インテル® コンパイラーを使用した OpenMP* GPU オフロードの基本

目的

- OpenMP* アプリケーションを GPU で実行する基本的な OpenMP* オフロード構造を理解します

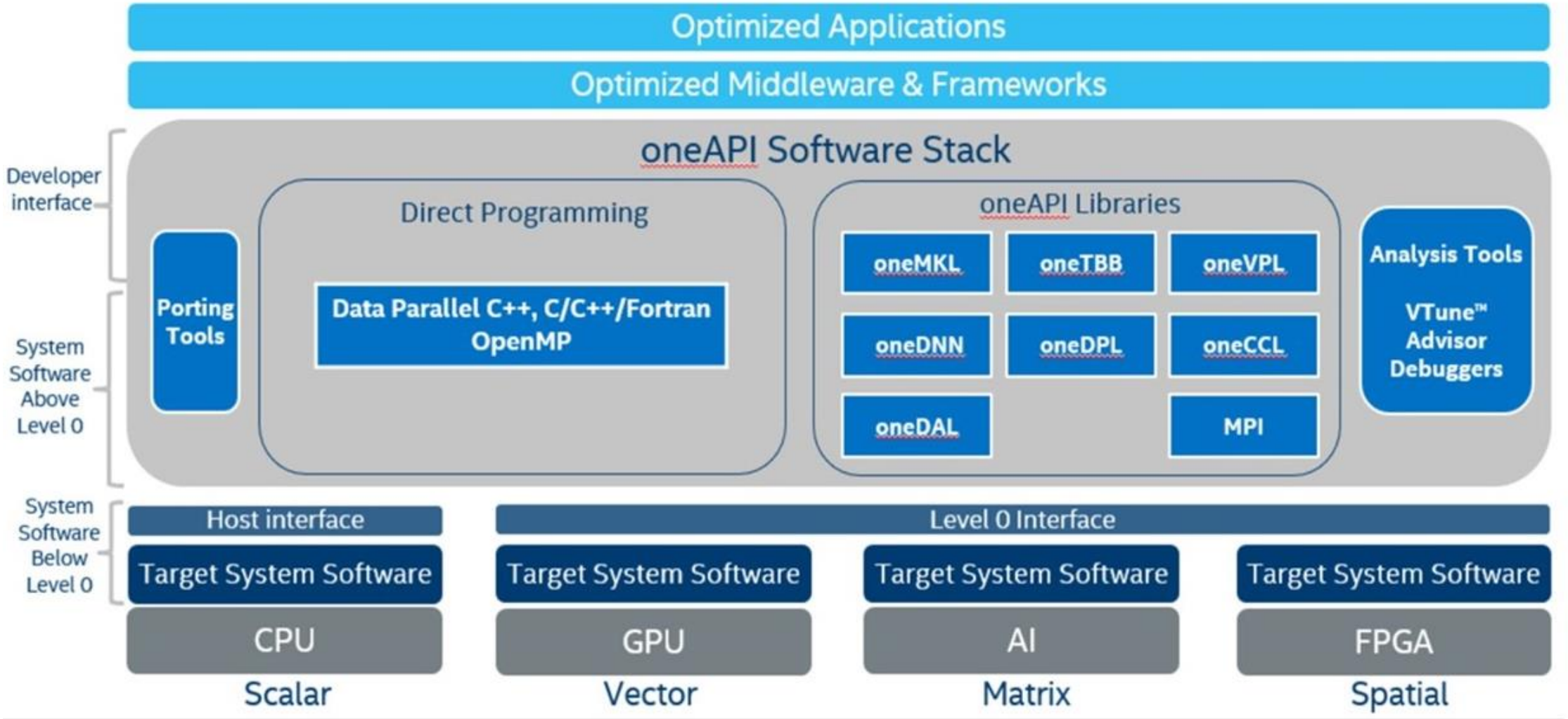
- 必要条件
  - CPU 上の Fortran、C または C++ で OpenMP* を使用する知識

OpenMP* 5.1 API シンタックス・クイック・リファレンス・カード:
https://www.isus.jp/products/c-compilers/openmp-ref-5-1-112001-released/

内容

- oneAPI ツールキット
- oneAPI と OpenMP* オフロード (新機能のまとめ)
- CPU 上の OpenMP* を確認
- OpenMP* オフロードの紹介
- デバイスデータを管理する構造
- 並列処理を利用する構造
- まとめ

HPC と AI 向け oneAPI ツールキット



HPC 向けのインテル® oneAPI ツール

インテル® oneAPI HPC ツールキット

スケーラブルで高速なアプリケーションを実現

ツールキットの概要

C++、Fortran、OpenMP* および MPI でエンタープライズ、クラウド、HPC、AI 向けのハイパフォーマンスでスケーラブルな並列アプリケーションを開発するためにインテル® oneAPI ベース・ツールキットに追加するツールキット

