx.html (日本語参考訳：https://www.isus.jp/products/c-compilers/adoption-of-llvm-complete-icx/)

[10] Matt Masten, Evgeniy Tyurin, K. Mitropoulou, Eric N. Garcia, and H. Saito Function/Kernel Vectorization via Loop Vectorizer, 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)

[11] Intel AVX-512-FP16 Architecture Specification https://software.intel.com/content/dam/develop/external/us/en/documents-tps/intel-avx512-fp16.pdf

[12] Intel Corporation, Vector Function Application Binary Interface https://docplayer.net/197118571-Vector-function-application-binary-interface.html

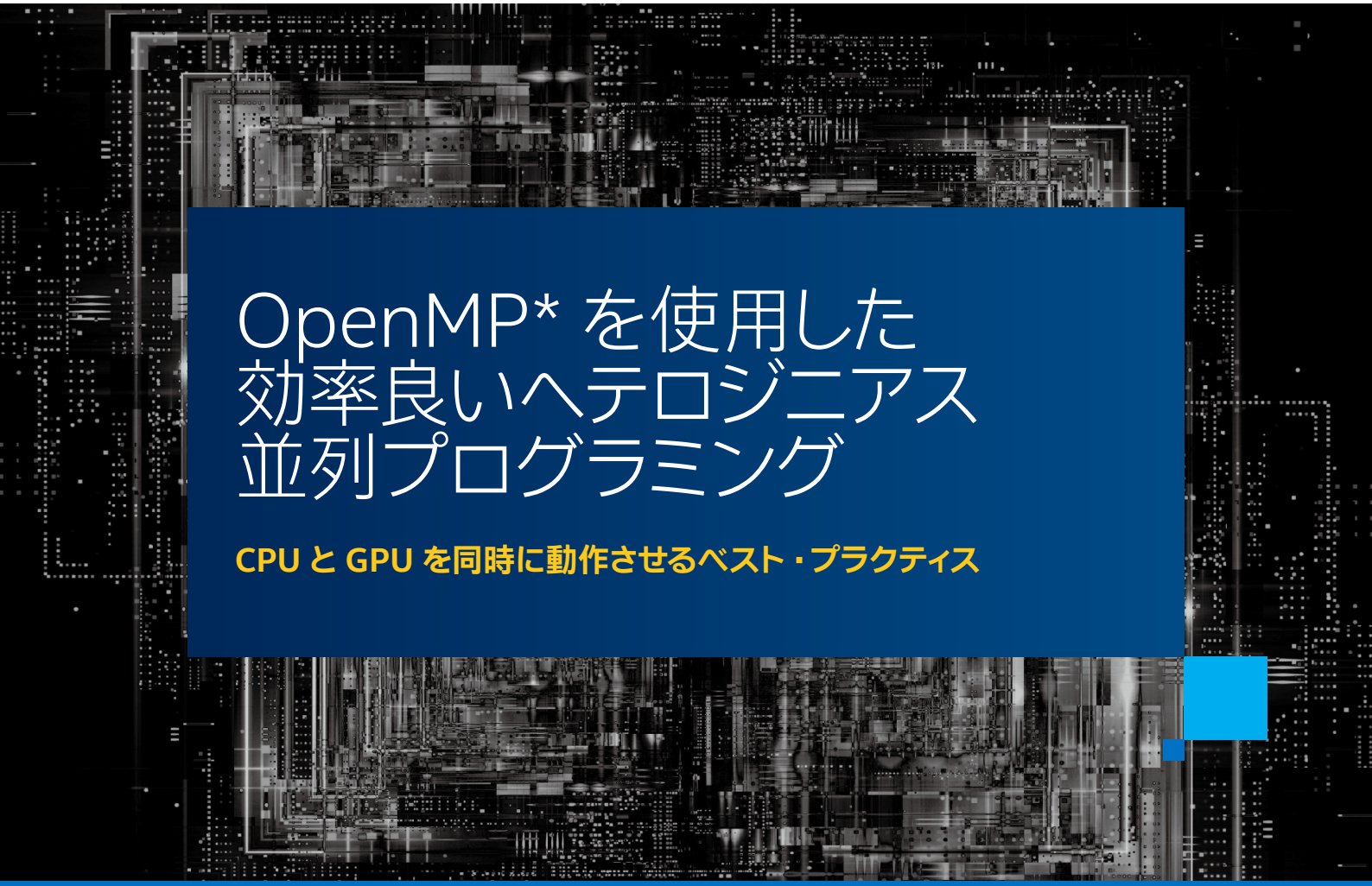
[13] GCC パッチ https:/gcc.gnu.org/git (インテル® AVX-512/VNNI/FP16 サポートを検索)、FP16 パッチについては、https://www.phoronix.com/scan.php?page=news_item&px=AFX-512-FP16-GCC-Patches も参照。

oneAPI

多様なワークロードには多様なアーキテクチャーが必要

インテル® oneAPI ツールキットを使用して、ヘテロジニアス・アプリケーションを素早く正確に開発

ツールキットの詳細 >



OpenMP* を使用した効率良いヘテロジニアス並列プログラミング

CPU と GPU を同時に動作させるベスト・プラクティス

Elmira Volkova 学部生インターン
Alexander Bobyr インテル コーポレーション ソフトウェア・イネーブリング & 最適化エンジニア
Igor Ermolaev インテル コーポレーション 主席エンジニア

GPU などのアクセラレーターに計算をオフロードすると、オフロードした計算が終了するまでホスト CPU がアイドルになるケースがあります。しかし、CPU と GPU のリソースを同時に使用することで、アプリケーションのパフォーマンスを向上させることができます。ヘテロジニアス並列処理を利用する OpenMP* プログラムでは、`master` 節 (注:OpenMP* 5.1 仕様以降では `masked` に変更されています) を使用して CPU と GPU を同時に実行できます。この記事では、OpenMP* を使用して CPU と GPU で非同期計算を行う方法を紹介します。

SPEC ACCEL* 514.pomriq (英語) MRI 再構成ベンチマークは、C 言語で記述され、OpenMP* で並列化されています。このベンチマークは、一部の計算をアクセラレーターにオフロードしてヘテロジニアス並列処理を実行できます。この記事では、ホスト CPU とインテルのディスクリート GPU に計算を分割し、両方のプロセッサーがビジー状態になるようにします。また、インテル® VTune™ プロファイラーを使用して、CPU と GPU の利用率を測定し、パフォーマンスを解析します。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

ここでは、ヘテロジニアス並列処理の開発とパフォーマンス・チューニングの 5 つのステージを紹介します。

1. 並列化に適したコード領域を探します。
2. CPU と GPU の両方がビジーになるように、これらの領域を並列化します。
3. 最適な作業配分係数を見つけます。
4. この配分係数でヘテロジニアス並列アプリケーションを起動します。
5. パフォーマンスの向上を測定します。

最初は、並列領域は GPU でのみ実行され、CPU はアイドルになります（**図 1**）。ご覧の通り、CPU では「OMP Primary Thread」のみが実行され、GPU は ComputeQ オフロードカーネルでフル稼働（**[GPU Execution Units] > [EU Array] > [Active]**）している状態です。

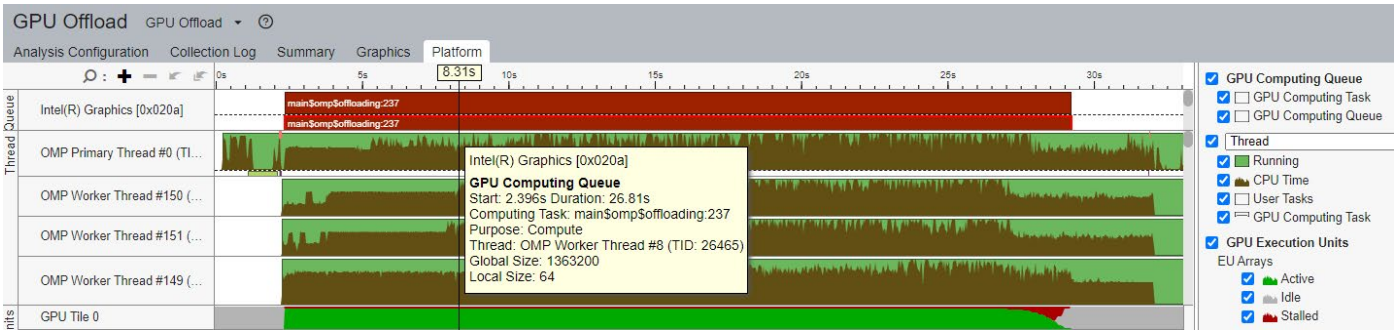


図 1. インテル® VTune™ プロファイラーで初期コードをプロファイルした結果

コードを調べた結果、各配列と各実行領域を複製し、1 つ目のコピーが GPU で実行され、2 つ目が CPU で実行されるようにしました。プライマリー・スレッドは OpenMP* `target` ディレクティブを使用して、GPU に作業をオフロードします。これを図式化したのが**図 2** です。`nowait` ディレクティブは、CPU と GPU で実行されるスレッド間の不要な同期を回避し、スレッド間のロードバランスを向上します。

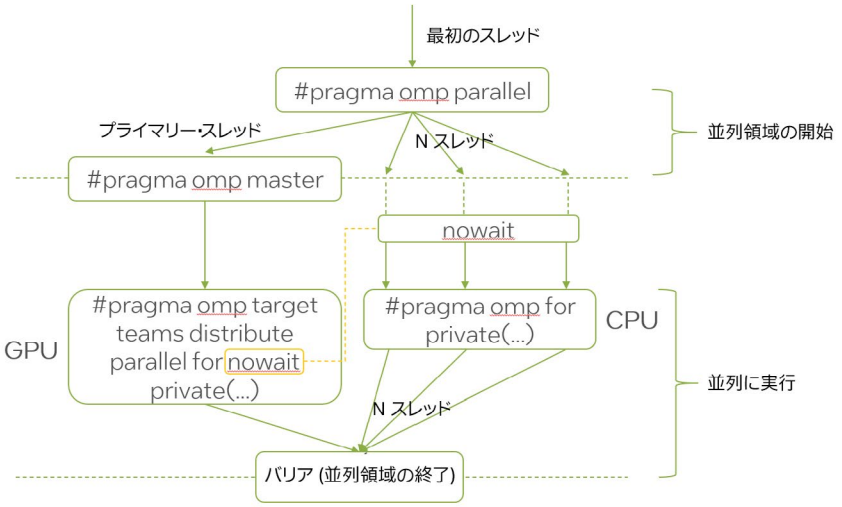


図 2. CPU と GPU をビジーに保つ OpenMP* 並列処理スキーム

CPU と GPU の作業配分のバランスは、`stdin` から読み込まれる `part` 変数で調節されます(**図 3**)。この変数は、GPU にオフロードされるワークロードの割合に `numX` を掛けた値になります。残りの作業は CPU で行われます。OpenMP* ヘテロジニアス並列処理の実装例を**図 4** に示します。

```
    float part;
    char *end;
    part   = strtof(argv[2], &end);

int TILE_SIZE = numX*part;

Qrc = (float*) memalign(alignment, numX * sizeof (float));
Qrg = (float*) memalign(alignment, TILE_SIZE * sizeof (float));
```

図 3. CPU と GPU 間の作業配分係数

```
{
    #pragma omp master
    #pragma omp target teams distribute parallel for nowait private(expArg, cosArg, sinArg)
    for (indexX = 0; indexX < TILE_SIZE; indexX++) {
        float QrSum = 0.0;
        float QiSum = 0.0;
        #pragma omp simd private(expArg, cosArg, sinArg) reduction(+:QrSum, QiSum)
        for (indexK = 0; indexK < numK; indexK++) {
            expArg = PIx2 * (GkVals[indexK].Kx * xg[indexX] + GkVals[indexK].Ky * yg[indexX] + GkVals[indexK].Kz * zg[indexX]);
            cosArg = cosf(expArg);
            sinArg = sinf(expArg);
            float phi = GkVals[indexK].PhiMag;
            QrSum += phi * cosArg;
            QiSum += phi * sinArg;
        }
        Qrg[indexX] += QrSum;
        Qig[indexX] += QiSum;
    }
    #pragma omp for private(expArg, cosArg, sinArg)
    for (indexX = TILE_SIZE; indexX < numK; indexX++) {
        float QrSum = 0.0;
        float QiSum = 0.0;
        #pragma omp simd private(expArg, cosArg, sinArg) reduction(+:QrSum, QiSum)
        for (indexK = 0; indexK < numK; indexK++) {
            expArg = PIx2 * (CkVals[indexK].Kx * xc[indexX] + CkVals[indexK].Ky * yc[indexX] + CkVals[indexK].Kz * zc[indexX]);
            cosArg = cosf(expArg);
            sinArg = sinf(expArg);
            float phi = CkVals[indexK].PhiMag;
            QrSum += phi * cosArg;
            QiSum += phi * sinArg;
        }
        Qrc[indexX] += QrSum;
        Qic[indexX] += QiSum;
    }
}
```

図 4. CPU と GPU を同時に利用する OpenMP* 実装のコード例

インテル® oneAPI DPC++/C++ コンパイラーで次のコマンドライン・オプションを指定してビルドします。

```
-O3 -Ofast -xCORE-AVX512 -mprefer-vector-width=512 -ffast-math
-qopt-multiple-gather-scatter-by-shuffles -fimf-precision=low
-fiopenmp -fopenmp-targets=spir64="-fp-model=precise"
```

表 1 は、CPU と GPU の作業比率（つまり、前述の `part` 変数）を変えた場合のパフォーマンスです。このシステムとワークロードでは、オフロード比率 0.65 が、CPU と GPU の間の最適なロードバランスとなり、プロセッサー・リソースの最適な利用率となります。インテル® VTune™ プロファイラーのプロファイルでは、CPU と GPU の間で作業がより均等に分散され、両方のプロセッサーが有効に活用されていることが分かります（**図 5**）。「OMP Primary Thread」がオフロードされたカーネル（`main:237`）を GPU 上で実行する間、ほかの「OMP Worker Threads」が CPU 上でアクティブになります。

オフロード比率	合計時間（秒）	GPU 時間（秒）
0.00	61.2	0.0
0.20	51.6	8.6
0.40	41.0	16.8
0.60	31.5	24.7
0.65	28.9	26.7
0.80	34.8	32.6
1.00	43.4	40.7

表 1. オフロードされる作業量（part 変数の値）別の hotspot 時間

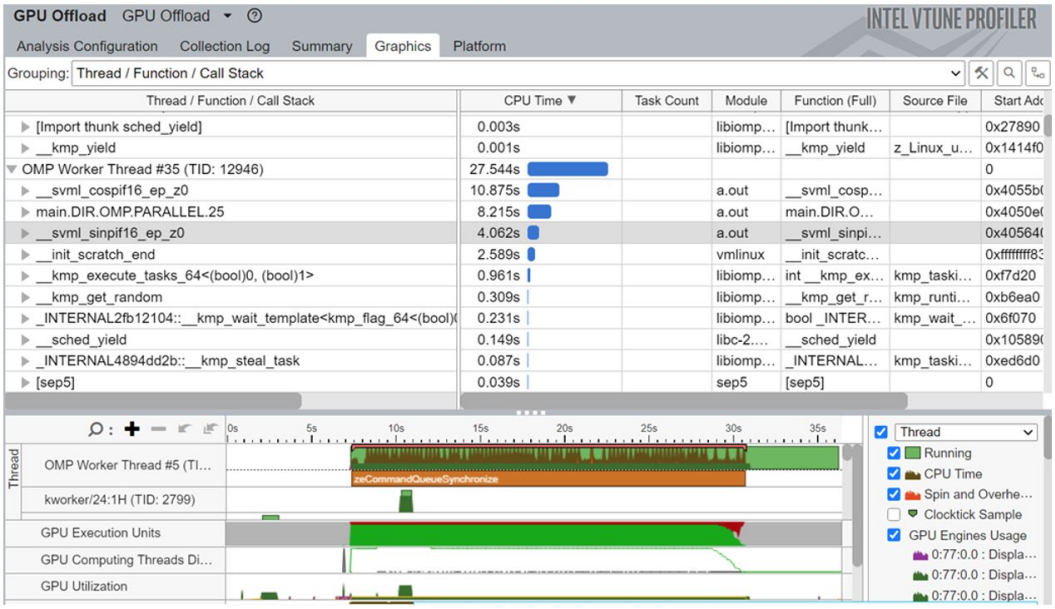


図 5. コードの 65% を GPU にオフロードした場合のプロファイル結果

図 6 は、`part` の値を変えた場合の実行時間です。`part` が `0` の場合は GPU に何もオフロードされず、`part` が `1` の場合はすべての作業がオフロードされます。CPU と GPU にバランス良く作業を振り分ければ、どちらか一方で作業するよりも優れたパフォーマンスが得られることは明らかです。

