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]。

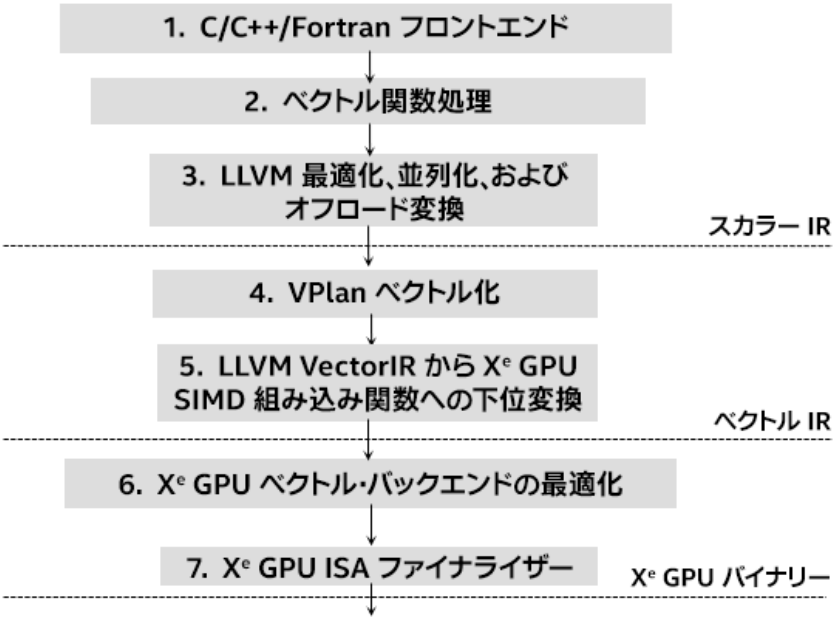

図 9. デバイスコンパイルの SIMD ベクトル化フレームワーク

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 >