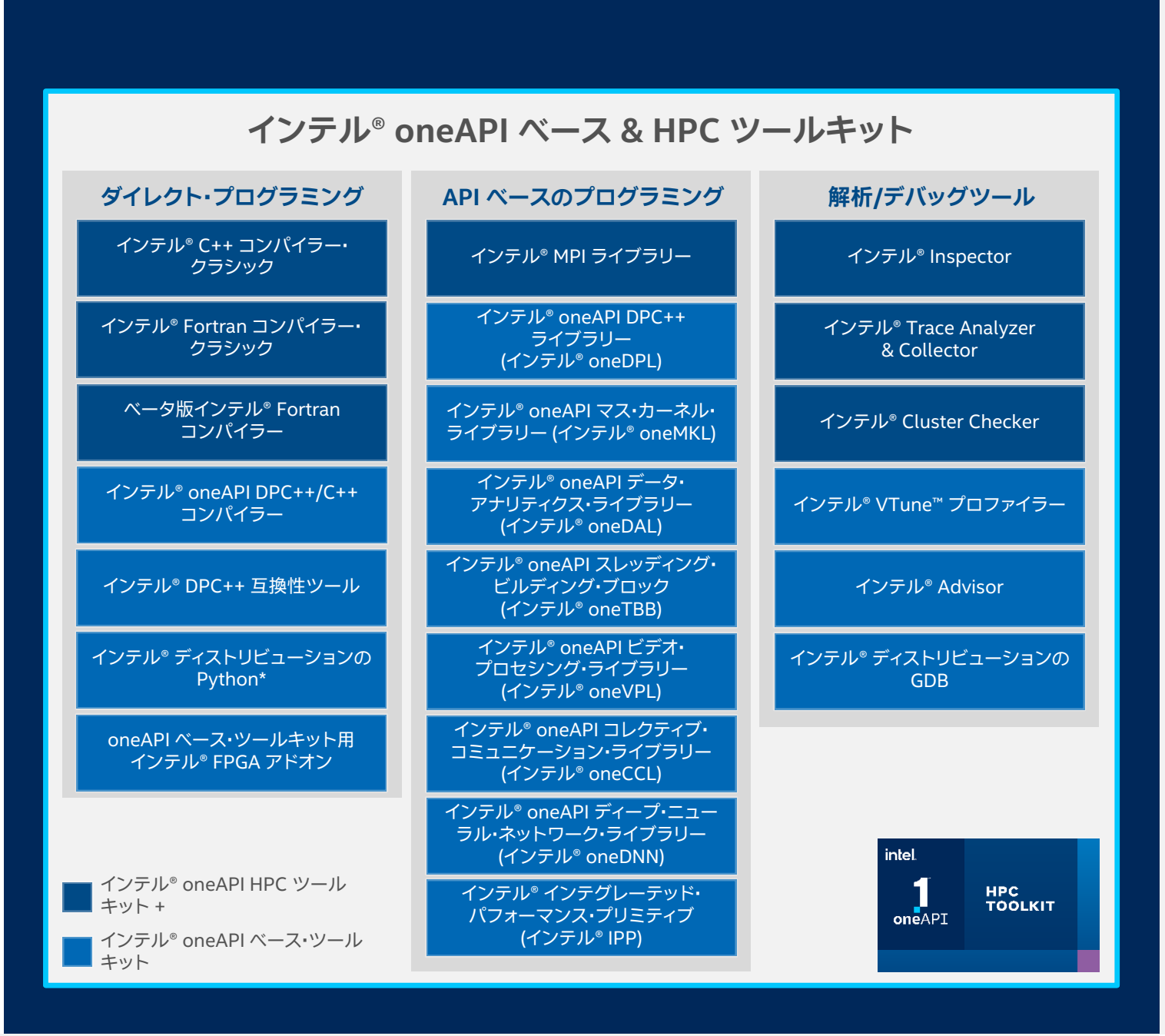
対象ユーザー

- OEM/ISV
- C++、Fortran、OpenMP*、MPI 開発者

ツールキットの重要性

- インテル® Xeon® プロセッサー、インテル® Core™ プロセッサーおよびインテル® アクセラレーターのパフォーマンスを向上
- 少ない労力で、高速かつ安定した、スケーラブルな業界標準の並列コードを作成

詳細: intel.com/oneAPI-HPCKit (英語)



oneAPI と OpenMP* オフロード

インテル® コンパイラーの OpenMP* サポート

- icc/icl と ifort では GPU 向けの OpenMP* 4.0/4.5 のオフロード機能はサポートされず、OpenMP* 5.0/5.1 にも対応していません
- OpenMP* 5.0/5.1/5.2 の機能は、clang/LLVM コミュニティーの作業を継続的に反映することで、icx/ifx で実装されます

インテル® コンパイラー	ドライバー	ターゲット	OpenMP* サポート	OpenMP* オフロードのサポート	インテル® oneAPI ツールキットに同梱
インテル® コンパイラー・クラッシック (ICC)	Icc	CPU	はい	いいえ	HOC、IoT
インテル® oneAPI DPC++/C++ コンパイラー (ICX)	dpcpp	CPU、GPUI、FPGA	はい	はい	ベース
	Icx	CPU、GPU	はい	はい	ベース
インテル® Fortran コンパイラー・クラシック (ifort)	Ifort	CPU	はい	いいえ	HPC
インテル® Fortran コンパイラー (ベータ) (IFX)	ifx	CPU、GPU	はい	はい	HPC

ICX と IFX コンパイラーの新機能

- JIT と AOT コンパイルのサポート
- OpenMP* 5.0/5.1 の機能
  - target variant dispatch (intel)、dispatch、declare variant (サブセット)、declare mapper (C/C++)、interop、loop、scope (C/C++)、allocate ディレクティブ/句、align 節/修飾子、task の nowaitm、target in_reduction 節、条件付き lastprivate 節、GPU 向けの SIMD 句
- 統合共有メモリー (USM)
- OpenMP* と DPC++ の構成の可用性
- マッパー宣言、クラスメンバーのファンクター、オフロード領域に関数ポインター
- 非同期オフロード
- Fortran 2008 と OpenMP* のオフロード
- 最適化レポート
- パフォーマンス最適化

https://www.isus.jp/products/c-compilers/openmp-features-and-extensions-supported-in-icx/

GPU 向けの OpenMP* SIMD

```
...
#pragma omp target enter data map (alloc:a[0:TOTAL_SIZE])
#pragma omp target enter data map (alloc:b[0:TOTAL_SIZE])
#pragma omp target enter data map (alloc:c[0:TOTAL_SIZE])
#pragma omp target update to(a[0:TOTAL_SIZE])
#pragma omp target update to(b[0:TOTAL_SIZE])

const int no_max_rep = 400;
double time = omp_get_wtime();
for (int irep = 0; irep < no_max_rep; ++irep){
   #pragma omp target teams distribute parallel for
   for (int isimd = 0; isimd < TOTAL_SIZE; isimd += SIMD_SIZE << 2){
      #pragma omp simd simdlen(32)
      for (int ilane = 0; ilane < SIMD_SIZE << 2; ++ilane){
         const int index = isimd + ilane;
         c[index] = a[index] + b[index];
      }
   }
}
time = omp_get_wtime() - time;
time = time/no_max_rep;
...
#pragma omp target update from(c[0:TOTAL_SIZE])
#pragma omp target exit data map (release:a[0:TOTAL_SIZE])
#pragma omp target exit data map (release:b[0:TOTAL_SIZE])
#pragma omp target exit data map (release:c[0:TOTAL_SIZE])
```

詳細な引数設定については、
oneAPI GPU 最適化ガイドを参照

統合共有メモリーのサポート

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#pragma omp requires unified_shared_memory
int main() {
   int deviceId = (omp_get_num_devices() > 0) ? omp_get_default_device() : omp_get_initial_device();
   int *a = (int *) omp_target_alloc(SIZE, deviceId);
   int *b = (int *) omp_target_alloc(SIZE, deviceId);
   for (int i = 0; i < SIZE; i++) {
      a[i] = i; b[i] = SIZE - i;
   }
   #pragma omp target parallel for
   for (int i = 0; i < SIZE; i++) {
      a[i] += b[i];
   }

   for (int i = 0; i < SIZE; i++) {
      if (a[i] != SIZE)
         printf("%s failed¥n", __func__); return EXIT_FAILURE;
   }
   omp_target_free (a, deviceId);
   omp_target_free (b, deviceId);
   printf("%s passed¥n", __func__)l
   return EXIT_SUCCESS;
}
```

マネージド・メモリー・アロケーターを介した USM サポート

インテルの拡張機能:
omp_target_alloc_host()
omp_target_alloc_shared()
omp_target_alloc_device()

DPC++/SYCL* と OpenMP* の構成の容易性

OpenMP* オフロードのコード

```
#include <CL/sycl.h>
#include <array>
#include <iostream>
// OpenMP Code
float computePi(unsigned N){
float Pi;
#pragma omp target map(from: Pi)
#pragma omp parallel for reduction(+: Pi)
   for (unsigned int i = 0; i < N; i++) {
      float T = (i + 0.5f) / N;
      Pi += 4.0f / (1.0 + T
   }
   return Pi / N;
}
// DPC++ Code
void iota(float *A, unsigned
   cl::sycl::range<1> R(N);
   cl::sycl::buffer<int, 1> X
   cl::sycl::queue().submit(
      auto Y = X.template get
      cgh.parallel_for<class
         Y[idx] = idx:
      });
   });
}
```

```
int main() {
   std::array<int, 2014u> V;
   float Pi;
#pragma omp parallel sections
   {
#pragma omp section
   iota(V.data(), V.size);
#pragma omp section
   Pi = computePi(8192u);
   }
```

Intel(r) oneAPI Tools

```
C> icx /Qiopenmp /Qopenmp-targets=spir64 -fsycl dpc_omp.cpp -nologo

C> set OMP_TARGET_OFFLOAD=mandatory

C> dpc_omp
SYCL: V[512] = 512
OpenMP: Pi = 3.14159
```

```
$ icpx -fiopenmp -fopenmp-targets=spir64 -fsycl compos.cpp -o run.y
$ OMP_TARGET_OFFLOAD=mandatory ./run.y
V[512] = 512
Pi = 3.14159
```

DPC++/SYCL* と OpenMP* USM の構成の容易性