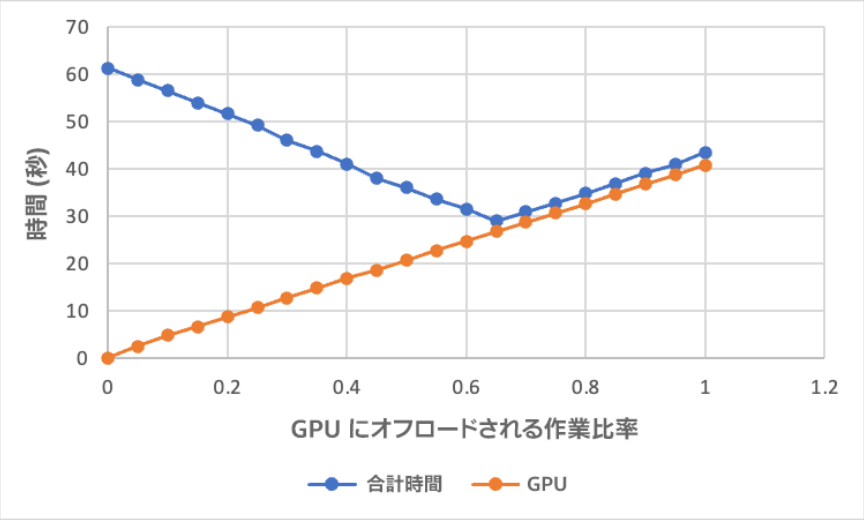


図 6. GPU にオフロードした場合の合計時間と GPU 時間（時間はすべて秒単位)

OpenMP* は、CPU + GPU システムで真の非同期ヘテロジニアス実行をサポートします。パフォーマンスの比較結果やインテル® VTune™ プロファイラーのプロファイル結果から、OpenMP* 並列領域で CPU と GPU をビジー状態に保つことで、最高のパフォーマンスが得られることは明らかです。ぜひこのアプローチを試してみてください。

ArrayFire* と oneAPI、各種ライブラリー、OpenCL* の相互運用性

oneAPI を活用してコードの書き換えを回避

Stefan Yurkevitch ArrayFire ソフトウェア・エンジニア

oneAPI は、ヘテロジニアス・アクセラレーター向けの開発を大幅に簡素化します。一度コードを記述したらどこでも実行できるようにコードを開発する強力な方法を提供します。ArrayFire*（英語）は、多くの計算ドメインで有用な関数の膨大なコレクションを提供している GPU ライブラリーです。ArrayFire* は、oneAPI がソフトウェア開発の世界にもたらす先見性を共有します。この記事では、oneAPI ディープ・ニューラル・ネットワーク (oneDNN)（英語）ライブラリーと SYCL* ベースのデータ並列 C++（DPC++）プログラミング言語を、既存のコードベースに統合する方法を探ります。そして、プログラマーが oneAPI を活用することで、新しいプログラミング・モデルへの移行時にしばしば必要となるコードの書き換えを回避できるようにします。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

SYCL* との相互運用性

oneAPI は、クロスアーキテクチャーの並列プログラミングを簡素化する DPC++ とライブラリーの組み合わせです。ライブラリーは、DPC++ 言語と緊密に統合されています。どちらも、さまざまな方法で OpenCL* 実装との相互運用性を提供します。ベースとなる言語では、ほとんどのユースケースをカバーする 3 つの主要な OpenCL* との相互運用性が提供されています（**図 1**）。既存のコードから SYCL* へ、またはその逆の相互運用関数のフローがあります。

1. カーネル文字列からカーネル・オブジェクトを作成して、DPC++ コード内で既存の OpenCL* カーネルを使用します。
2. 既存の SYCL* オブジェクトから OpenCL* オブジェクトを抽出します。
3. 既存の OpenCL* オブジェクトから SYCL* オブジェクトを作成します。

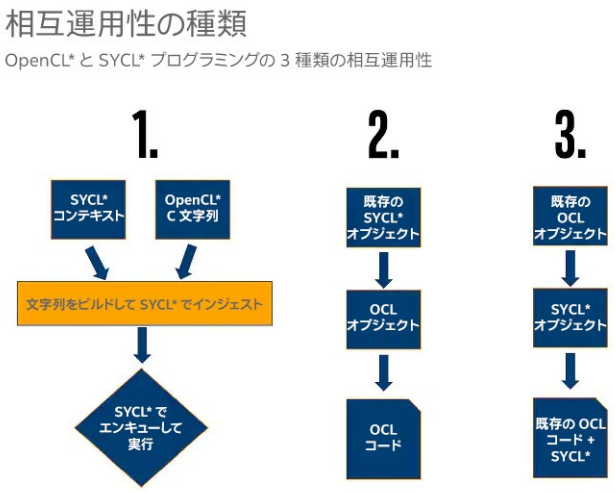


図 1. SYCL* と OpenCL* の相互運用性

これらの相互運用オプションによって既存の ArrayFire* コードベースを統合する方法を考えてみます。最初のケースでは、ArrayFire* カーネルを直接再利用することができます（**図 1** の左）。

```
queue q{gpu_selector()};        // GPU をターゲットとするコマンド キューを作成
program p(q.get_context());     // q と同じコンテキストからプログラムを作成

// R" で示される C++ 生文字列として表現された OpenCL* vecAdd カーネルをコンパイル
p.build_with_source(R"( __kernel void existingArrayFireVecAdd(__global int *a,
                                                              __global int *b,
                                                              __global int *c)
                        {
                                int i = get_global_id(0);
                                c[i] = a[i] + b[i];
                        } )");
  // ここにバッファーを記述 ...
  q.submit([&](handler& h) {
          // ここにアクセサーを記述...
          // カーネルへの引数としてバッファーを設定
          h.set_args(A, B, C);
          // p プログラム・オブジェクトから N 要素に対して vecAdd カーネルを起動
          h.parallel_for(range<1> (N), p.get_kernel("vecAdd"));
  });
```

実際には、ArrayFire* カーネルは単純なバッファーよりも複雑なデータ構造を利用するため、CL 文字列をコピー & ペーストするだけでカーネルを再利用できるわけではありません。ほかの 2 つの方法のいずれかで、データ交換を処理する必要があります。

SYCL* オブジェクトから OpenCL* コンポーネントを抽出する 2 つ目の方法（**図 1** の中央）は、既存の SYCL* オブジェクトに `.get()` を使用するだけの単純なものです。SYCL* オブジェクトの各呼び出しは、対応する OpenCL* オブジェクトを返します。例えば、`cl::sycl::queue::get()` は OpenCL* の `cl_command_queue` を返します。

3 つ目の方法（**図 1** の右）は、既存の OpenCL* オブジェクトを取り込み、それを使用して SYCL* オブジェクトを作成します。`sycl::queue::queue(cl_command_queue,…)` などの SYCL* オブジェクトのコンストラクターを使用します。この場合、コンストラクターは、構築時に OpenCL* リソースの参照カウントを増やすため OpenCL* インスタンスを維持し、SYCL* オブジェクトを破棄する際にインスタンスを解放します。

oneAPI ライブラリーとの相互運用性

oneAPI ライブラリーにも同様の相互運用規則があります。oneMKL のように、DPC++ の相互運用性に直接依存しているライブラリーもあります。これらの関数は、統合共有メモリー（USM）ポインターを受け付けます。この例で使用する oneDNN のように、同様の `.get()` や `constructor()` メカニズムを提供するライブラリーもあります。

oneDNN のデータ構造は、DPC++ と似ていますが、少し異なります。`sycl::device` と `sycl::context` は単一の `dnnl::engine` オブジェクトにまとめられ、`sycl::queue` の代わりに `dnnl::stream` が使用されます。これらの違いはありますが、OpenCL* の相互運用（英語）メカニズムは同じです。OpenCL* オブジェクトは `get` 関数で取得でき、新しい oneDNN オブジェクトは既存の OpenCL* オブジェクトからコンストラクターで作成できます。oneDNN は、同じ機能を持つ明示的な相互運用ヘッダーも提供します。

oneDNN は、サポートするランタイム・バックエンドに柔軟性があります。ハードウェアと対話するため、CPU および GPU エンジンに OpenCL* ランタイムまたは DPC++ ランタイムのいずれかを使用できます。開発者は、OpenCL*（英語）または DPC++（英語）のいずれかを使用するほかのコードで oneDNN を必要とすることがあります。そのため、oneDNN は、対応するオブジェクトと相互運用性がある API 拡張を提供しています。ターゲットに応じて、相互運用 API は `dnnl_ocl.hpp` または `dnnl_sycl.hpp` ヘッダーのいずれかで定義されています。ここで紹介するユースケースは、oneDNN の推論エンジンの機能を ArrayFire* ライブラリーが提供する既存の前処理機能で補完することを目的としています。現時点では、これは OpenCL* の相互運用関数を介して行われます。

ArrayFire* と oneDNN : 詳細

cnn_inference_f32.cpp（英語）サンプルを使用して、OpenCL* と oneDNN の相互運用性を詳しく調査します。このサンプルは、oneDNN を使用して推論用の AlexNet ネットワークをセットアップします。ここでは、ArrayFire* の多くの OpenCL* 画像処理関数を使用して、既存の推論エンジンにデータを供給する前にユーザー入力画像の前処理を行います。ワークフローには、以下のステップが含まれます。

- 関連する相互運用ヘッダーをインクルードします。
- ArrayFire* と `cl_context` を共有し、GPU エンジンを作成します。
- OpenCL* 相互運用インターフェイスを利用して GPU コマンドキューを作成します。
- ArrayFire* で前処理とデータの準備を行います。
- GPU メモリー記述子 / オブジェクトを作成します。
- OpenCL* 相互運用インターフェイスを介して GPU メモリーにアクセスして、入力を取得します。
- oneDNN のプリミティブ / 記述子 / メモリーを作成し、ネットワークを構築します。
- oneDNN で通常通りネットワークを実行します。
- GPU メモリーを解放します。

最初に、ArrayFire* と oneDNN の両方の相互運用ヘッダーをインクルードします。OpenCL* ヘッダーもインクルードします。

```
#include "oneapi/dnnl/dnnl.hpp"     // oneDNN ヘッダー
#include "oneapi/dnnl/dnnl_ocl.hpp" // oneDNN OpenCL* 相互運用ヘッダー

#include <CL/cl.h>       // OpenCL* ヘッダー

#include <arrayfire.h>   // ArrayFire* ヘッダー
#include <af/opencl.h>   // ArrayFire* OpenCL* 相互運用ヘッダー
```

次に、ArrayFire* から OpenCL* のコンテキストとキューを取得し、oneDNN と共有します。

```
cl_device_id     af_device_id = afcl::getDeviceId();
cl_context       af_context   = afcl::getContext();
cl_command_queue af_queue     = afcl::getQueue();
```

OpenCL* オブジェクトを使用して、対応する oneDNN オブジェクトを作成します。相互運用ヘッダー（英語）で定義された相互運用関数を使用します。これらの関数は、追加の `ocl_interop` 名前空間に存在します。オブジェクトは oneDNN スコープのライフタイムを通して保持されることを忘れないでください。

```
dnnl::engine eng = dnnl::ocl_interop::make_engine(af_device_id, af_context);
dnnl::stream s   = dnnl::ocl_interop::make_stream(eng, af_queue);
```

そして、ArrayFire* ライブラリーの GPU アクセラレーション機能を利用して、画像の読み込みや前処理を行います。

```
// oneDNN と同じコンテキストに空の配列を作成
af::array images = af::constant(0.f, h, w, 3, batch);
images = read_images(directory);
images = af::resize(images, 227, 227) / 255.f; // AlexNet の入力サイズに変更して
                                               // 正規化 [0-1]
     images = af::reorder(images, 3, 2, 0, 1); // hwcn -> nchw
...                                            // その他の前処理
```

oneDNN は、dnnl::memory オブジェクトを必要とします。これは生のメモリーではなく、メモリーと dnnl::descriptor のような追加のメタデータを一緒にしたものです。oneDNN はバッファーと USM メモリーの両方のモデルをサポートしており、バッファーがデフォルトです。相互運用性をサポートする oneDNN メモリー・オブジェクトを構築するには、以下の相互運用関数（英語）を使用します。

```
ocl::interop make_memory(
   const memory::desc& memory_desc, // メモリーの形状とレイアウトを示す記述子
   const engine& aengine,           // 相互運用エンジン
   memory_kind kind,                // バッファーまたは USM
   void* handle = DNNL_MEMORY_ALLOCATE // ストレージへのハンドル
   )
```

記述子は、AlexNet の入力が 227 × 227 の NCHW 画像であることを想定したサンプルの記述子と同じです。engine は、ArrayFire* と oneDNN で共有している実行エンジンです。memory_kind は、USM またはバッファーのどちらのインターフェイスを使用するかを指定します。ハンドルポインターを渡す場合、メモリーの種類に応じて処理する必要があります。ハンドルが USM ポインターまたは OpenCL* バッファーの場合、oneDNN ライブラリーはバッファーを所有しないため、ユーザーがメモリーを管理する責任があります。特別な DNNL_MEMORY_ALLOCATE 値を指定すると、ライブラリーがユーザーに代わって新しいバッファーを割り当てます。

oneDNN はバッファーと USM の両方のメモリーモデルをサポートしているので、エンジンとキューを ArrayFire* と共有するオブジェクトに置き換えると、互換性のないメモリー作成モードになります。dnnl::memory オブジェクトの生成時に、以下のエラーが発生します。

```
oneDNN error caught:
        Status: invalid_arguments
        Message: could not create a memory object
```

デフォルトの dnnl::memory 生成の代わりに、以下のように相互運用関数を使用する必要があります。

```
cl_mem *src_mem = images.device<cl_mem>(); // ArrayFire*  から cl_mem を取得
dnnl::memory user_src_memory = ocl_interop::make_memory( // 相互運用メモリー関数
       {{conv1_src_tz}, dt::f32, tag::nchw},             // 記述子を作成
       eng,                                              // エンジンを指定
       ocl_interop::memory_kind::buffer,                 // メモリーの種類を指定
       *src_mem);                                        // cl_mem ハンドルを渡す
```

これは、デフォルトの dnnl::memory 割り当てのすべてのインスタンスに適用されます。ocl_interop::memory_kind::buffer を指定する相互運用関数を使用する必要があります。

```
ocl_interop::make_memory(descriptor, engine, ocl_interop::memory_kind::buffer);
```

最後に、すべての重みをロードしたら、推論プリミティブを作成して、通常通り呼び出します。ネットワークの実行が完了したら、リソースを解放します。

```
// 追加の AlexNet ネットワークのセットアップ
// cnn_inference_f32.cpp の後に重みをロード
...

// 入力を使用して完全な推論を行うためすべてのプリミティブ・ステップを実行
for (size_t i = 0; i < net.size(); ++i) {
  net.at(i).execute(s, net_args.at(i));
}

s.wait();   // ストリームがメモリーへの書き込みを終了するのを待機

images.unlock();  // メモリーの所有権を ArrayFire* に返してリソースを解放
```

DPC++ ランタイムではなく、OpenCL* ランタイムで oneDNN を実行していることを確認します。これは、SYCL_DEVICE_FILTER=opencl 環境変数を指定することで達成できます。修正後の動作する cnn_inference_f32.cpp は、Gist（英語）で参照できます。