```
#include <CL/sycl.h>
#include <omp.h>
#include <algorithm>
#include <iostream>

extern "C" void *omp_target_get_context(int);
#pragma omp requires unfined_shared_memory

int main() {
   const unsigned Size = 200;
   int D = omp_get_default_device();
   cl::sycl::queue Q (
   cl::sycl::context(static_cast<cl_context>(omp_target_get_context(D))),
   cl::sycl::gpu_selector());
   std::cout << "SYCL: Running on "
          << Q.get_device().
             get_info<cl::sycl::info::device::name>() << "¥n";
   if (!Q.get_device().
         get_info<cl::sycl::info::device::usm_shared_allocations>()) {
      std::cout << "Skipping test\n";
      return 0;
   }
   auto testOmp = [&](int* Data) {
      std::fill_n(Data, Size, -1);
      #pragma omp target parallel for device(D)
      for (unsigned I = 0; I < Size; ++I) {
         Data[I] = 100 + I;
      }
      return validate(Data);
   };
   auto testDpc = [&](int* Data) {
      std::fill_n(Data, Size, -1);
      Q.submit([&](cl::sycl::handler& cgh) {
         cgh.parallel_for<class K>
               (cl::sycl::range<1>(Size), [=](cl::sycl::id<1> I) {
            Data[I] = 100 + I;
         });
      }); Q.wait();
      return validate(Data);
   };

   int* ompMem = (int*)omp_target_alloc_shared(Size * sizeof(int), D);
   int* dpcMem = cl::sycl::malloc_shared<int>(Size, Q);
   std::cout << "SYCL and OMP memory: " << testDpc(ompMem) << "¥n";
   std::cout << "OMP and SYCL memory: " << testOmp(ompMem) << "¥n";
   std::cout << "OMP and SYCL memory: " << testOmp(dpcMem) << "¥n";
   std::cout << "SYCL and OMP SYCL memory: " << testDpc(dpcMem) << "¥n";
   omp_target_free(ompMem, D);
   cl::sycl::free(dpcMem, Q);
   return 0;
}
```

Intel(r) oneAPI Tools

```
C> dpc_omp_mem
Start
SYCL: Running on Intel(R) Graphics [0x3e98]
SYCL and OMP memory: passed
OMP and SYCL memory: passed
OMP and SYCL memory: passed
SYCL and OMP SYCL memory: passed
```

オフロード領域中のクラスのメンバー・ファンクター

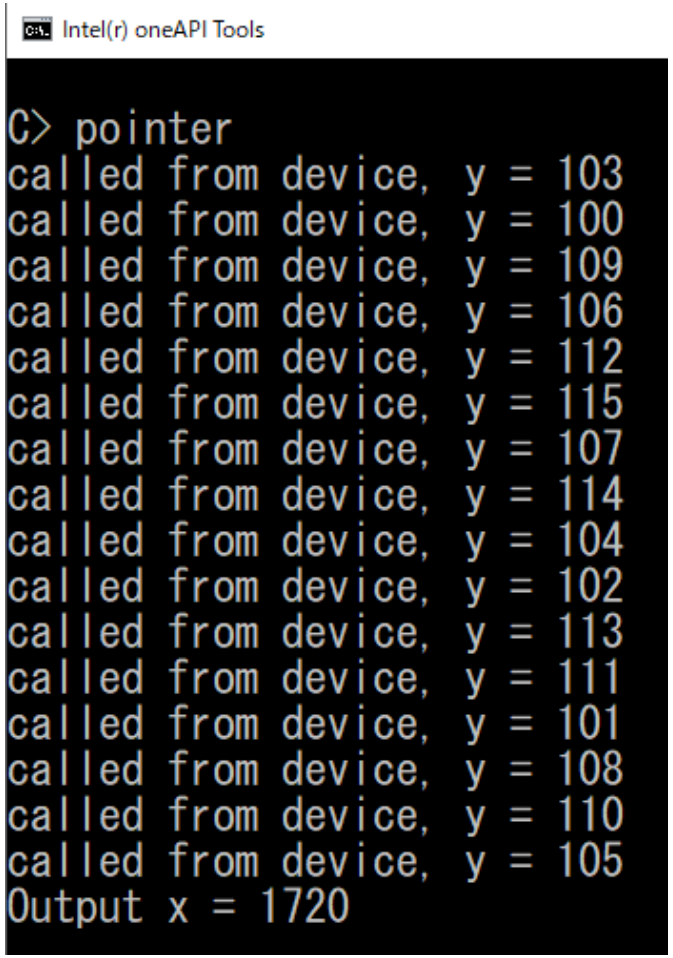
```
#include <bits/stdc++.h>
using namespace std;

// ファンクター
class inc
{
private:
   int num;
public:
   int(int n) : num(n) { }
   int operator () (int arr_num) const {
      return num + arr_num;
   }
};
```

```
int main() {
   int arr[] = {1, 2, 3, 4, 5};
   int n = sizeof(arr)/sizeof(arr[0]);
   int add5 = 5;
   inc a_inc(add5);

   #pragma omp target teams distribute ¥
         parallel for map(arr[0:n]) map(to: a_inc)
   for(int k = 0; k < n; k++) {
      arr[k] = arr[k] + a_inc(k);
   }

   for(int i = 0; i < n; i++)
      cout << arr[i] << " ";
   cout << "¥n" << "Done ......¥n";
}
```

また、仮想メンバー関数を **omp declare target** 構文内のクラスまたは **map** 節で使用されているオブジェクトで使用できるようになりました

オフロード領域での関数ポインターのサポート

```
#include <stdio.h>
#include <string.h>

#pragma omp begin declare target
int foo(int y) {
   printf("called from device, y = %d¥n", y);
   return y;
}
#pragma omp end declare target

int main() {
   int x = 0;
   int y = 100;
   int (*fptr)(int) = foo;
#pragma omp target teams distribute parallel for ¥
            firstprivate(y) reduction(+: x)
   for(int k = 0; k < 16; k++)
      x = x + fptr(y + k);
   printf("Output x = %d¥n", x);
}
```



ユーザー定義のマッパー (declare mapper)

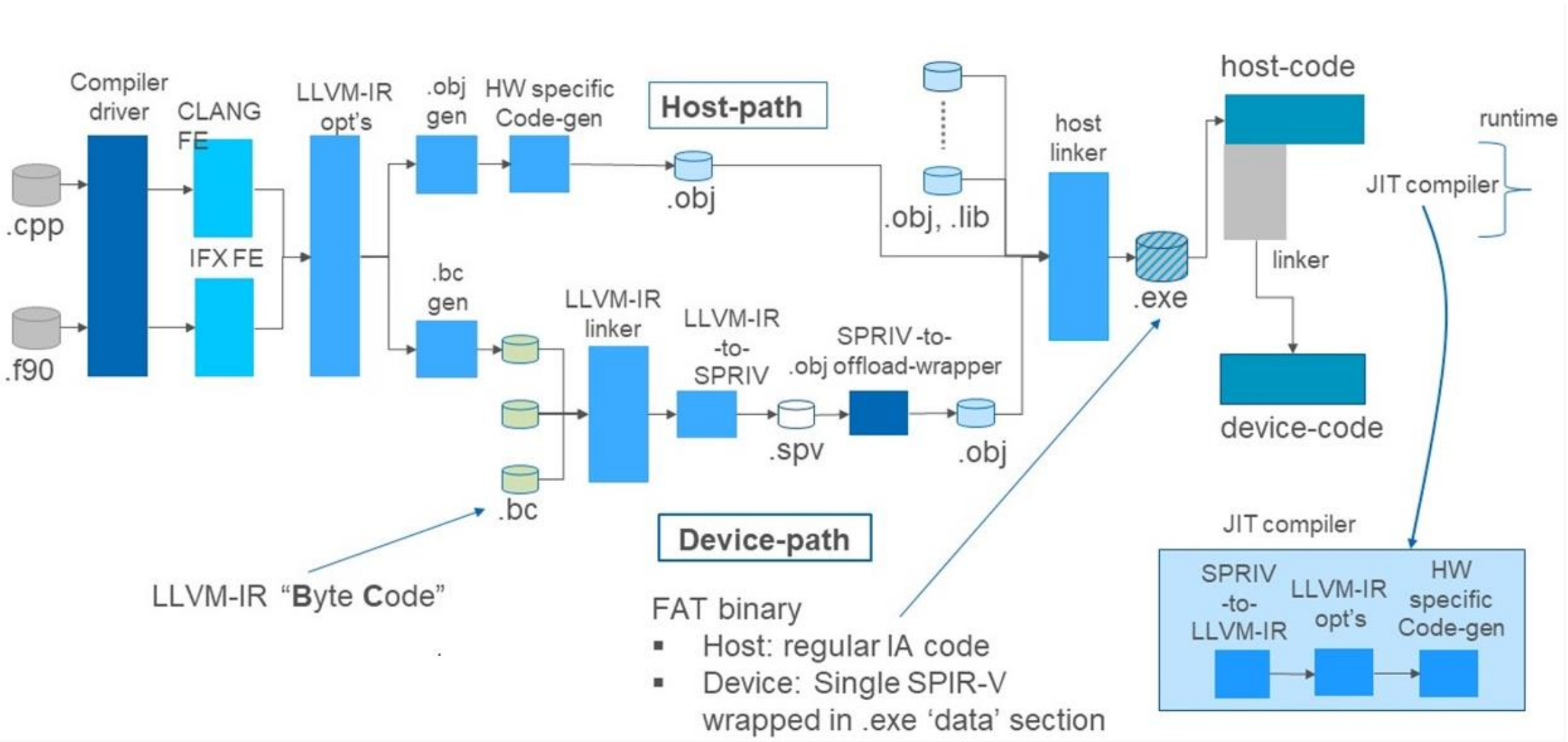
```
#include <stdio.h>
struct C { int num; int *arr; );