まとめ

oneAPI は、既存の OpenCL* コードベースを新しいヘテロジニアス・プログラミング・アプローチと統合するために必要なすべてのツールを提供します。DPC++ では、OpenCL* オブジェクトをどちらの方向でも共有できます。oneAPI ライブラリーは、相互運用タスクを処理する独自の方法を備えています。わずかなコード変更で、OpenCL* ライブラリー全体を書き直すのではなく、再利用することができます。oneAPI は、すでに有用なコードの再開発作業を回避することで、将来の開発時間を節約します。

intel software

コードの境界を乗り越える

インテル® DevCloud for oneAPI でクロスアーキテクチャー・プログラミングのパワーを体験してください。

デモ

Mandelbrot デモを異なるアーキテクチャーで実行して、クロスアーキテクチャーのパフォーマンスを確認できます。

学習

サンプルコードを含む 25 の Jupyter* Notebook でデータ並列 C++ を実際に体験できます。

開発

最新のインテルの CPU、GPU、FPGA で将来に対応したアプリケーションを計画してテストできます。

今すぐ開始（英語）>

コンパイラーの最適化に関する詳細は、最適化に関する注意事項（software.intel.com/articles/optimization-notice#opt-jp）を参照してください。
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。
© Intel Corporation

oneAPI レベルゼロ・インターフェイスの使用

レベルゼロ API の概要紹介

Rama Kishan Malladi Amazon Web Services ソリューション・アーキテクト
Nitya Hariharan インテル コーポレーション アプリケーション・エンジニア

oneAPI 仕様（英語）は、アクセラレーター・アーキテクチャー間で同じ言語、API、プログラミング・モデルを提供することで、ソフトウェア開発を簡素化するものです。この仕様では、さまざまなアーキテクチャーに共通するデータ並列ドメイン用の API セットを定義しています。API とダイレクト・プログラミングのいずれのアプローチも、データ並列処理（各データ要素に対して同じ計算を行うこと）を基本としています。oneAPI プラットフォームは、ホストとアクセラレーター・デバイスで構成されています（**図 1**）。API プログラミング・モデルは、oneMKL、oneDPL、oneDNN、oneCCL などのライブラリーを使用して実装されています。ダイレクト・プログラミングは DPC++ で行われます。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

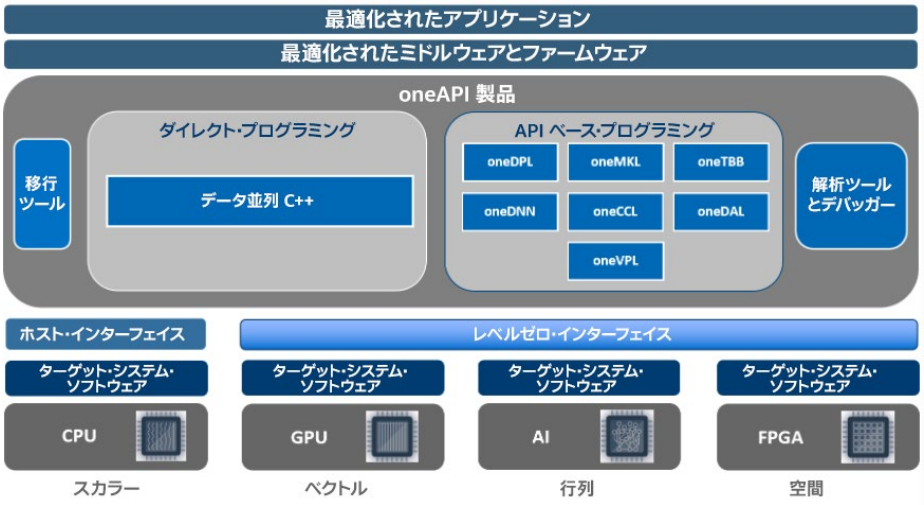


図 1. oneAPI プラットフォームの実行モデル。インテルの oneAPI 仕様の実装には、プログラミング・ツールも含まれています。

レベルゼロ：はじめに

oneAPI レベルゼロは、oneAPI プラットフォームのデバイスに対して、低レベルのメタルに直結したインターフェイスを提供します。レベルゼロは、デバイスの検出、メモリー割り当て、プロセス間通信、カーネルのサブミット、同期、メトリックのレポートを行う、きめ細かい明示的な制御 /API に加えて、広範な言語機能のサポートを提供します。また、デバイスの論理レベルと物理レベルの抽象化の両方を利用できる API を備えています。ほかの低レベル API（OpenCL* など）の影響を大きく受けていますが、レベルゼロは独自に進化するように設計されており、GPU や FPGA のようなほかの計算デバイスをサポートしています。ほとんどのアプリケーションでは、レベルゼロ API が提供する追加の制御は必要ありません。レベルゼロ API は、次のような、上位のランタイム API やライブラリーで必要とされる明示的な制御を想定しています。

- デバイスの検出と切り分け
- カーネルの実行とスケジュール
- ピアツーピア通信
- メトリックとプロファイル
- カーネル・プロファイル、インストルメンテーション
- システム管理、電力やパフォーマンスの照会

レベルゼロの C API は、共有インポート・ライブラリーによってアプリケーションに提供されます。そのため、C/C++ アプリケーションでは「`ze_api.h`」をインクルードして、「`ze_api.lib`」（または共有ライブラリー）とリンクする必要があります。

レベルゼロ：API

レベルゼロ API は、コア、ツール、およびシステム・プログラミングに分類されますが、この記事ではコア・プログラミング API（**図 2**）のみを説明します。コア・プログラミング API は、デバイス、ドライバー、コンテキスト、メモリー、コマンドキュー / リスト、同期、バリア、モジュール、カーネルをサポートしています。**表 1** と**表 2** は、よく使用される API の一覧です。**図 3** は、レベルゼロを使用した実行フローです。

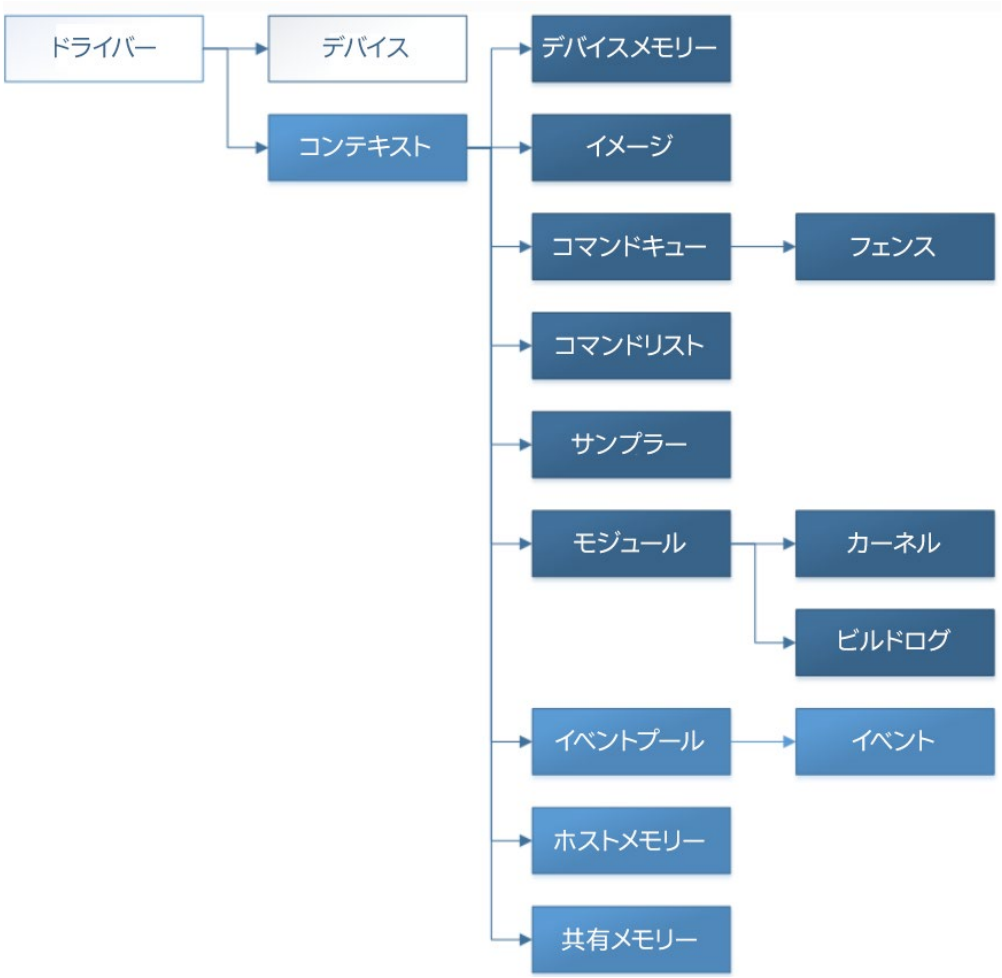


図 2. コア・プログラミング API のコンポーネント

デバイス、コンテキスト、キューの API	説明
zeInit、zeDriverGet	すべてのドライバーを初期化して検出します。
zeDeviceGet、zeDeviceGetProperties	DEVICE_TYPE のドライバー・インスタンスを探します。
zeContextCreate	コンテキストを作成します。
zeMemAllocHost、zeMemAllocDevice、zeMemAllocShared	ホスト、デバイス、または共有メモリー上にメモリーを割り当てます。
zeCommandQueueCreate	コマンドキューを作成します。
zeCommandListCreate	コマンドリストを作成します。
zeCommandQueueExecuteCommandLists	コマンドキューのコマンドリストを実行します。
zeCommandQueueSynchronize	ホストとデバイスを同期します。

表 1. デバイス、コンテキスト、およびキューのレベルゼロ API

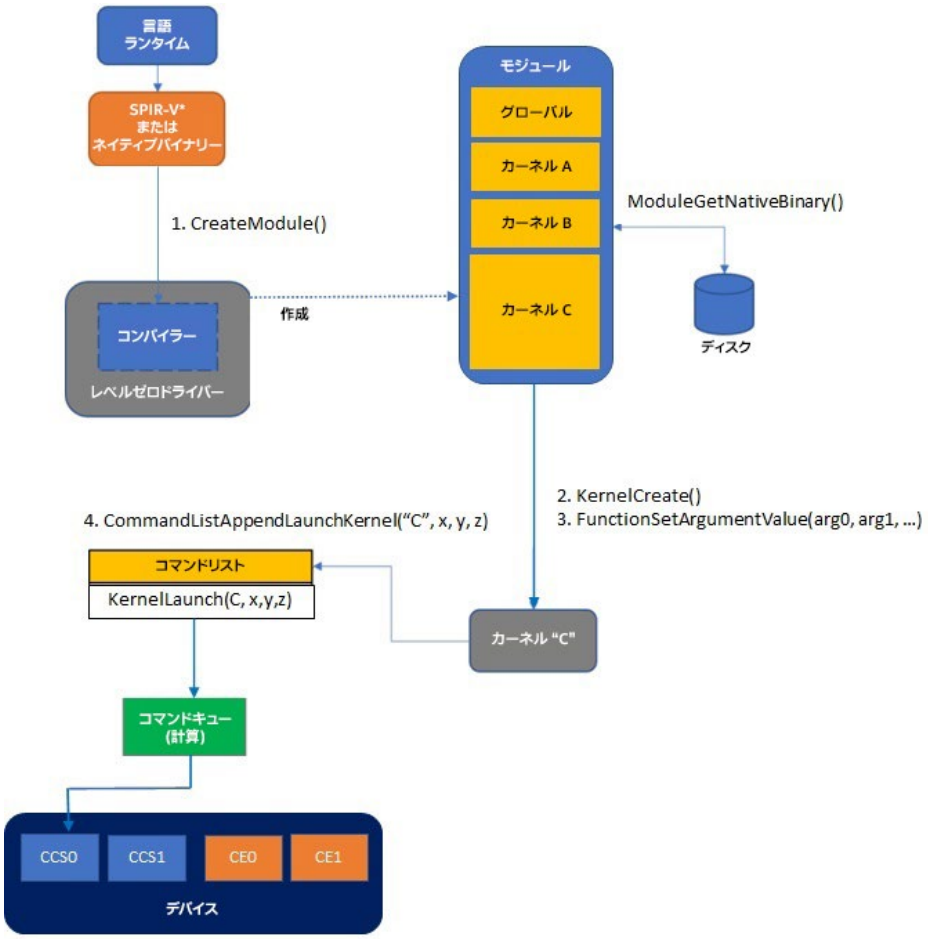


図 3. 高レベルのフロー：コマンドリスト、キュー、モジュール、およびデバイス上のカーネル実行

同期、モジュール、カーネルの API	説明
zeEventPoolCreate	イベントプールを作成します。
zeCommandListAppendLaunchKernel	カーネル、引数、シグナルをコマンドリストに追加します。
zeEventHostSynchronize	イベントの完了を待機します。
zeModuleCreate	IL コードをコンパイルするか、ネイティブバイナリーをロードしてモジュールを作成します。
zeKernelCreate	モジュール内のカーネルを参照します。
zeKernelSetArgumentValue	カーネル起動時の引数を設定します。

表 2. 同期およびカーネル関連関数のレベルゼロ API

リスト 1 は、ドライバーとデバイスの検出を行う main プログラムです。**リスト 2** は、関数 RunTest のカーネルの実行を示しています。

```
// ドライバーコード(main)
int main(int argc, char *argv[])
{
   zeInit(ZE_INIT_FLAG_GPU_ONLY);

   ze_driver_handle_t driverHandle;
   zeDriverGet(&driverCount, &driverHandle);

   uint32_t deviceCount = 1;
   ze_device_handle_t device;
   zeDeviceGet(driverHandle, &deviceCount, &device);

   ze_device_properties_t deviceProperties = {};
   zeDeviceGetProperties(device, &deviceProperties);

   uint32_t subDeviceCount = 0;
   zeDeviceGetSubDevices(device, &subDeviceCount, nullptr);
   ze_device_handle_t subDevices[2] = {};
   zeDeviceGetSubDevices(device, &subDeviceCount, subDevices);

  for (uint32_t i = 0; i < subDeviceCount; i++) {
      ze_device_properties_t deviceProperties = {};
      zeDeviceGetProperties(subDevices[i], &deviceProperties);
   }

   RunTest(driverHandle, subDevices, device, subDeviceCount, outputValidBool);
}
```

リスト 1. ドライバーとデバイス検出のレベルゼロの例

```
void RunTest(ze_driver_handle_t &driverHandle, ze_device_handle_t *subDevice,
             ze_device_handle_t rootDevice, uint32_t subDeviceCount,
             bool &validRet)
{
  // 変数の初期化、ホストメモリーの割り当て
  ...
  // コンテキスト、コマンドキュー、リストの作成
  zeContextCreate(driverHandle, &contextDesc, &context);
  ...
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeCommandQueueCreate(context, subDevice[i], &cmdQueueDesc, &cmdQueue[i]);
    zeCommandListCreate(context, subDevice[i], &cmdListDesc, &cmdList[i]);
  }
  ...
  // デバイスで実行するカーネルが含まれる IL ファイル(SPIR-V* 形式)のロード
  const char *modulePath = "Gpu_Module_Kernel.spv";
  uint32_t spirvSize = 0;
  auto spirvModule = readBinaryFile(modulePath, spirvSize);
  ...
  zeModuleCreate(context, subDevice[i], &moduleDesc, &module[i], nullptr);
  zeKernelCreate(module[i], &kernelDesc, &kernel[i]);
  zeKernelSetGroupSize(kernel[i], groupSizeX, groupSizeY, groupSizeZ);
  ...
  // デバイスメモリーの割り当て、コマンドリストへのメモリーコピー操作の追加
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeMemAllocDevice(context, &deviceDesc, bufferWidth*sizeof(float), 0,
                     subDevice[i], &d_input[i]);
    zeCommandListAppendMemoryCopy(cmdList[i], d_input[i], input[i],
                     bufferWidth*sizeof(float), nullptr, 0, nullptr);
    ...
  }
    // ホストからデバイスへのデータのコピー(割り当てとコピーコマンドを実行)
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListClose(cmdList[i]);
    zeCommandQueueExecuteCommandLists(cmdQueue[i], 1, &cmdList[i], nullptr);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandQueueSynchronize(cmdQueue[i], UINT32_MAX);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListReset(cmdList[i]);
  }
  ...
  // カーネル引数の設定
  for (uint32_t i=0; i<num_tiles; i++)
  {
    arg_indx = 0;
    start_idx = i * segment_size;

    zeKernelSetArgumentValue(kernel[i], arg_indx++, sizeof(d_input[i]),
                             &d_input[i]);
    zeKernelSetArgumentValue(kernel[i], arg_indx++, sizeof(d_input[i]),
                             &d_input[i + 1 == num_tiles ? 0 : i + 1]);
    ...
  }
  // イベントプールの作成、カーネル起動コマンドへの追加
  ze_event_pool_handle_t eventPool;
  zeCommandListAppendLaunchKernel(cmdList[i], kernel[i], &group_count,
                                  kernelTsEvent[i], 0, nullptr);
  ... // (続く)
```