#pragma omp declare mapper(id: struct C c) map(c.num, c.arr[0:c.num])

void foo(int num, int *arr, int *arr_one){
   int i; struct C c; c.num = num; c.arr = arr;
   for(i=0; i<num; ++i) printf("%s%3d %s",(i==0?"In : ":""),c.arr[i],(i==num-1?"¥n":""));

   #pragma omp target map(mapper(id), tofrom: c)
   { int j; for(j = 0; j < c.num; ++j) c.arr[j] *= 2;}
   for(i=0; i<num; ++i) printf("%s%3d %s",(i==0?"Out : ":""),c.arr[i],(i==num-1?"¥n":""));

   struct C c_one; c_one.num=num; c_one.arr=arr_one;
   for(i=0; i<num; ++i) printf("%s%3d %s",(i==0?"In : ":""),c_one.arr[i],(i==num-1?"¥n":""));

   #pragma omp target map(mapper(id), tofrom: c_one)
   { int j; for(j = 0; j < c_one.num; ++j) c_one.arr[j] *= 2;}
   for(i=0; i<num; ++i) printf("%s%3d %s",(i==0?"Out : ":""),c_one.arr[i],(i==num-1?"¥n":""));
}
int main() {
   int arr4[] = {1,2,4,8}; int arr8[] = {1,2,4,8,16,32,64,128};
   int arr4one[] = {1,2,4,8}; int arr8one[] = {1,2,4,8,16,32,64,128};
   foo(sizeof(arr4)/sizeof(arr4[0]),arr4,arr4one); foo(sizeof(arr8)/sizeof(arr8[0]),arr8,arr8one);
   return 0;
}
```

非同期オフロードの例

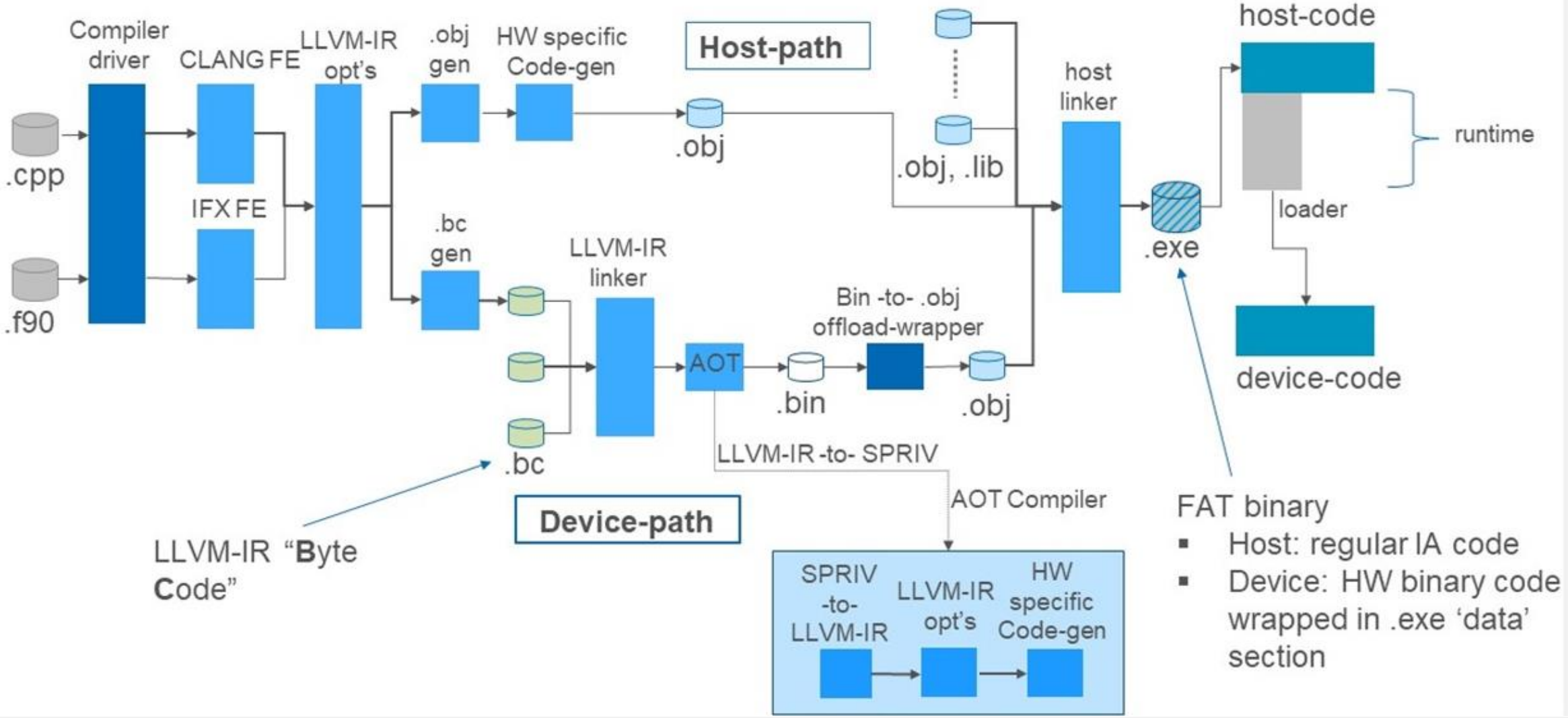
```
#include <stdio.h>
#include <omp.h>