リスト 2. カーネル実行のレベルゼロの例

```
  // コマンドリストの実行、キューのコマンド実行の同期
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListClose(cmdList[i]);
    zeCommandQueueExecuteCommandLists(cmdQueue[i], 1, &cmdList[i], nullptr);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandQueueSynchronize(cmdQueue[i], UINT32_MAX);
  }
  for (uint32_t i=0; i<num_tiles; i++) {
    zeCommandListReset(cmdList[i]);
  }
  // カーネルイベント統計の取得、実行時間の計算
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeEventQueryKernelTimestamp(kernelTsEvent[i], &kernelTsResults);
    uint64_t kernelDuration = kernelTsResults.context.kernelEnd -
                              kernelTsResults.context.kernelStart;
  }
  // デバイスからホストへデータをコピー
  for (uint32_t i=0; i<num_tiles; i++)
  {
    zeCommandListAppendMemoryCopy(cmdList[i], output[i], d_output[i],
                      bufferWidth*sizeof(float), nullptr, 0, nullptr);
  }
  ...
  // カーネル、メモリー、コンテキスト、イベント、およびその他のオブジェクトの破棄
  zeEventPoolDestroy(eventPool);
  for (size_t i=0; i<num_tiles; i++) {
    zeMemFree(context, d_input[i]);
    zeKernelDestroy(kernel[i]);
    zeCommandListDestroy(cmdList[i]);
    ...
  }
  zeContextDestroy(context);
  ...
}
```

リスト 2. カーネル実行のレベルゼロの例（続き）

OpenMP* の例

レベルゼロ API は、OpenMP* オフロードコードのコンパイル時にもバックエンドで生成されます。これらの API 呼び出しは、`LIBOMPTARGET_DEBUG` 環境変数と `LIBOMPTARGET_INFO` 環境変数が `1` 以上に設定されている場合はダンプされます。**リスト 3** は、デバイス上に構造体配列（AoS）を割り当て、ホスト上で初期化したデータをコピーして、デバイス上で更新した後、ホストへ転送する例を示します。ログは、前述の 2 つの環境変数を `99` に設定して生成されたものです。

リスト 4 は、ログから抜粋したレベルゼロ API 呼び出しの一部です。`zeMemAllocDevice` 呼び出しでデバイス上にデータを割り当て、続く `zeCommandListAppendMemoryCopy` 呼び出しでホストからデバイスへデータをコピーします。カーネルの計算が終了すると、2 つ目の `zeCommandListAppendMemoryCopy` 呼び出しで、デバイスからホストへ更新されたデータをコピーします。そして、`zeMemFree` 呼び出しで AoS を削除し、デバイスメモリーをメモリープールに返します。これらの呼び出しは、**リスト 2** と似ています。ここでは、OpenMP* プラグマに直接関連するレベルゼロ呼び出しに注目します。ログには、ホストからデバイスへのポインターのコピー、メモリーブロックのプロパティーの取得、ホストからデバイスポインターへのマッピング、終了時のデバイスのクリーンアップに使用される多くの呼び出しが表示されます。このほかにも、レベルゼロ API は、メモリーの割り当て、ホストからデバイスへのコピー、またはホストとデバイスの間での共有をより細かく制御できる呼び出しを提供します。

```
struct force_data
{
  float Mass;
  int index;
};

#pragma omp declare target
struct force_data *myData;
#pragma omp end declare target

int main()
{
  int a[10], max=10;

  // ホスト上に構造体配列を割り当て
  myData = (struct force_data*) malloc(max  * sizeof(struct force_data));

  for(int i = 0; i < max; i++)
  {
    myData[i].index = 1; a[i] = 2;
  }

 // 1. デバイス上にデータを割り当て
  #pragma omp target enter data map(alloc:myData[0:max])
  {
// 2. デバイス上のデータを更新
  #pragma omp target teams distribute parallel for map(to:a)
    for(int i=0; i < max; i++)
      myData[i].index = myData[i].index + a[i];
  }

// 3. デバイス上のデータを削除
  #pragma omp target exit data map(delete:myData[0:max])
  for(int i=0; i < max; i++)
     printf("%d\n", myData[i].index);
}
```

リスト 3. OpenMP* オフロードの例

```
Libomptarget (pid:6780)  --> Entering OpenMP data region at unknown:31:31 with 1 arguments:
Libomptarget (pid:6780)  --> alloc(myData[0:10])[80]
…
Target LEVEL0 RTL (pid:10428)  --> ZE_CALLER: zeMemAllocDevice ( context, &deviceDesc, Size,
Align, Device, &mem )
…
Libomptarget (pid:10428)  --> Copying data from host to device, HstPtr=…., TgtPtr=….,
Size=80, Name=myData[0:10]
…
Target LEVEL0 RTL (pid:10428)  --> Copy Engine is used for data transfer
Target LEVEL0 RTL (pid:10428)  --> ZE_CALLER: zeCommandListAppendMemoryCopy ( cmdList, Dest,
Src, Size, nullptr, 0, nullptr )
…
Libomptarget (pid:6780)  --> Entering OpenMP kernel at unknown:41:41 with 6 arguments:
Libomptarget (pid:6780)  --> tofrom(myData)[8] (implicit)
…

Libomptarget (pid:6780)  --> Updating OpenMP data at unknown:46:46 with 1 arguments:
Libomptarget (pid:6780)  --> from(myData[0:10])[80]

Libomptarget (pid:6780)  --> Copying data from device to host, TgtPtr=0xffffd556aa640000,
HstPtr=0x0000000000d362b0, Size=80, Name=myData[0:10]

Target LEVEL0 RTL (pid:6780)  --> Copy Engine is used for data transfer
Target LEVEL0 RTL (pid:6780)  --> ZE_CALLER: zeCommandListAppendMemoryCopy ( cmdList, Dest,
Src, Size, nullptr, 0, nullptr )

Libomptarget (pid:6780)  --> Deleting tgt data 0xffffd556aa640000 of size 80

Target LEVEL0 RTL (pid:6780)  --> Returned device memory 0xffffd556aa640000 to memory pool
Target LEVEL0 RTL (pid:6780)  --> ZE_CALLER: zeMemFree ( Context, (void *)block->Base )
Target LEVEL0 RTL (pid:6780)  --> ZE_CALLEE: zeMemFree (
Target LEVEL0 RTL (pid:6780)  -->     hContext = 0x0000000000d33500
Target LEVEL0 RTL (pid:6780)  -->     ptr = 0xffffd556aa640000
```

リスト 4. OpenMP* オフロード向けにバックエンドで生成されるレベルゼロ API 呼び出し

まとめ

レベルゼロ API は、主にライブラリーやフレームワーク開発者を対象としており、ほとんどのアプリケーション開発者は、レベルゼロ API が提供する追加の制御を必要としないでしょう。レベルゼロ API は、デバイス検出、メモリー管理、カーネルのサブミット、プロセス間通信などに対して、よりきめ細かい明示的な制御を提供します。この記事では、基本的な例を使用してレベルゼロのプログラミングを紹介しました。OpenMP* オフロードの例では、オフロード・アクセラレーター・デバイスへのメタルに直結したインターフェイスを提供する、バックエンドで生成される一連の呼び出しに関する情報も得られます。レベルゼロ API の詳細は、oneAPI レベルゼロ仕様（英語）を参照してください。

Habana* Gaudi* 上での MLPerf* トレーニングの SigOpt を使用したハイパーパラメーター最適化

AI トレーニングでより高い精度とより高速な収束を実現

Basem Barakat インテル コーポレーション Habana Labs シニアエンジニア
Evelyn Ding インテル コーポレーション Habana Labs シニアエンジニア
Joshua Mora インテル コーポレーション Habana Labs 主席エンジニア

大規模なディープラーニング（DL）のトレーニング・ワークロードのパフォーマンスを最適化するには、コストがかかります。多くの専門分野にわたるチームからの貢献に加え、DL トレーニング用に設計された大規模な計算インフラストラクチャーが必要です。最適化手法は多岐にわたり、データ並列トレーニング・パラダイムの集合演算の計算 / 通信関連の最適化、独立タスクのオーケストレーション / スケジューリング、データセットの前処理 / 拡張のパイプライン化、収束率を高める軽量なアルゴリズムの最適化、`fp32` と `bfloat16` の混合精度を利用した数値的最適化などがあります。

この記事では、ハイパーパラメーター最適化（HPO）に基づき、目標精度（収束エポック）を維持しつつ、トレーニング・エポック数を削減する最適化手法に注目します。ここでは、Habana* Gaudi* トレーニング・プロセッサーの MLPerf* トレーニング・ワークロードに HPO を適用します。AI に特化したインテルの Habana Labs と SigOpt が共同で行ったこの作業では、モデルのトレーニング時間が改善され、ResNet50（RN50）モデルの最適なハイパーパラメーターを得るために必要な計算リソースが減少し、グリッド検索の利点を活用して MLPerf* モデルのトレーニング時間を短縮し、グリッド検索に関連した Gaudi* 時間を短縮できました。

MLPerf* トレーニング・ワークロード

2018 年以降、MLPerf*（英語）ベンチマーク・スイートは、多種多様なコンピューティング・インフラストラクチャー上で動作する幅広いニューラル・ネットワーク・モデルを評価するため、AI コミュニティーによって使用されています。AI の急速な進化を反映するため、ニューラル・ネットワークは提出のたびに再評価されています。

RN50 トレーニングには、LARS（Layer-wise Adaptive Rate Scaling）（英語）アルゴリズム（**図 1**）を活用します。ハイパーパラメーターとそれぞれの MLPerf* の制約を**表 1** に示します。このワークロードでは、75.9% という特定の目標精度（AC）を達成する必要があります。

Algorithm 1 LARS

Input: $x_1 \in \mathbb{R}^d$, learning rate $\{\eta_t\}_{t=1}^{T}$, parameter $0 < \beta_1 < 1$, scaling function ϕ, $\epsilon > 0$
Set $m_0 = 0$
for $t = 1$ **to** T **do**
  Draw b samples S_t from $\mathbb{P}$
  Compute $g_t = \frac{1}{|\mathcal{S}_t|} \sum_{s_t \in \mathcal{S}_t} \nabla \ell(x_t, s_t)$
  $m_t = \beta_1 m_{t-1} + (1 - \beta_1)(g_t + \lambda x_t)$
  $x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi(\|x_t^{(i)}\|)}{\|m_t^{(i)}\|} m_t^{(i)}$ for all $i \in [h]$
end for

図 1. HPO で使用されるハイパーパラメーターを示す LARS の疑似コード

RN50 のハイパーパラメーター	シンボル	タイプ	制約
トレーニング・エポックの数	NTE	INT	正の整数
ウォームアップ・エポックの数	NWE	INT	正の整数
基本学習率	BLR	FLOAT	正の実数
重みの減衰	WD	FLOAT	0.0001 x 2^N、N は整数
運動量	MM	FLOAT	正の実数、グローバル・バッチ・サイズに依存

表 1. ハイパーパラメーターと MLPerf* の制約

大規模なトレーニングと HPO ワークフロー・プロセスの統合

図 2 は、オンプレミスまたはクラウドベースのトレーニング・クラスターの概念図です。AI ユーザーはこのクラスターにアクセスして、トレーニング・ジョブのバッチをコンピューティング・インフラストラクチャーに送信し、実行したジョブの精度（AC）と収束したエポック数（CE）を受け取ります。この情報は SigOpt に送られ、SigOpt は新しいハイパーパラメーターの提案をユーザーに返します。ユーザーは、目標精度に到達し、可能な限り低い収束エポックを見つけるという最適化目標が達成された時点で、ハイパーパラメーター検索を継続、調整、停止するように条件をプログラムできます。

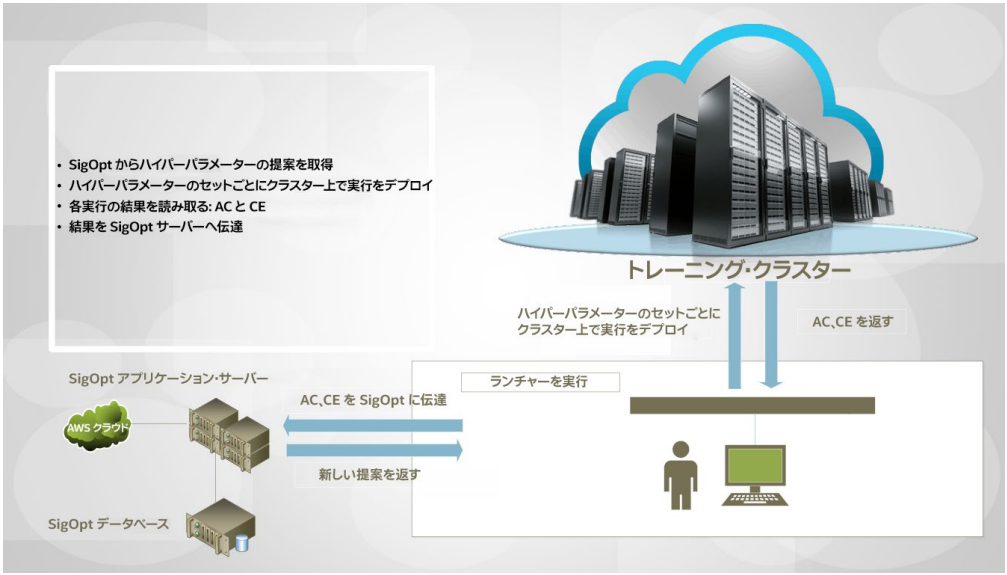


**図 2. SigOpt と AI ユーザーが HPO の進捗を制御する
トレーニング・クラスターでのトレーニング・ジョブの実行**

SigOpt を使用した HPO ワークフロー

図 3 は、SigOpt を使用した HPO ワークフローの現時点の実装です。以下の変数を定義して SigOpt を起動します。

- ハイパーパラメーターの初期境界
- 評価メトリック（AC など）のしきい値
- 実験予算（時間予算内で可能な実行回数)
- 計算リソース（使用する計算ノード数)

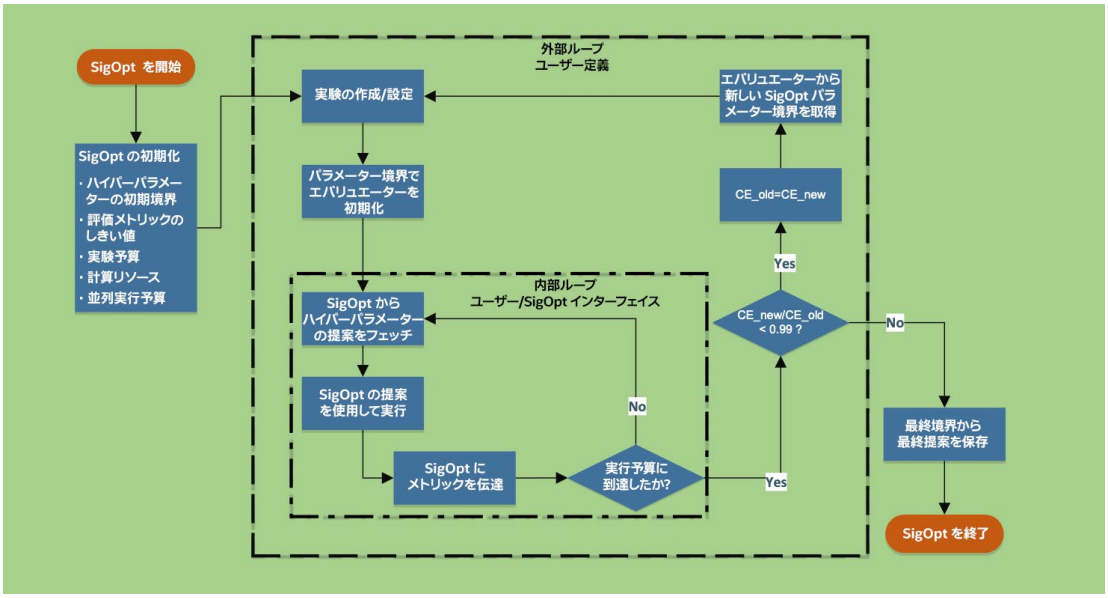


図 3. 目標精度を満たし、収束までのエポック数を最小にするハイパーパラメーターを見つける、SigOpt を使用した HPO ワークフロー

- 並列実行予算（許容される計算リソース内でハイパーパラメーター検索を高速化する同時実行数)