int main() {
   int ret = 0;
#pragma omp target map(ret) nowait
   {
   for(int i = 0; i < 1000; i++)
      for(int j = 0; j < 1000; j++)
         ret--;
      if(ret <= 0)
          ret = 1;
      printf("Device ret = %d¥n", ret);
   }
   printf("Before explicit offload sync ret = %d¥n", ret);
#pragma omp taskwait
   printf("After explicit offload sync ret = %d¥n", ret);
   return 0;
}
```

最初のスレッドと同時実行されるフリー・エージェントの
ヘルパースレッドを有効にする機能が追加されました

ジャストインタイム (JIT) コンパイル



事前 (AOT) コンパイル


Compiler driver
CLANG FE
LLVM-IR opt's
.obj gen
HW specific Code-gen
Host-path
.cpp
.f90
IFX FE
.obj
.obj, .lib
host linker
host-code
runtime
loader
.exe
device-code
.bc gen
LLVM-IR linker
Bin -to- .obj offload-wrapper
AOT
.bin
.obj
.bc
LLVM-IR -to- SPRIV
AOT Compiler
Device-path
LLVM-IR "Byte Code"
SPRIV -to- LLVM-IR
LLVM-IR opt's
HW specific Code-gen
FAT binary
Host: regular IA code
Device: HW binary code wrapped in .exe 'data' section

CPU 上の OpenMP*

OpenMP* の概要

- C、C++、Fortran で共有メモリー・マルチプロセス・プログラミングをサポートするクロスプラットフォーム規格
  - マルチスレッド・アプリケーションを記述する API
  - 並列アプリケーション開発者向けのコンパイラー・ディレクティブとライブラリー・ルーチン
  - Fortran、C および C++ でマルチスレッド・プログラムの記述を大幅に簡素化
  - ベンダーやプラットフォームを超えた移植性
  - さまざまなタイプの並列処理をサポート

OpenMP* の歴史

- 1997: Fortran 向けのバージョン 1.0 をリリース
- 1998: C/C++ 向けのバージョン 1.0 をリリース
- 2002-2005: バージョン 2.0 - 2.5: Fortran と C/C++ 仕様を統合
- 2008: バージョン 3.0: タスク並列処理の導入
- 2013: バージョン 4.0: アクセラレーターと SIMD のサポート
- 2015: バージョン 4.5: 4.0 の強化、GPU サポートの改善
- 2018: バージョン 5.0: C11/C++17/Fortran 2008 のサポート
- 2020: バージョン 5.1:
  https://www.isus.jp/products/c-compilers/openmp51_tr9_ja/

OpenMP* の歴史

- 1997: Fortran 向けのバージョン 1.0 をリ
- 1998: C/C++ 向けのバージョン 1.0 をリ
- 2002-2005: バージョン 2.0 - 2.5: Fortra
- 2008: バージョン 3.0: タスク並列処理の
- 2013: バージョン 4.0: アクセラレーターと
- 2015: バージョン 4.5: 4.0 の強化、GPU
- 2018: バージョン 5.0: C11/C++17/Fortr
- 2020: バージョン 5.1:
  https://www.isus.jp/products/c-compilers/openmp51_tr9_ja/

OpenMP* スレッド

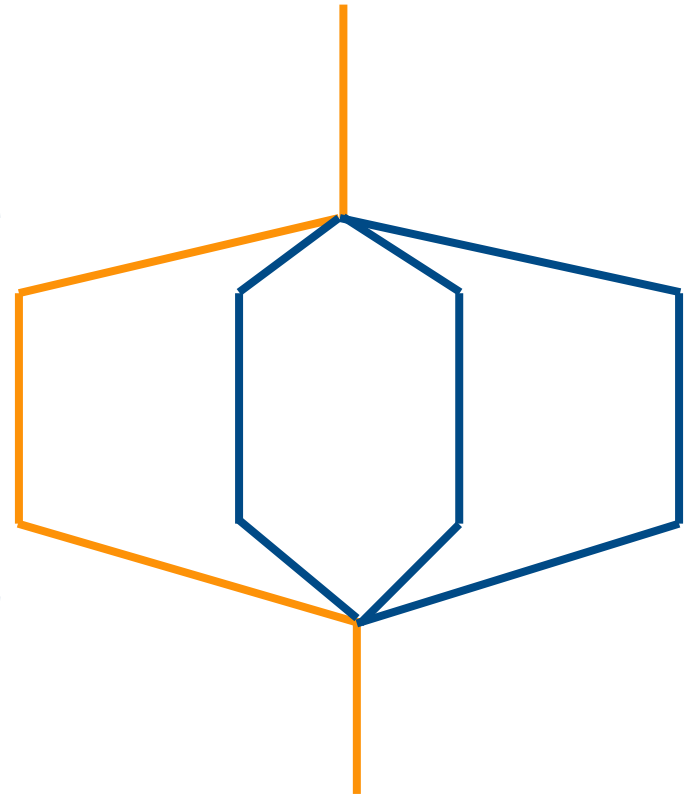
- **parallel** 構造でスレッドを生成します

```
#include <omp.h>

void saxpy()
{
   float a, x[ARRAY_SZ], y[ARRAY_SZ];
  #pragma omp parallel
  {
    int id=omp_get_thread_num();
    int nthrs=omp_get_num_threads();
    for (int i=id; i < ARRAY_SZ; i+=nthrs) {
       y[i] = a * x[i] + y[i];
    }
  }
}
```

並列領域

スレッドのチームを生成

各スレッドは同じコードを重複して実行



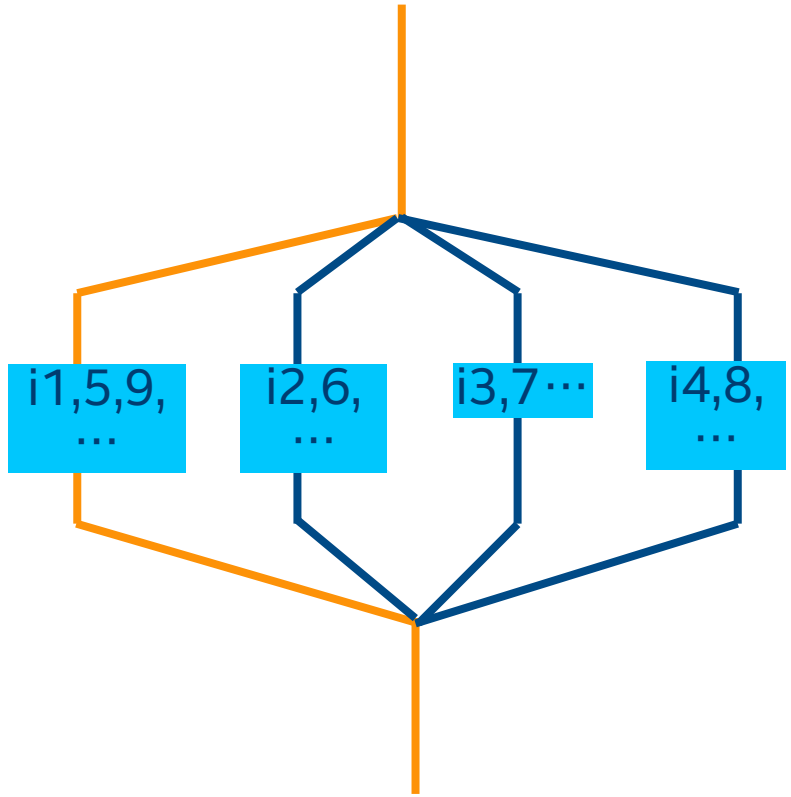
ループ

- ワークシェアに for/do ループを使用

```
#include <omp.h>

void saxpy()
{
   float a, x[ARRAY_SZ], y[ARRAY_SZ];
  #pragma omp parallel
  {
  #pragma omp for
      for (int i=0; i < ARRAY_SZ; i++) {
         y[i] = a * x[i] + y[i];
      }
  }
}
```

ワークシェア

ループ反復の
実行をスレッド
間で分散します



基本的な例

C/C++

```
#include <omp.h>

…
#pragma omp parallel for reduction (+:sum)
{
   for (int i=0; i<ARRAY_SZ; i++) {
      sum += x[i];
   }
}
…
```

Fortran