ワークフローは、特定の機能を持つビルディング・ブロックのセットである内部ループ(破線点線)と外部ループ(破線)の 2 つの入れ子構造のループで構成されています。SigOpt の起動後、新しい実験が作成および設定され、エバリュエーター・アルゴリズムがハイパーパラメーターの境界で初期化されます。そして、SigOpt から提供されたハイパーパラメーターの提案値で内部ループが開始されます。この提案値は、トレーニング・クラスター上でユーザーが DL モデルをトレーニングした際のメトリックを基に推奨された値です。内部ループは、事前に定義された予算に達すると処理を完了します。

次に、外側ループは、エバリュエーター・ビルディング・ブロックを使用して、SigOpt のハイパーパラメーターの提案を検証し、ユーザー条件に基づいて境界の絞り込みを開始します。これにより、SigOpt の提案に加えて、独自の最適化を追加できます。ここで使用したエバリュエーターは、次のセクションで説明する K 平均法の教師なしマシンラーニング（ML）クラスタリング手法を利用しています。

ハイパーパラメーターの境界を調整後、外部ループは別の反復の処理を開始するか、収束エポックに改善が見られない場合は停止します。そして、ハイパーパラメーター値の最終提案が保存され、評価が停止されます。

HPO ワークフローに独自のエバリュエーターを取り入れる

図 4 ～ 7 に K 平均法クラスタリングに基づくエバリュエーターの実装を示します。SigOpt から受け取ったすべてのデータについて、精度要件を満たすデータのみを保存し、さらに保存したデータの中から、収束エポックが良い 75% のデータポイントのみを残します（**図 4**）。次に、データポイントの提案部分（NTE、BLR、NWE、WD）に対して K 平均法分類を適用し、それぞれが少なくとも 10 個のデータポイントを持つクラスターに分割します（**図 5**）。そして、平均実行時間の最小値に基づいて、最もパフォーマンスの良いクラスターを選択します（**図 5** と**図 6** の青破線枠）。

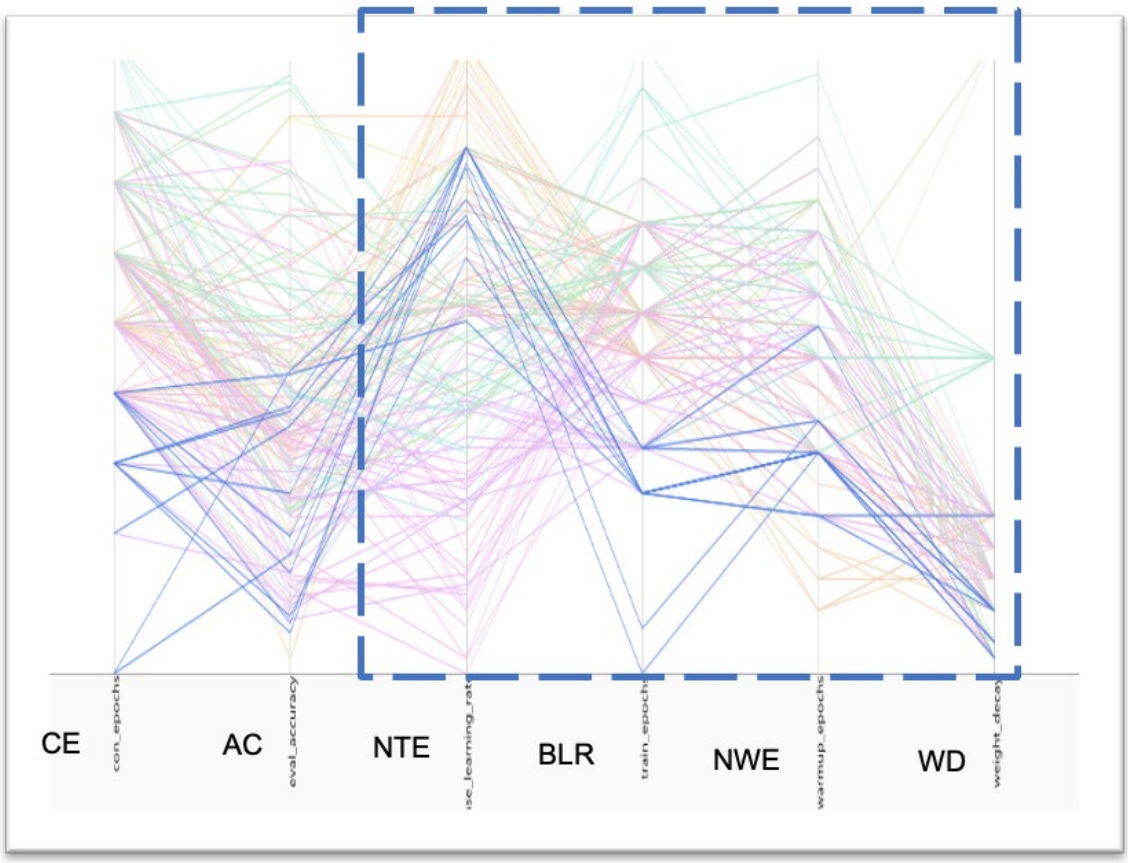


図 4. 測定ポイントのパラメーター空間に K 平均法クラスタリングを適用して個別のクラスターにグループ化

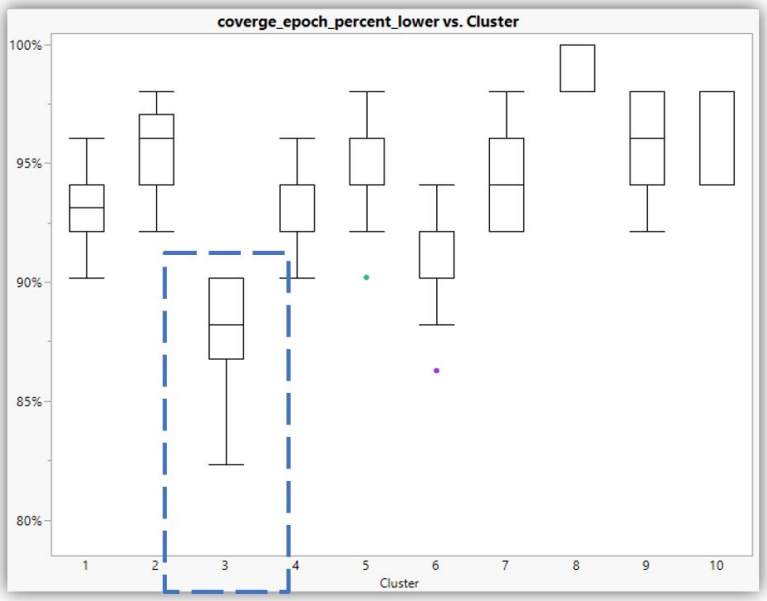


図 5. 与えられたクラスターの測定セットごとに平均収束エポックを計算。青色の点線で囲まれた平均収束エポックが最小のクラスターを選択。

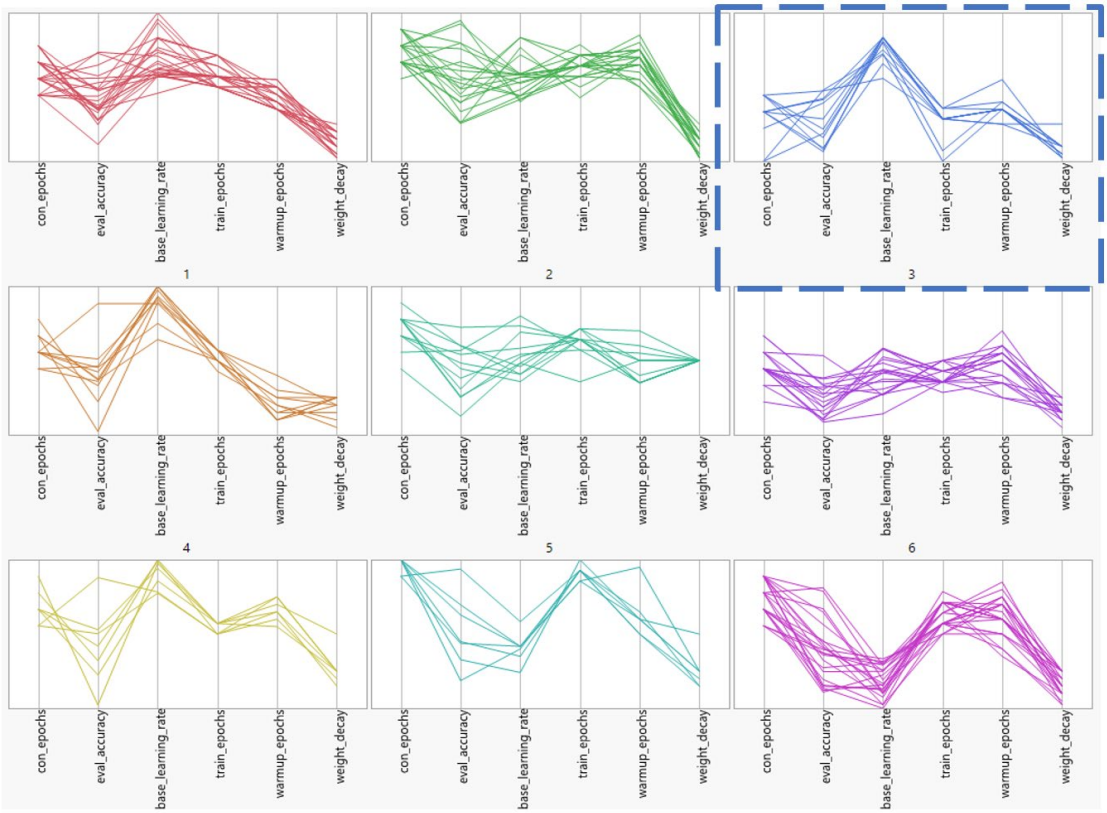


図 6. K 平均法アルゴリズムでクラスタリングした測定結果をプロット。青色の点線で囲まれているのが図 5 で選択されたクラスター。

与えられたハイパーパラメーターについて、(実行時間 / 収束エポックが最小である）最良のクラスターからの値の範囲（境界）を、SigOpt の事前定義された境界と比較します（**図 7**)。事前定義された境界の上限と下限のいずれかが、検出された最良のクラスターの値に近すぎる場合、エバリュエーターは対応する SigOpt 境界を調整し、新しいハイパーパラメーター値の検索を可能にします。一方、SigOpt 境界の上限と下限のいずれかが、検出された最良のクラスターの値から遠い場合、エバリュエーターは対応する SigOpt 境界を最良のクラスターのハイパーパラメーター値に近づけ、その値近辺での検索を可能にするように調整します。**図 7** では、経験に基づき、RN50 の係数に対して 0.1（10%）の調整を適用しています。新たに調整された SigOpt 境界で新しい実験を再設定します。学習率ハイパーパラメーターについて**図 8** でグラフに示すような調整を行うことで、ハイパーパラメーター値の検索が高速化されます。