```
program main
   use omp_lib
   …
   !$omp parallel do reduction (+:total)
   do i=0,ARRAY_SZ
      total = total + x(i)
   end do
   !$omp end parallel do
…
end program main
```

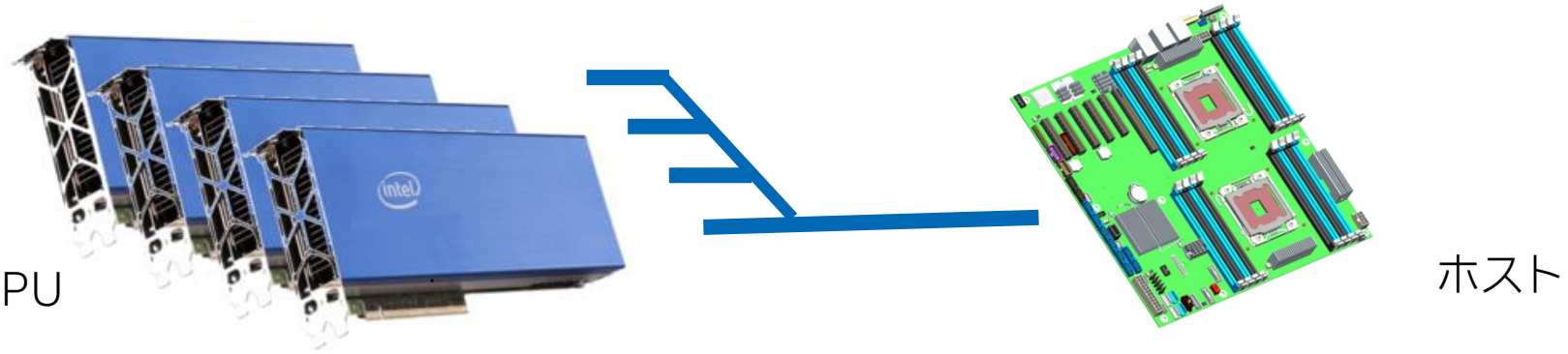
その他の注目すべき OpenMP* 構造

- `sections/section`
  - コードブロック (セクション) を既存のスレッドに分散します
- `task`
  - スレッドで実行する独立したワーク単位 (コード、データ、内部制御変数) を生成します
- `simd`
  - ループ反復を SIMD 命令を使用して実行することを指定します
    - コンパイラーはベクトルの依存関係を無視します

OpenMP* オフロードについて

OpenMP* デバイスモデル

- OpenMP 4.0+ はアクセラレーター/コプロセッサー (デバイス) をサポートします
  - GPU 固有ではありません
- デバイスモデル:
  - 1 つのホスト
  - 同種の複数のアクセラレーター/コプロセッサー



OpenMP* オフロードのコンパイラー・サポート

- OpenMP* オフロードはインテル® oneAPI HPC ツールキットでサポートされます
  - OpenMP* 4.5 (`-fiopenmp`) と OpenMP* 4.5 オフロード (`-fopenmp-targets=spir64`) のサポートを有効にする必要があります
  - インテル® oneAPI C++ コンパイラー

```
icx -fiopenmp -fopenmp-targets=spir64 <source>.c

icpx -fiopenmp -fopenmp-targets=spir64 <source>.cpp
```

  - インテル® Fortran コンパイラー

```
ifx -fiopenmp -fopenmp-targets=spir64 <source>.f90
```

デバイス向けの OpenMP* 4.0 構造

- **target** 構造は、ホストからデバイスへ制御とデータを転送します
- 構文 (C/C++)

```
#pragma omp target [節[[,] 節],…]
    構造化ブロック
```

- 構文 (Fortran)

```
!$omp target [節[[,] 節],…]
    構造化ブロック
!$omp end target
```

- 節

device(スカラー整数式)
map([{alloc | to | from | tofrom}:] リスト)
if(スカラー式), **nowait, depend**

実行モデル

- **`target`** 構造は、制御フローをターゲットデバイスへ転送します
  - 制御の転送はシーケンシャルで同期されます
  - 転送節はデータフローの方向を制御します
  - 配列表記は配列のレングスを指定するために使用されます

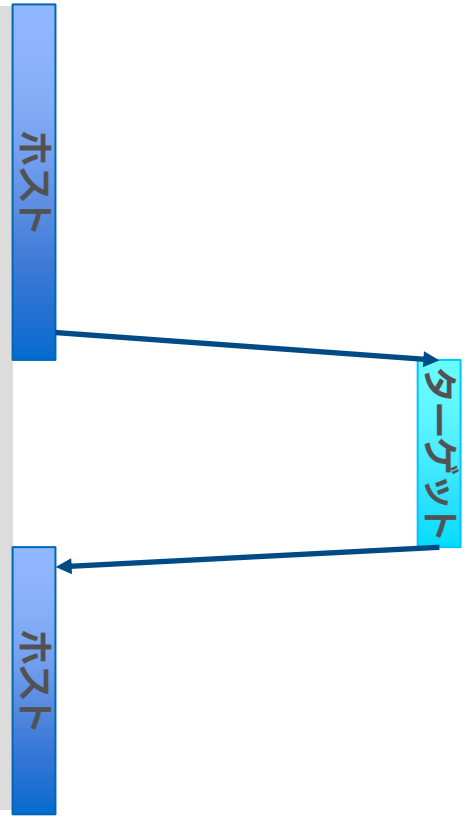
ターゲット領域の例: saxpy

シーケンシャルな
ホストコード

ターゲット領域

シーケンシャルな
ホストコード

```
void saxpy() {
    float a, x[ARRAY_SZ], y[ARRAY_SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target
    for (int i = 0; i < ARRAY_SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf¥n", t);
}
```




```
icx -fiopenmp -fopenmp-targets=spir64 -o saxpy saxpy.c
```

device 節

- 複数デバイス環境でオフロードするデバイスを指定します

```
#pragma omp target device(1)
```

  - 整数値のデバイス番号
    - 割り当ては実装依存です
    - 通常は 0 で始まり 1 つずつ増加します
  - **target**、**target data**、
    target enter/exit data、
    target update ディレクティブで指定できます

ターゲット領域で呼び出し可能な関数

- **declare target** 構造は、ターゲットデバイス向けの関数/サブルーチンを生成します
  - デフォルトでホストとターゲットで実行する両方の関数がコンパイルされます

```
#pragma omp declare target
int devicefunc(){
…
}
#pragma omp end declare target

#pragma omp target
{
        result = devicefunc();
}
```

```
subroutine devicefunc()
!$omp declare target device_type(device)
…
end subroutine

program main
!$omp target
        call devicefunc()
!$omp end target
end program
```

オプションの device_type は、ホストまたはデバイスでの実行を指定します

デバイスが指定される場合、常に利用可能でなければなりません

環境変数でターゲットデバイスを選択

- OMP_TARGET_OFFLOAD 環境変数で、ターゲット領域を実行するデバイスを指定します
  - デバッグに便利
  - OMP_TARGET_OFFLOAD={“MANDATORY” | “DISABLED” | “DEFAULT”}

タイプ	説明
MANDATORY	ターゲット領域のコードを GPU またはほかのアクセラレーターで実行します
DISABLED	ターゲット領域のコードを CPU で実行します
DEFAULT	デバイスが利用可能な場合はターゲット領域のコードを GPU で実行し、利用できなければ CPU にフォールバックされます

非同期オフロード

- OpenMP* ターゲット構造はデフォルトで同期されます
  - ホストスレッドはターゲット領域の完了を待機してから処理を続行します
- **nowait** 節はターゲット構造を非同期にします
  - ターゲット領域は OpenMP* タスクとなります (タスク同期を使用)

```
#pragma omp task                                                depend(out:in1)
          init_data(in1);

#pragma omp target map(to:in1) map(from:out1) nowait            depend(in:in1) depend(out:out1)
          compute_1(in1, out1, N);

#pragma omp target map(to:in2) map(from:out2) nowait            depend(out:out2)
          compute_2(in2, out2, N);

#pragma omp target map(to:out1) map(to:out2) nowait             depend(in:out1) depend(in:out2)
          compute_3(out1, out2, N);