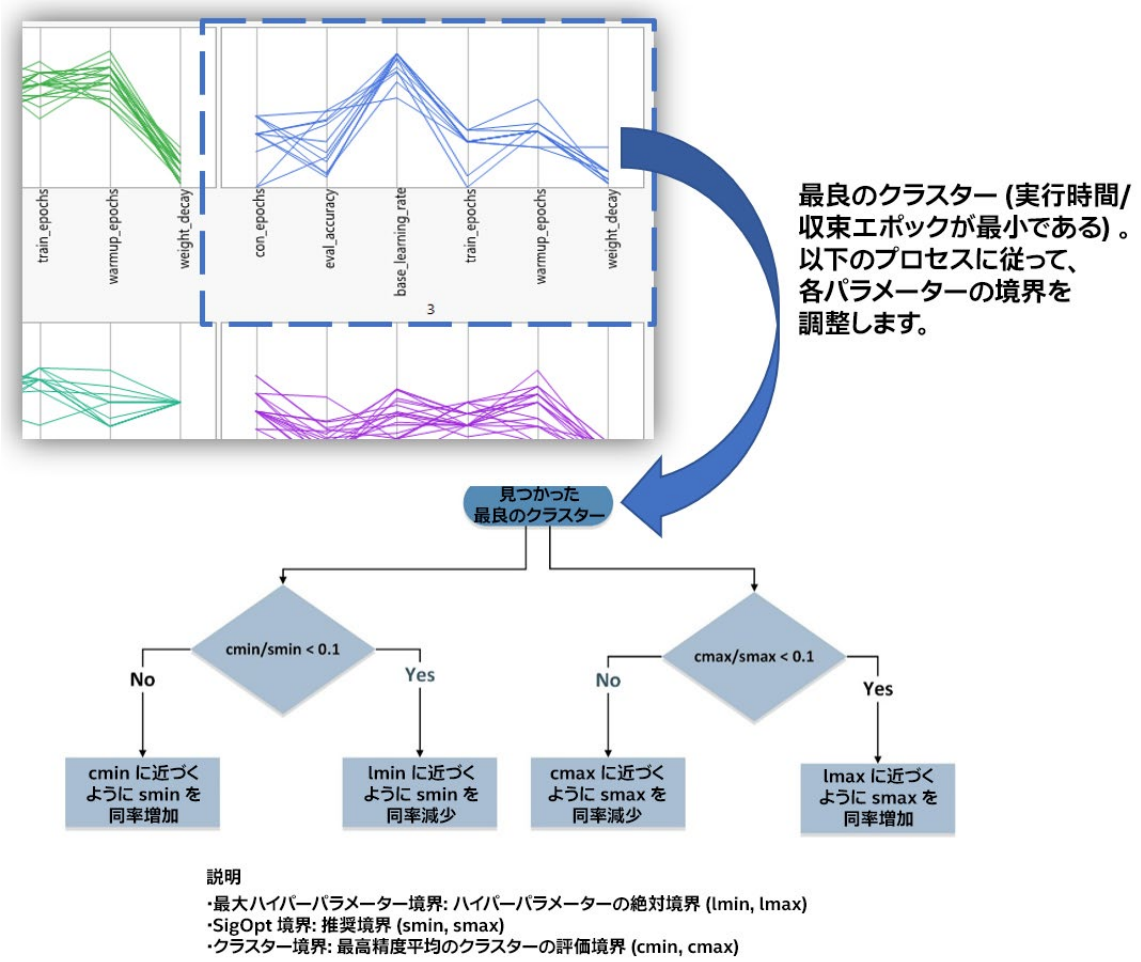


図 7. エバリュエーター・アルゴリズムの説明図

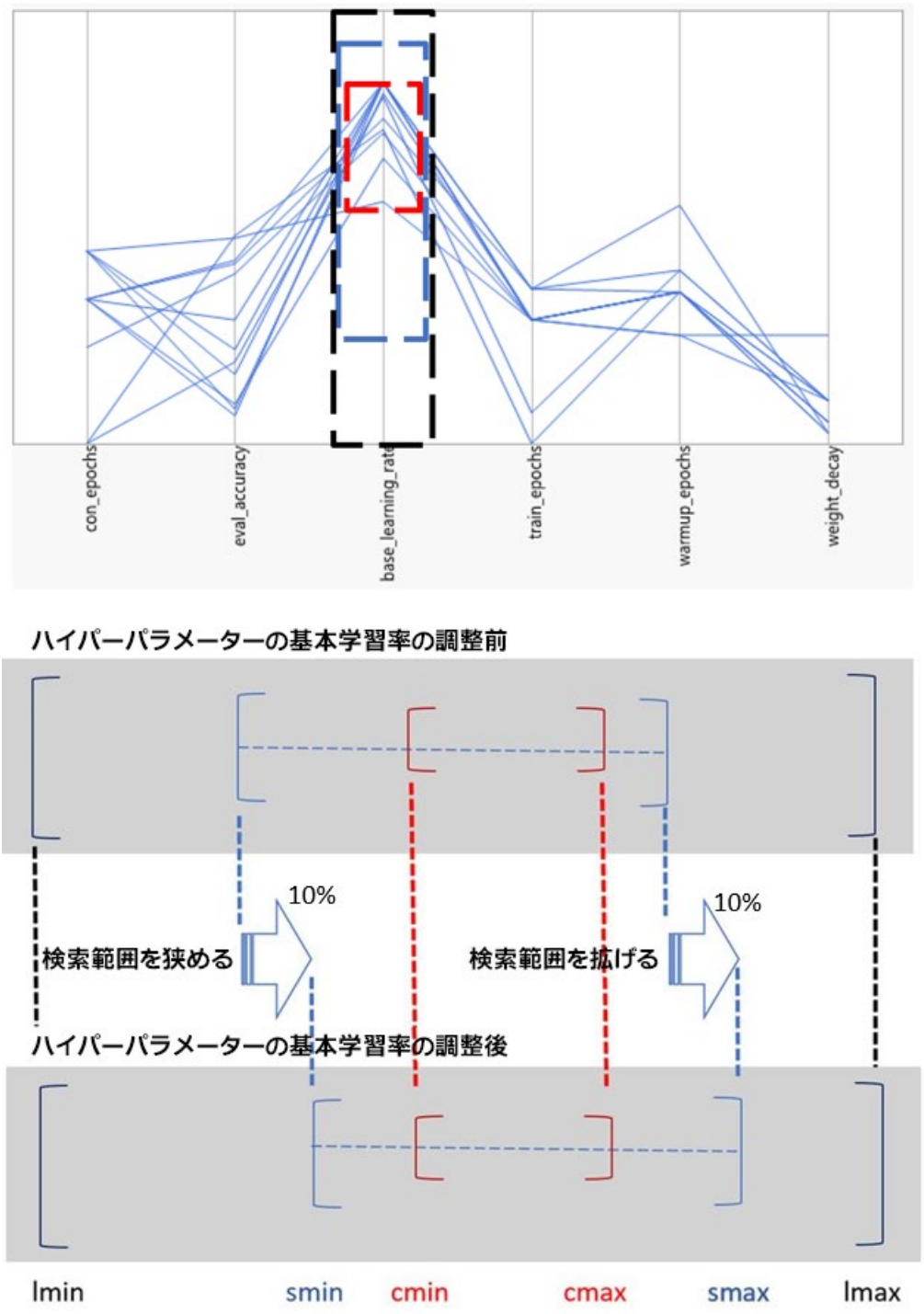


図 8. 最良のクラスターのハイパーパラメーターを基にハイパーパラメーター値を調整

ここでは、DL フレームワークの初期条件のランダム性のため、パラメーター値のランダムシードを変えて各実行を複数回実行し、各メトリックの平均値と標準偏差を SigOpt に報告しています。検索時間を短縮し、計算リソースを節約するため、検索プロセス中に各データポイントに対して 3 回の実行を行っています。しかし、実験の最後には、SigOpt によって発見された最適なポイントセットを選択し、より多くの実行回数（5 ～ 10 回）でそれらを検証しています。このプロセスにより、結果の安定性と再現性を確保することができます。また、与えられたグローバル・バッチ・サイズで最適化を実行することで、異なるローカル・バッチ・サイズ間で同じハイパーパラメーターを共有する、それぞれ異なる計算リソースを持つシナリオのセットを同時にカバーするため、計算リソースを節約できます。

SigOpt ダッシュボードのグラフ

SigOpt ダッシュボードに組込まれた並列座標、寄与率 / 感度、実験記録などのグラフは、実験中のハイパーパラメーターとその収束への影響の理解を深める一連の補完的なビューを提供します。RN50 では 400 回実行しました。並列座標プロット（**図 9**）は、SigOpt によって提案されたハイパーパラメーターと、これらの実行のメトリック（結果）との関係を示しています。灰色の線は目標とする評価精度（SigOpt ではこれをしきい値と呼ぶ）を満たさない実行を示し、青色の線はしきい値を満たした実行を示します。オレンジ色でハイライトされているのは最良の実行です（**図 9**、**図 10**）。並列座標プロットは、ハイパーパラメーター間の相関に加え、精度と収束までのエポック数を示しています。例えば、`base_learning_rate` と `weight_decay` は、収束したエポック数と逆相関していることが容易に分かります。

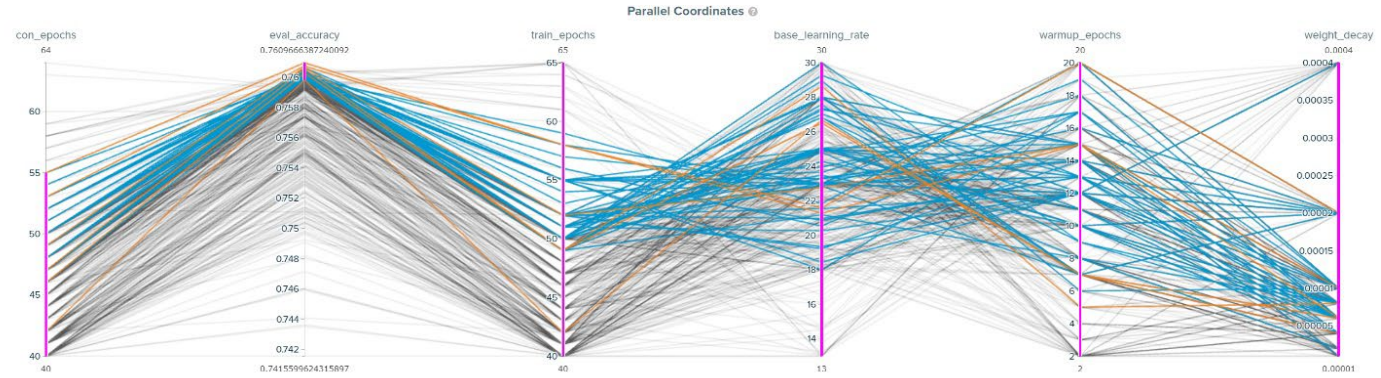


図 9. HPO と測定されたメトリック値の関係を示す RN50 並列座標プロット

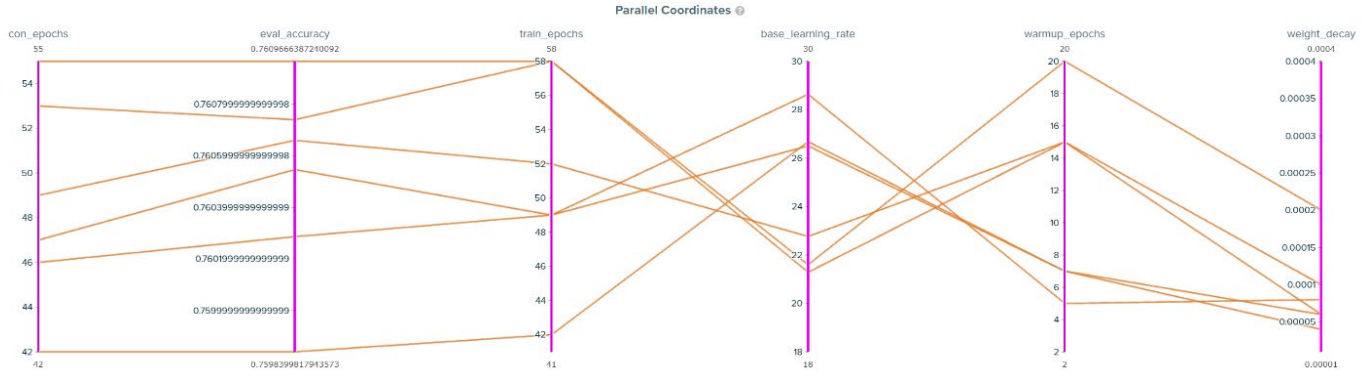


図 10. 最良の観測の RN50 並列座標

図 11 は、`train_epochs` と `con_epochs` の間に強い相関があることを示しています。`con_epochs` は、目標の `evaluation_accuracy` に到達した `train_epochs` の数であるため、強い相関が予想されます。また、`con_epochs` は `train_epochs` より約 1 ～ 3 エポック少ないことが分かります。

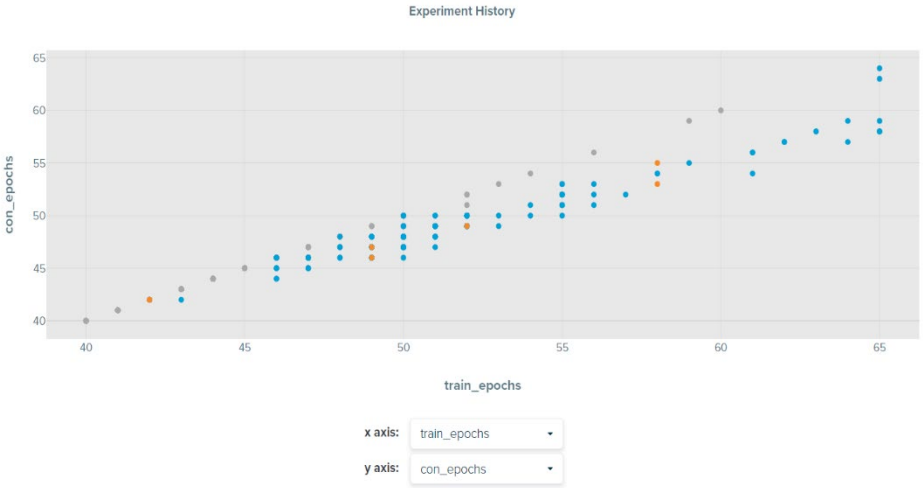


図 11. train_epochs と con_epochs の間の RN50 相関

図 12 は、各メトリック値に対する各ハイパーパラメーターの寄与率を示しています。オレンジ色の棒グラフは `eval_accuracy` に、青色の棒グラフは `con_epochs` にそれぞれ寄与していることを示しています。ランクのスケールは 0 ～ 1（100%）です。SigOpt では、これらの関係を測定するため、決定木回帰モデルを使用しています。この図では、直感的に予想されるように、`train_epochs` が `eval_accuracy` と `con_epochs` に最も影響を与えていることが分かります。また、`weight_decay` パラメーターは `con_epochs` の値のみに影響し、`base_learning_rate` は `eval_accuracy` のみに影響していることが分かります。

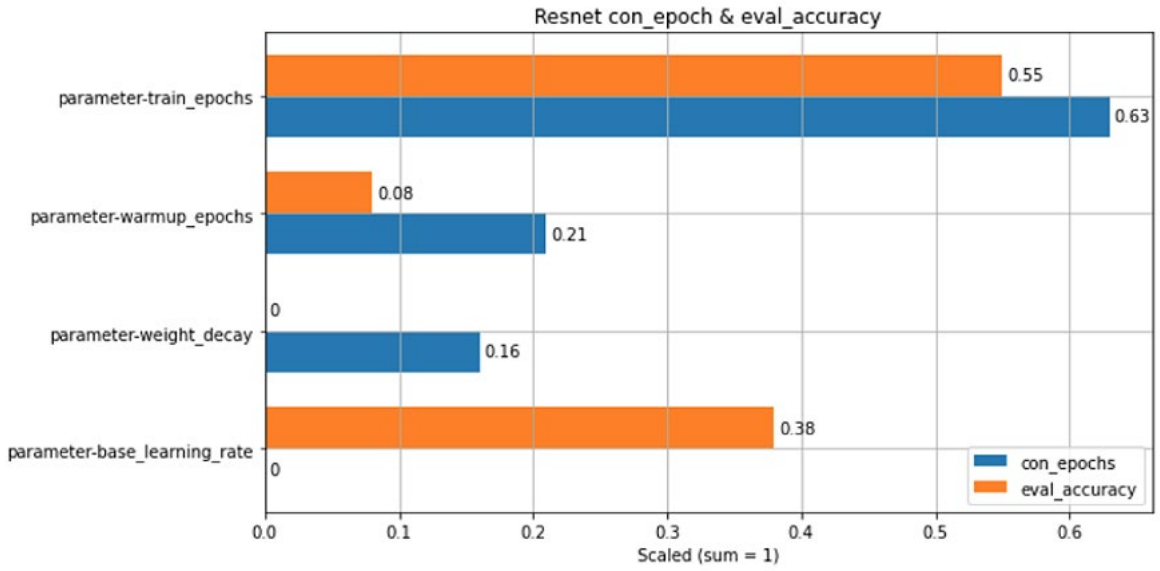


図 12. RN50 のメトリック結果に対するハイパーパラメーターの相対寄与率 (累積寄与率 100% に基づく)

図 13 は、すべてのメトリック値間の散布図を示す、SigOpt ダッシュボードのメイン・トラッキング・プロットです。このプロットは実験中に更新され、SigOpt オプティマイザーが `eval_accuracy` を向上させながら、`con_epochs` を減少させている様子を示しています。最良のポイントはオレンジ色でハイライトされ、最良の結果は赤色のボックスで表示されます。

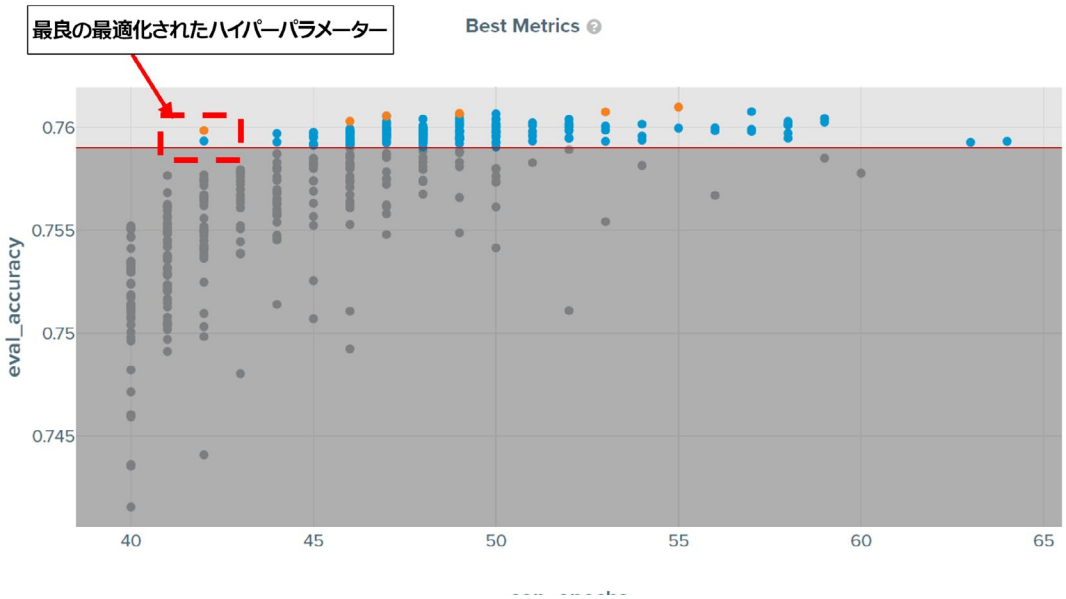


図 13. 最良のメトリックの相関

MLPerf* RN50 の SigOpt HPO の結果

最初に、両ワークロードの HPO で使用された値を示します。以下は、MLPerf* RN50 に渡されたハイパーパラメーター値と、HPO ワークフローの停止条件です。

ハイパーパラメーター	シンボル	タイプ	制約
トレーニング・エポックの数	NTE	INT	41 ~ 50
ウォームアップ・エポックの数	NWE	INT	2 ~ 15
基本学習率	BLR	FLOAT	18 ~ 30
重みの減衰	WD	FLOAT	1.0e-5 ~ 4.0e-4