#pragma omp taskwait
```

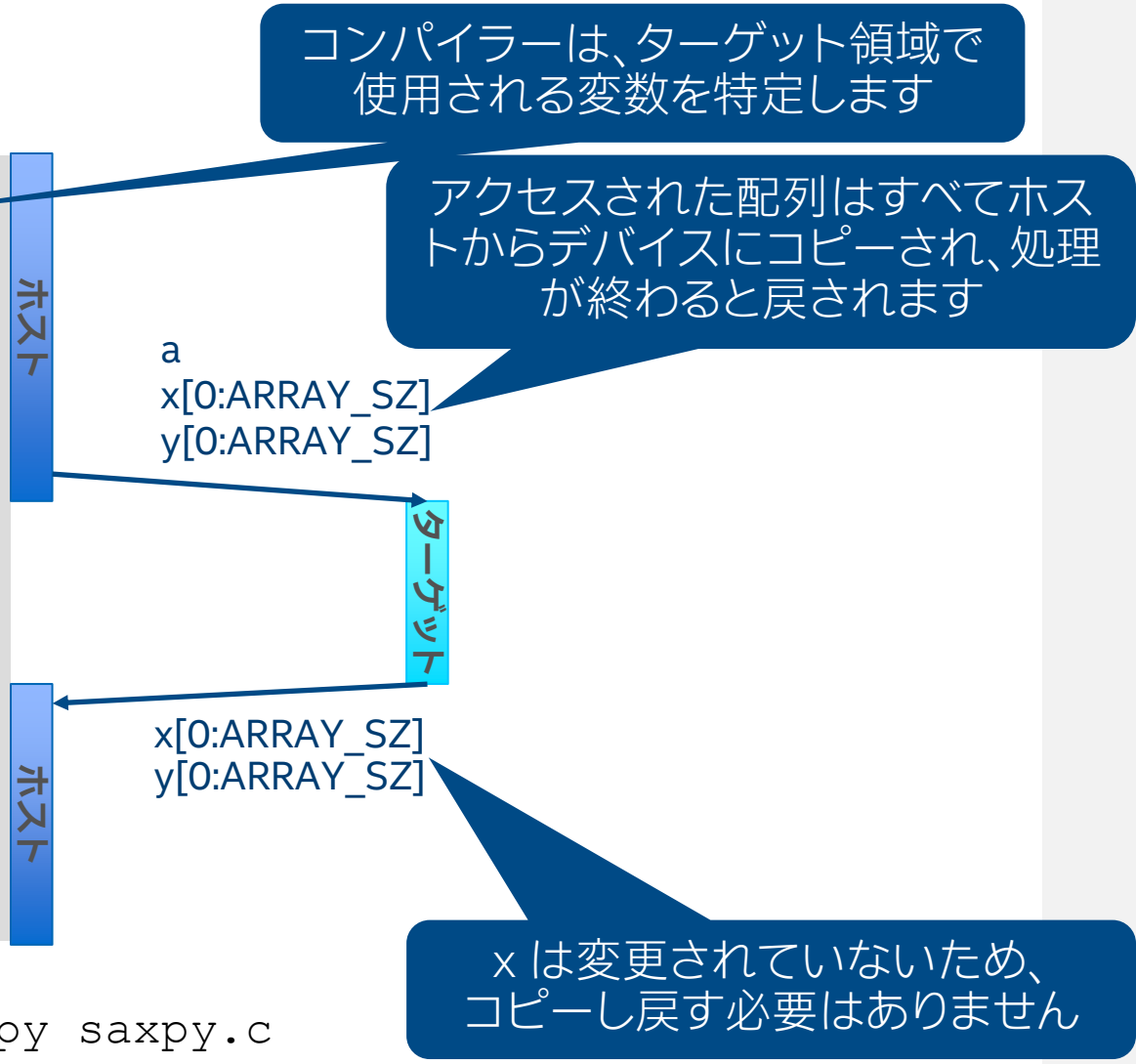
デバイスデータの管理

オフロードデータ

- ホストとデバイスは異なるメモリー空間を持ちます
  - ターゲット領域内でデータにアクセスする前に、データをターゲットデバイスにマップする必要があります
  - ターゲット領域内でアクセスされる変数のデフォルトは以下です
    - スカラー - **`firstprivate`** として扱われます
    - 静的配列 - ターゲット領域の入り口と出口でホストとデバイス間でコピーされます
  - データ環境は字句でスコープされます
    - データ環境は、波括弧を閉じた時点で破棄されます
    - 割り当てられたバッファー/データは、自動的に解放されます

例: saxpy

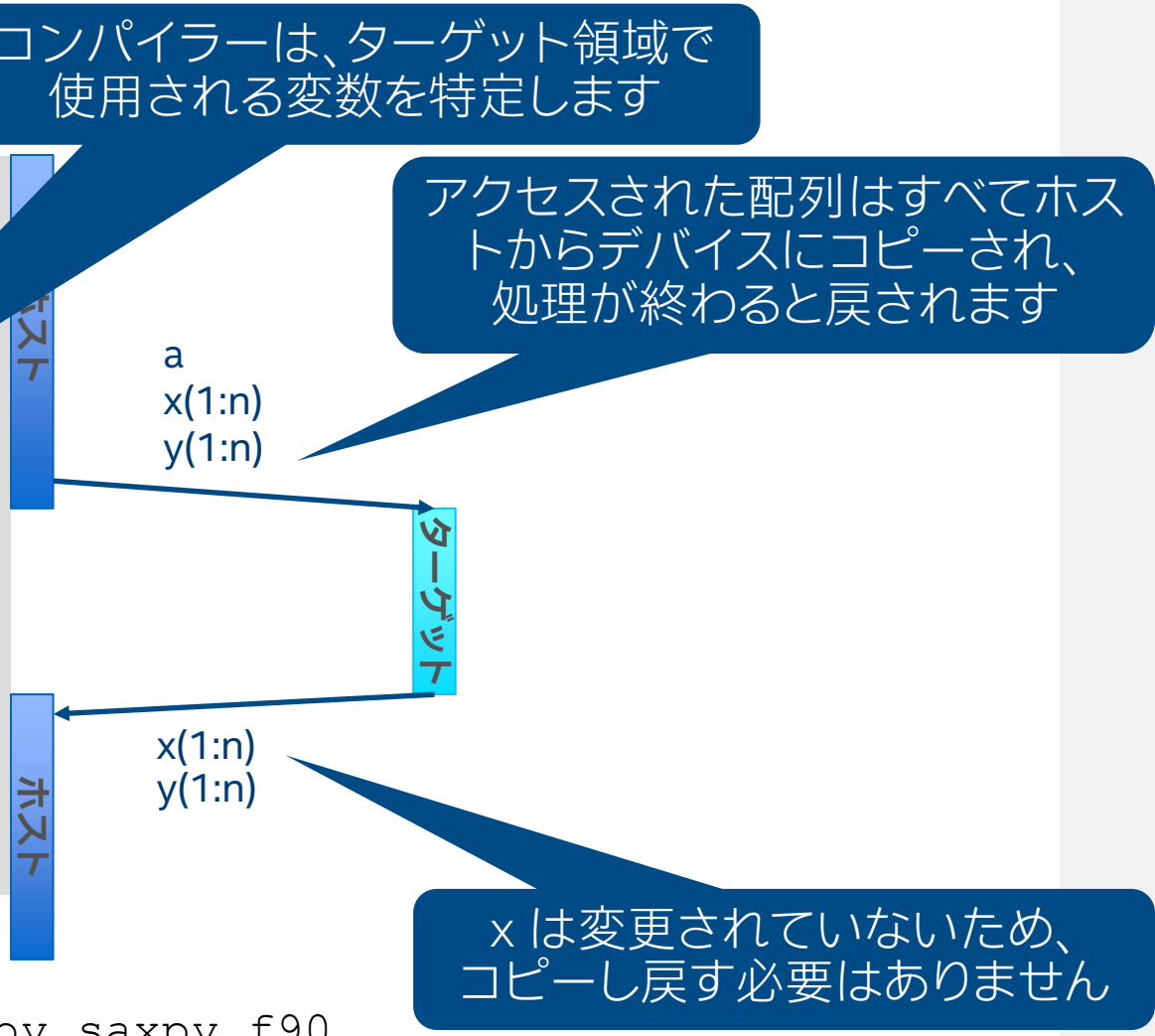
```
void saxpy() {
    float a, x[ARRAY_SZ], y[ARRAY_SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target
    for (int i = 0; i < ARRAY_SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf¥n", t);
}
```