メトリック	シンボル	タイプ	目標
評価精度	AC	FLOAT	0.759 より大きくなるようにする
収束エポック	CE	INT	最小化する

そして、MLPerf* RN50 の HPO で見つかった、グリッド検索に対する SigOpt の利点を紹介します。

- より少ない収束エポック（CE）のより優れたハイパーパラメーターを発見
- ベイズ最適化とグリッド検索
- グリッド検索と比較してエポックが 6% 減少、つまり学習時間が 6% 短縮
- より効率的なリソース利用

以下の表は、グリッド検索と SigOpt 検索による HPO にかかった計算リソースの利用時間と、達成された収束エポックの減少率です。

クラスター実行	グリッド検索 計算リソースの利用時間	SigOpt 検索 計算リソースの利用時間
Kubernetes* 実行	53,948	21,333
ベアメタル実行	31,464	0
合計実行	85,413	21,333

	グリッド検索	SigOpt 検索
収束エポックの減少率	28%	+6%（グリッド検索から）

SigOpt 検索は、グリッド検索の後に、すでに最適化された検索ポイントから開始されたため、計算リソースの利用時間はグリッド検索の 85,413 時間から 21,333 時間に短縮され、収束エポックの減少率はグリッド検索からさらに 6% 増加しています。

これらの結果から、AI 開発者が Habana* Gaudi* のコスト効率を活用し、SigOpt の HPO を利用して、モデル開発を加速させることを期待しています。

Modin で pandas* のワークフローをスケーリング

書き換え不要のスケーラブルなデータ・アナリティクス

Vasilij Litvinov インテル コーポレーション AI フレームワーク・エンジニア

[この記事は Anaconda.com（英語）に掲載された記事を許可を得て掲載したものです。]

AI やデータサイエンスが急速に進化することで、膨大な量のデータが利用できるようになり、日々複雑化し続けるアイデアや解決策を導き出すことができます。一方で、これらの進化は、価値抽出からシステム・エンジニアリングへと焦点が移っています。また、ハードウェアの性能は、人が適切な活用方法を学ぶよりも速いスピードで成長しています。

このような傾向から、「データエンジニア」という新しいポジションが登場したり、データ・サイエンティストがデータサイエンスの中核である調査ではなく、インフラストラクチャー関連の問題に対処する必要性が出ています。その主な理由の 1 つは、データ・サイエンティスト向けに最適化されたデータサイエンスとマシンラーニングのインフラストラクチャーが存在しないことです。すべてのデータ・サイエンティストがソフトウェア・エンジニアリングに精通しているわけではありません。この 2 つは、別々の、ときどき重複するスキルセットと考えることができます。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

データ・サイエンティストは習慣の生き物です。彼らは、pandas*、scikit-learn*、NumPy*、PyTorch* など、Python* データスタックで慣れ親しんだツールを好みます。しかし、これらのツールは、通常、並列処理やテラバイトのデータには適していません。

Anaconda とインテルは、データ・サイエンティストにとって最も重要な、使い慣れたソフトウェア・スタックと API をいかにスケーラブルかつ高速にするかという問題を解決するため協力しています。この記事では、Anaconda の「defaults」チャネル (および conda-forge) から利用可能な、インテル® oneAPI AI アナリティクス・ツールキット（AI キット）（英語）に含まれるインテル® ディストリビューションの Modin（英語）を紹介します。

pandas* だけでは十分でない理由

pandas* は業界標準ですが、本質的に多くの場合シングルスレッドで動作し、大規模なデータセットでは遅くなります。メモリーに収まらないデータセットでは動作しない場合もあります。この問題を解決するほかの選択肢（例えば、Dask、pySpark、vaex.io など）もありますが、どれも pandas* と完全に互換性のあるインターフェイスを提供していないため、ユーザーはワークロードを変更する必要があります。

Modin はエンドユーザーに何を提供するのでしょうか？ Modin は、「ツールはデータ・サイエンティストの役に立つべきであって、その逆はない」という考えに忠実であろうとしています。「`import pandas as pd`」文を「`import modin.pandas as pd`」文に変更するだけで、多くのユースケースでスケーラビリティーを向上できます。

Modin を使用する理由

「pandas* ワークフローを別のフレームワークに書き換える」必要をなくすことで、データインサイトの開発サイクルの短縮が可能になります（**図 1**）。

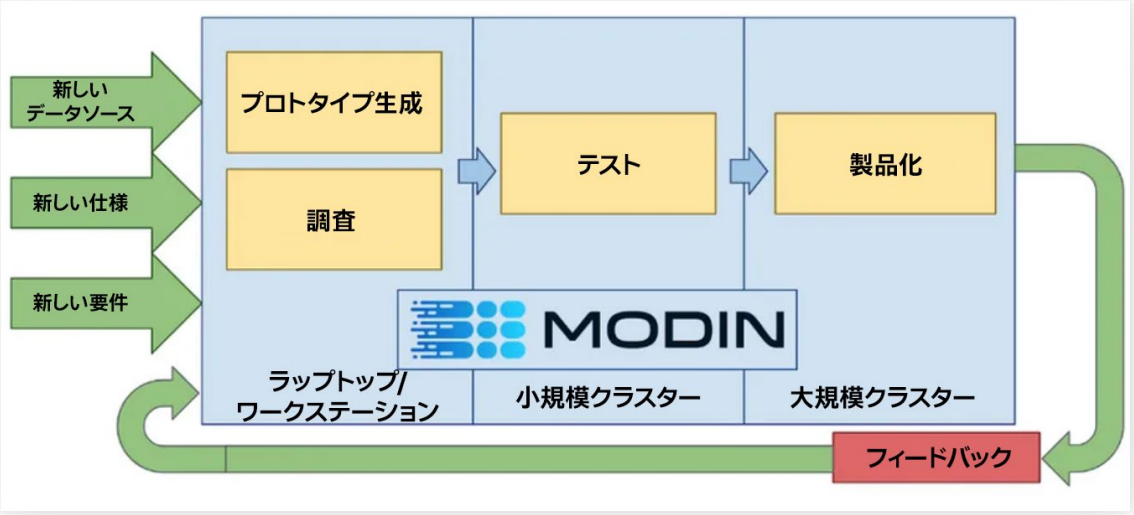


図 1. 継続的な開発サイクルでの Modin の使用

Modin は、DataFrame をグリッド分割することで、特定の操作をセル単位、列単位、行単位で並列分散して実行し、ハードウェアを効率良く活用します（**図 2**）。特定の操作では、OmniSci* エンジンとの実験的な統合を利用して、マルチコアのパワーをさらに活用することが可能です。

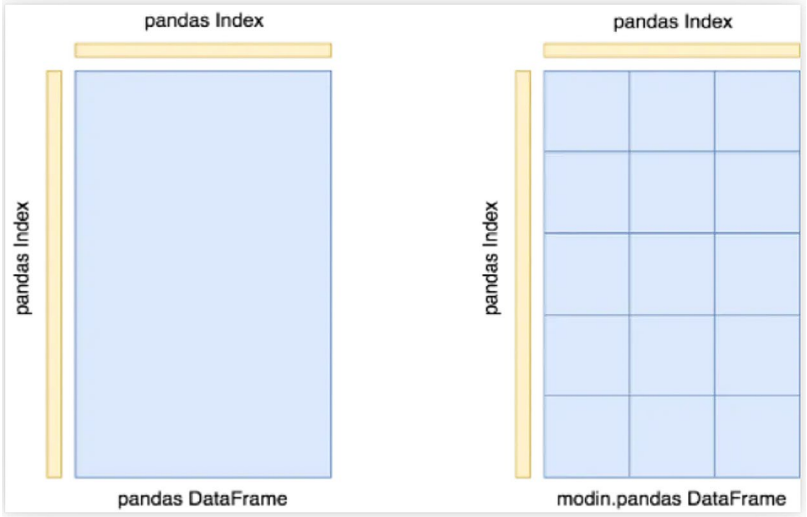


図 2. pandas* と Modin の DataFrame の比較

AI Kit や Anaconda defaults（または conda-forge）チャネルから Modin をインストールすることで、さらに高速な OmniSci* バックエンドも Modin で利用できるようになります。実験的な OmniSci* バックエンドは、以下の簡単なコード変更で有効になります。

```
import modin.config as cfg
cfg.Engine.put('native')
cfg.Backend.put('omnisci')
import modin.experimental.pandas as pd
```

実行結果

ベンチマークを見てみましょう。異なる Modin エンジンの詳細な比較は、http://modin.org/modin-bench (英語)にあるコミュニティーによって測定されたマイクロベンチマークを参照してください。Modin リポジトリーにコミットされたさまざまなデータサイエンス操作のパフォーマンスを追跡しています。

このアプローチのスケーラビリティーを実証するため、インテル® Xeon® Platinum 8368 プロセッサー・ベースのサーバー（下記のシステム構成を参照）上で、Modin を介して OmniSci* を使用して大規模なエンドツーエンドのワークロード（英語）を実行します（**図 3 ~ 5**）。

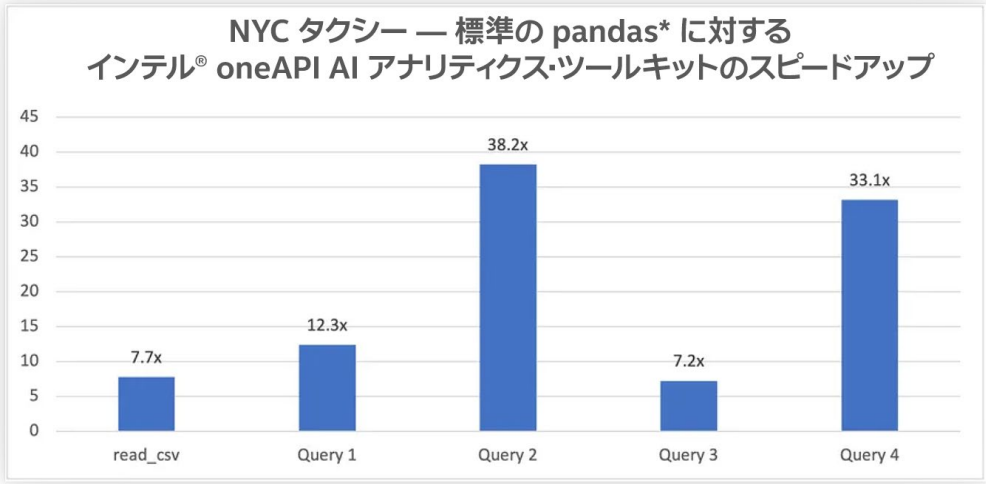


図 3. NYC タクシー・ワークロードの実行：レコード 2 億件、入力データセット 79.2GB

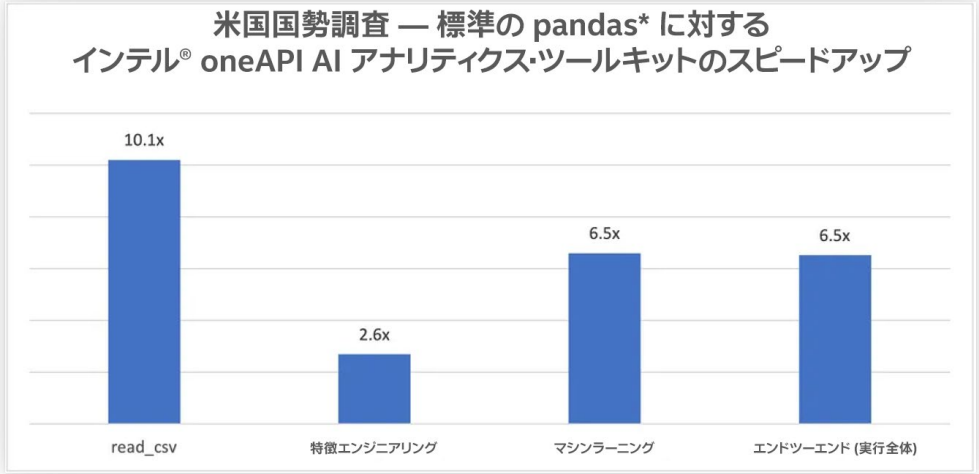


図 4. 米国国勢調査（英語）ワークロードの実行：レコード 2100 万件、入力データセット 2.1GB

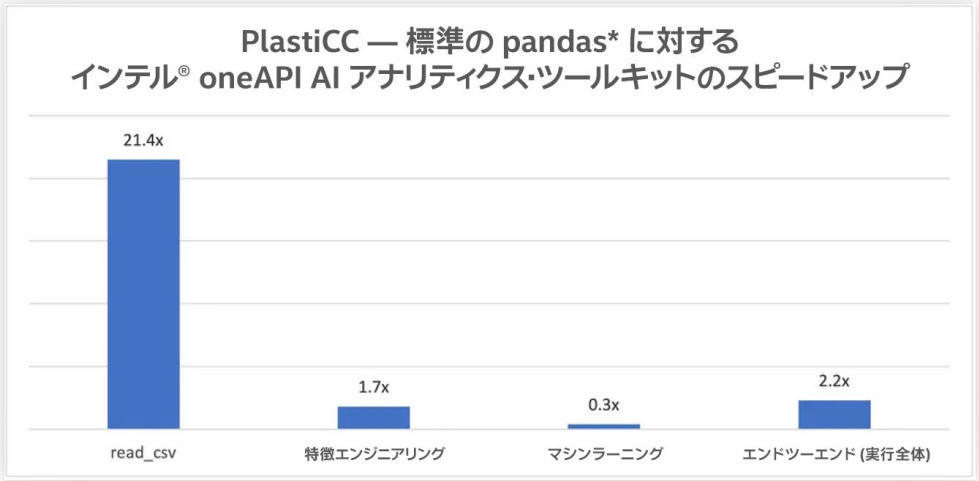


図 5. PlastiCC（英語）ワークロードの実行：レコード 4.6 億件、入力データセット 20GB

システム構成：ハードウェア：インテル® C620 ボード上の 2 基の第 3 世代インテル® Xeon® Platinum 8368 プロセッサー、合計 DDR4 メモリー 512GB（16 スロット /32GB/3200）、マイクロコード 0xd0002a0、インテル® ハイパースレッディング・テクノロジー有効、インテル® ターボ・ブースト・テクノロジー有効、インテル® SSD 960GB x1（OS ドライブ）、インテル® SSD 1.90TB x3（データドライブ）。ソフトウェア:CentOS* 7.9.2009 カーネル 3.10.0-1160.36.2.el7.x86_64、Python* 3.8.10、pandas* 1.3.2、Modin 0.10.2、OmniSci* DB 5.7.0、Docker* 20.10.8。2021 年 10 月 5 日現在のインテル社内での測定値。

追加情報

単一ノードでの実行では十分でない場合、Modin は Ray クラスター（英語）や Dask クラスター（英語）での分散実行をサポートしています。また、実験的な XGBoost との統合を使用することで、特別な設定なしに自動的に Ray ベースのクラスターを利用することも可能です。

関連情報

- インテル® oneAPI AI アナリティクス・ツールキットのインストール：
  `conda install intel-aikit -c intel`
- Modin のドキュメント（英語）
- Modin のソースと問題トラッカー（英語）
- Modin から OmniSci* バックエンドへの切り替え（英語）

Ray から Chronos へ

Ray 上に BigDL でエンドツーエンドの AI ユースケースを構築

Wesley Du、Junwei Deng、Kai Huang、Shan Yu、Shane Huang
インテル コーポレーション AI & アナリティクス担当ソリューション・アーキテクト

[この記事は anyscale.com（英語）に掲載された記事を許可を得て掲載したものです。]

Ray（英語）は、革新的な AI アプリケーションを開発するシンプルなユニバーサル API を備えたフレームワークです。BigDL（英語）は、分散したビッグデータ上にスケーラブルなエンドツーエンドの AI を構築するオープンソースのフレームワークです。Ray とそのネイティブ・ライブラリーを活用して、AutoML や自動時系列解析などの高度な AI ユースケースをサポートします。この記事では、BigDL のコア・コンポーネントの一部を紹介し、Ray を活用して基礎となるインフラストラクチャー（RayOnSpark、AutoML など）を構築し、これらが Chronos（英語）プロジェクトを使用した AI アプリケーションの構築にどのように役立つかを説明します。

RayOnSpark : Apache Spark* 上で Ray アプリケーションをシームレスに実行

Ray（英語）は、深層強化学習（英語）や自動マシンラーニング（AutoML）などの新しい AI アプリケーションを簡単に実行するためのオープンソースの分散型フレームワークです。BigDL は、RayOnSpark を利用して Ray をビッグデータの前処理パイプラインにシームレスに統合します。すでに、AutoML や Chronos などの特定分野において、複数の高度なエンドツーエンド AI アプリケーションの構築に使用されています。RayOnSpark は、ビッグデータ・クラスター（Apache Hadoop* や Kubernetes* クラスターなど）上の Apache Spark* 上で Ray プログラムを実行します。その結果、インメモリー DataFrame などのオブジェクトを Ray アプリケーションに直接ストリーミングし、高度な AI アプリケーションを実現することが可能です。RayOnSpark を使用することで、ユーザーは新しい革新的な AI アプリケーションを本番環境にある既存のビッグデータ・クラスター上で直接試すことができます。また、Ray アプリケーションをビッグデータ処理パイプラインにシームレスに統合し、インメモリー DataFrame 上で直接実行できるようになります。

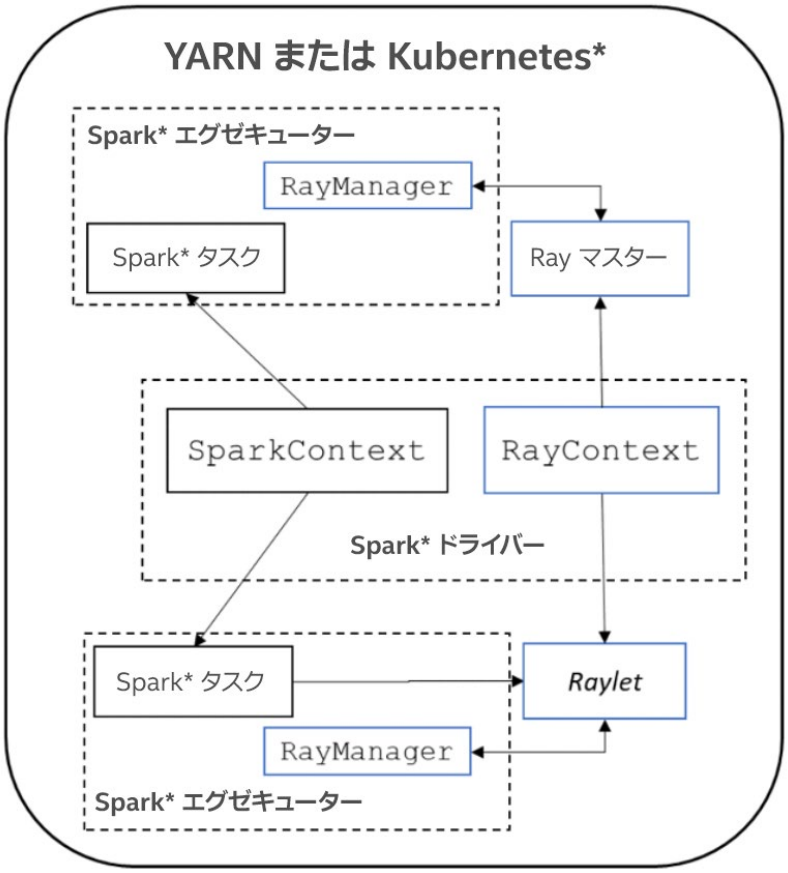


図 1. RayOnSpark アーキテクチャー

図 1 に RayOnSpark のアーキテクチャーを示します。Spark* の実装では、Spark* プログラムはドライバーノード上で実行し、Spark* ジョブを実行するためクラスター上で複数の Spark* エグゼキューターを起動する SparkContext オブジェクトを持つ SparkSession を作成します。RayOnSpark では、Spark* ドライバープログラムはさらに RayContext オブジェクトを作成し、同じクラスターで各 Spark* エグゼキューターとともに Ray プロセスを自動的に起動します。また、RayContext は各 Spark* エグゼキューター内に RayManager を作成し、Ray プロセスを管理します（例えば、プログラム終了時にプロセスを自動的にシャットダウンするなど）。**図 2** は、RayOnSpark を初期化した後、標準的な Spark* アプリケーション内で Ray コードを直接記述する方法を示しています。

```
import ray
from bigdl.orca import init_orca_context
from bigdl.orca.ray import RayContext

# Initialize SparkContext on the underlying cluster (e.g. the Hadoop/Yarn cluster)
sc = init_orca_context(cluster_mode="yarn", cores=...,memory=...,num_nodes=...)
# Initialize RayContext and launch Ray under the same cluster.
ray_ctx = RayContext(sc, object_store_memory=...,...)
ray_ctx.init()

@ray.remote
class Counter(object)
   def __init__(self):
      self.n = 0
   def increment(self):
      self.n += 1
      return self.n

# The Ray actors are created across the big data cluster
counters = [Counter.remote() for i in range(5)]
ray.get([c.increment.remote() for c in counters])
ray_ctx.stop()
sc.stop()
```

図 2. RayOnSpark のコード例

AutoML（orca.automl）：Ray Tune で AI アプリケーションを簡単にチューニング

マシンラーニング（ML）やディープラーニング（DL）モデルの精度やパフォーマンスなどを実現するには、ハイパーパラメーター最適化（HPO）が重要です。しかし、手作業による HPO は時間がかかり、十分に最適化されない可能性があります。一方、分散環境での HPO は、実装が困難な場合があります。BigDL は、データ・サイエンティストを支援するため、Ray Tune（英語）上に構築された AutoML 機能（orca.automl を使用）を提供しています。

orca.automl とは？

多くの場合、データ・サイエンティストは AI アプリケーションのプロトタイプ、デバッグ、チューニングをラップトップで行いたいと考えており、同じコードをそのままクラスターに移行できれば、エンドツーエンドの生産性が大幅に向上します。BigDL の Orca プロジェクトは、ユーザーがラップトップからクラスターにコードをシームレスにスケールするのを支援します。さらに、BigDL の orca.automl は、RayOnSpark と Ray Tune を活用し、AutoEstimator（英語）という分散型のハイパーパラメーター・チューニング API を提供します。Ray Tune はフレームワークに依存しないので、AutoEstimator は PyTorch* と TensorFlow* の両方のモデルに適しています。ユーザーは、ラップトップ、ローカルサーバー、Kubernetes* クラスター、Hadoop*/YARN クラスターなどで、一貫した方法でモデルをチューニングできます。

これらの機能により、多くの AI アプリケーションでは、BigDL の orca.automl を使用して検索空間（モデル、ハイパーパラメーターなどを含む）を自動検索できます。一例として、BigDL の orca.automl を使用して、XGBoost モデルを自動的に適合して最適化する AutoXGBoost（HPO 対応の XGBoost）を実装したところ、NVIDIA* A100 上の同様のソリューションと比較して、AutoXGBoost によるトレーニングは約 1.7 倍高速になり、より高い最終モデルの精度を達成しました。詳細は、「Analytics Zoo の AutoML を使用したスケーラブルな AutoXGBoost」（英語）を参照してください。『orca.automl ユーザーガイド』（英語）の設計情報と、『AutoXGBoost クイックスタート』（英語）や「任意のモデルの自動チューニング（英語）の演習も参考にしてください。

Chronos：Ray 上での AutoTS を使用した自動時系列解析の構築

別の例として、Chronos プロジェクト（英語）と呼ばれる自動時系列解析のフレームワークも構築しました。Chronos プロジェクトでは、自動解析中にハイパーパラメーターを調整するため orca.automl を利用しています。

Chronos が必要な理由

時系列（TS）解析は、現在、多くの実世界のアプリケーション（通信におけるネットワーク品質解析、データセンター運用のログ解析、高価値機器の予知保全など）で広く利用されており、その重要性はますます高まっています。正確な予測や検知は最も求められている課題であり、従来のアプローチでは大きな困難が伴うことが分かっています。DL 手法は、時系列の予測と検出をシーケンスモデル問題として捉えることが多く、近年、これらの問題に適用され、多くの成功を収めています。

一方、時系列の予測と検出を行う ML アプリケーションの構築は、手間がかかり、多くの知識を必要とします。ハイパーパラメーターの設定、前処理、特徴量エンジニアリングのすべてが、専用の DL モデルのボトルネックになる可能性があります。効率良く、強力で、使いやすい時系列解析ツールキットを提供するため、大規模時系列解析アプリケーションを構築するフレームワークである Chronos プロジェクトが立ち上げられました。Chronos は、Ray Tune、Ray Train（英語）、および RayOnSpark 上に構築されているため、AutoML や分散トレーニングを適用することが可能です。

Chronos アーキテクチャー

Chronos は、時系列予測、検出、シミュレーション用の 10 以上の組込み DL/ML モデル、および 70 以上のデータ処理 / 特徴エンジニアリング・ユーティリティーを備えています。ユーザーは、スタンドアロン・アルゴリズムやモデル（予測、検出、シミュレーター）を呼び出すことで、高い柔軟性を得ることができます。また、高度に統合されたスケーラブルな自動時系列ワークフロー（AutoTS）を使用することもできます。推論プロセスは、ONNX* ランタイム（英語）の統合など、さまざまな方法で最適化されています。

図 3 は、BigDL と Ray 上の Chronos アーキテクチャーを示したものです。ここでは、AutoTS コンポーネントに注目します。AutoTS フレームワークは、ハイパーパラメーター検索エンジン（RayOnSpark 上で実行）として Ray Tune を使用します。自動データ処理では、検索エンジンが予測タスクに最適なルックバック値を選択します。自動特徴抽出では、さまざまな特徴生成ツール（tsfresh など）により自動生成された特徴のセットから、検索エンジンが最適なサブセットを選択します。自動モデル化では、検索エンジンが隠れた次元、学習率などのハイパーパラメーターを検索します。

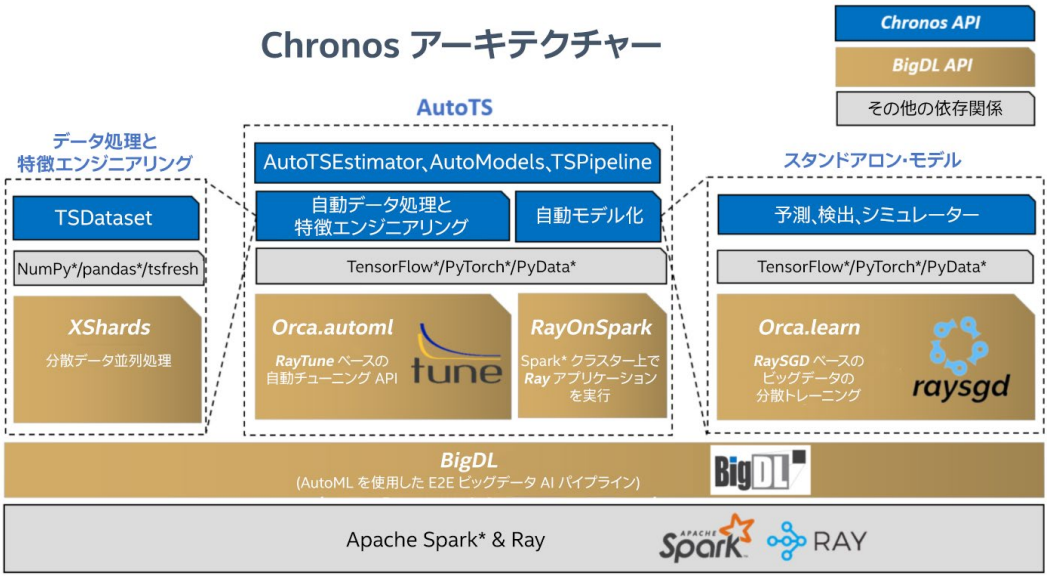


図 3. Chronos プロジェクトのアーキテクチャー

Chronos の AutoTS ワークフローの演習

次のコードは、Chronos のフレンドリーで高度に統合された AutoTS ワークフローを使用した時系列予測パイプラインのトレーニングと推論プロセスを示しています。

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from bigdl.chronos.data import TSDataset

# data initialization and split
df = pd.read_csv("table.csv")
tsdata_train, tsdata_val, tsdata_test = TSDataset.from_pandas(df,
                      dt_col="StartTime",
                      target_col="AvgRate",
                      with_split=True,
                      val_ratio=0.1)

# data processing and feature engineering
standard_scaler = StandardScaler()
for tsdata in [tsdata_train, tsdata_val, tsdata_test]:
   tsdata.gen_dt_feature()\
         .impute(mode="last")\
         .scale(standard_scaler, fit=(tsdata is tsdata_train))
```

このワークフローでは、`TSDataset` のシンプルでわかりやすい API を利用して、典型的な時系列処理（インピュテーション、スケーリングなど）と特徴量の生成を行います。

次に、モデル（組込みモデル名 / サードパーティー・モデルの場合はモデル作成関数)、ルックバック、層位を指定して、`AutoTSEstimator` を初期化できます。`AutoTSEstimator` は、Ray Tune 上で検索処理を実行します。各実行では、一度に複数の試行（それぞれ異なるハイパーパラメーターと特徴のサブセットの組み合わせ）を生成し、試行を Ray クラスターに分散させます。すべての試行が終了すると、ターゲットメトリックに従って、最適なハイパーパラメーターのセット、最適化されたモデル、およびデータ処理手順が取得され、それらを使用して `TSPipeline` が構成されます。

```
from bigdl.chronos.autots import AutoTSEstimator
import bigdl.orca.automl.hp as hp

# create a AutoTSEstimator
auto_estimator = AutoTSEstimator(model='tcn',
                                 past_seq_len=hp.randint(50,100),
                                 future_seq_len=1)

# fit on the AutoTSEstimator with HPO, auto feature, past_seq_len selector
ts_pipeline = auto_estimator.fit(data=tsdata_train,
                                 validation_data=tsdata_val)
```

`TSPipeline` は、予測、評価、インクリメンタルな適合に使用できます。

```
# predict/evaluate with TSPipeline
y_pred = ts_pipeline.predict(tsdata_test)
test_mse = ts_pipeline.evaluate(tsdata_test, metrics = ['mse'])
```

詳細は、『Chronos ユーザーガイド』（英語）を参照してください。

Chronos AutoTS を使用した 5G ネットワークの時系列解析

Chronos は、通信や AI の運用など、多くの分野で広く採用されています。Capgemini Engineering は、5G 媒体アクセス制御（MAC）で Chronos AutoML ワークフローと推論の最適化を活用し、「Intelligent to RAN Controller」ノードの一部として認識機能を実現しています。Capgemini Engineering のタスクでは、UE のモビリティー予測に Chronos を使用し、2 つのキーの KPI に基づいて MAC スケジューラーが効率良くリンク適合を行えるように支援しています。Chronos AutoTS を利用して、モデルを組込み TCN モデルに変更し、ルックバック値を大きくすることで、Capgemini Engineering は、AI の精度を 55% 向上させることに成功しました。詳細は、ホワイトペーパー（英語）を参照してください。

まとめ

この記事では、BigDL で Ray とそのライブラリーを活用して、ビッグデータ向けのスケーラブルな AI アプリケーションの構築（RayOnSpark を使用）、エンドツーエンドの AI 開発の生産性の向上（RayTune 上で AutoML を使用）、Chronos プロジェクトによる自動時系列解析などのドメイン固有の AI ユースケースの構築を行えることを紹介しました。BigDL は、ほかでも Ray を採用しています。例えば、Ray Train（英語）は、BigDL の Orca プロジェクトで、シングルノードの Python* ノートブックを大規模クラスター間でシームレスにスケールアウトするために使用されています。Ray 上に構築された AutoML 機能を活用する推薦システム、強化学習などのほかのユースケースも検討されています。

oneAPI
ヘテロジニアス・アプリケーションの開発を高速化
次のハードウェア・プラットフォーム向けにコードを書き直すのではなく、イノベーションに取り組む
インテル® oneAPI ツールキット >

intel software

THE PARALLEL UNIVERSE

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ー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ー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ーに関して、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ー依存の最適化は、インテル® マイクロプロセッサー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ー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/en-us/articles/optimization-notice#ja-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ー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。

JPN/2203/PDF/XL/SPI/ND