```
icx -fiopenmp -fopenmp-targets=spir64 -o saxpy saxpy.c
```

例: saxpy

```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    integer :: n, i
    real(kind=real32) :: a
    real(kind=real32), dimension(n) :: x
    real(kind=real32), dimension(n) :: y
!$omp target
    do i=1,n
       y(i) = a * x(i) + y(i)
    end do
!$omp end target
end subroutine
```



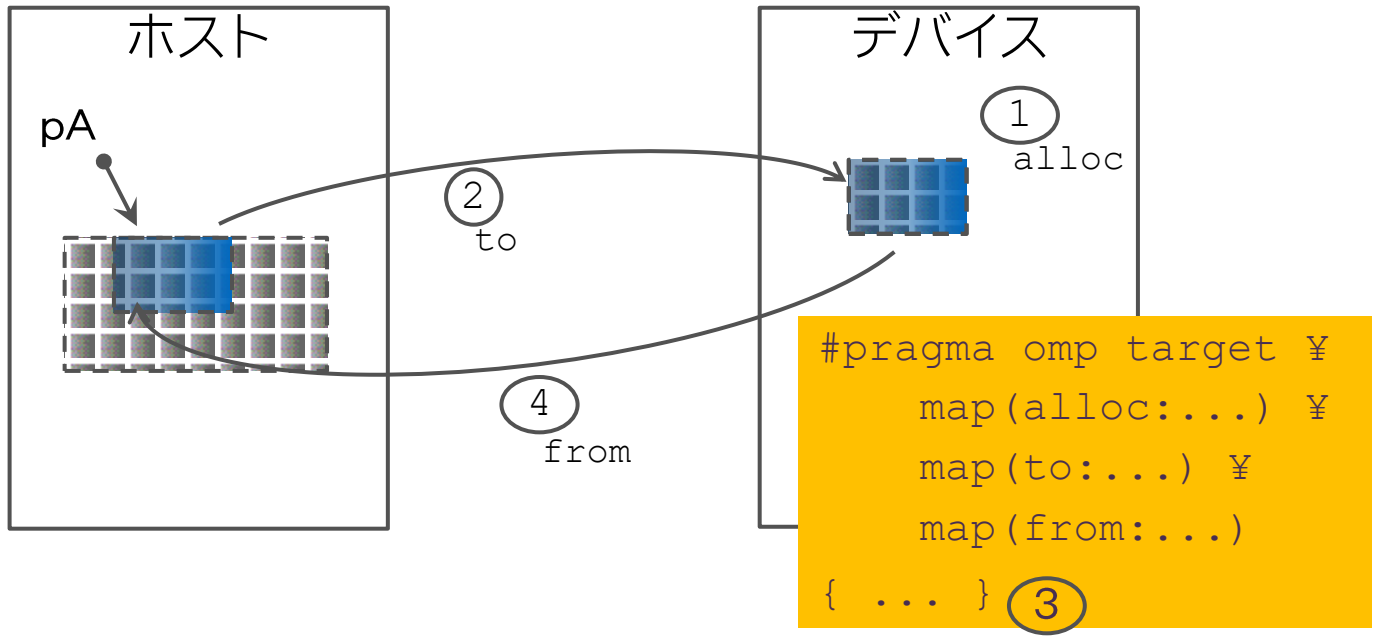

```
ifx -fiopenmp -fopenmp-targets=spir64 -o saxpy saxpy.f90
```

map 節

- データ環境の元の変数をデバイスのデータ環境の変数にどのようにマップするか、map 節により手動で決定します
  - **omp target map (**マップタイプ**:** リスト**)**
  - 利用可能なマップタイプ
    - alloc: 変数はターゲットデバイスのストレージに割り当てられます (値はコピーされません)
    - to: 変数はターゲットデバイスのストレージに割り当てられ、値はコピーされます
    - from: ターゲット領域の最後でデバイスのストレージに割り当てられた変数を元の変数に戻します
    - tofrom: デフォルトであり、to と from の両方が適用されます

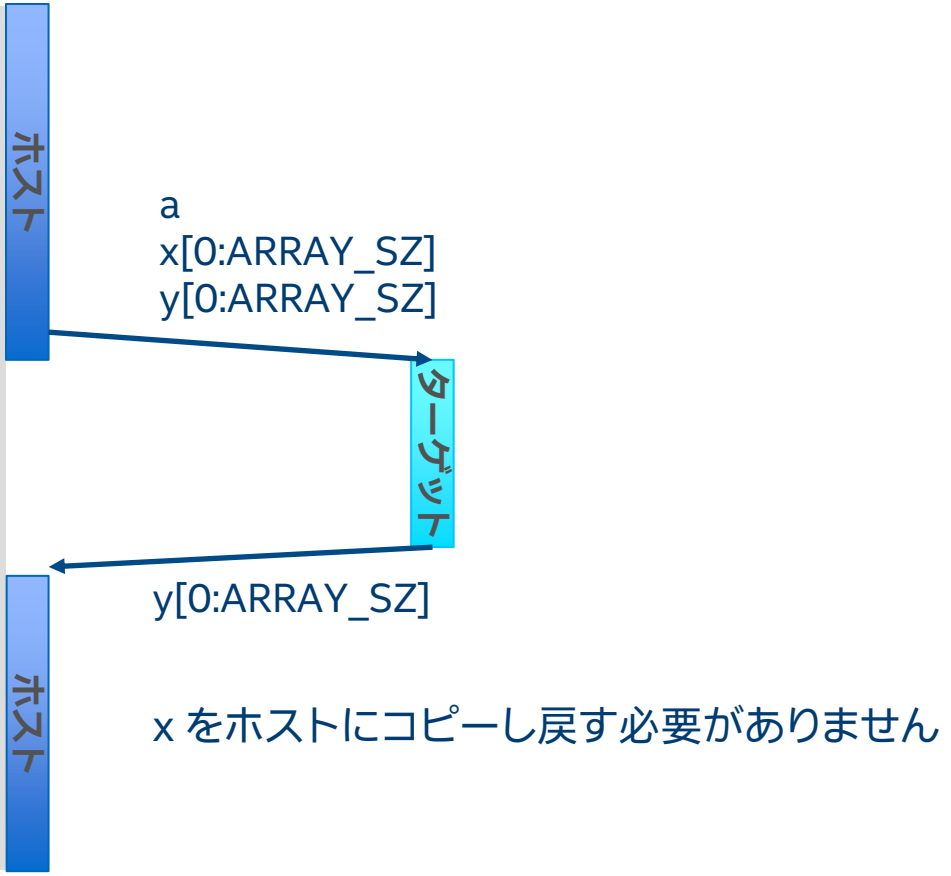
map 節

- データ環境の元の変数をデバイスのデータ環境の変数にどのようにマップするか、**map** 節により手動で決定します



例: saxpy

```
void saxpy() {
    double a, x[ARRAY_SZ], y[ARRAY_SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x) ¥
                   map(tofrom:y)
    for (int i = 0; i < ARRAY_SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf¥n", t);
}
```




```
icx -fiopenmp -fopenmp-targets=spir64 -o saxpy saxpy.c
```

動的に割り当てられたデータのマップ

- ポインターが動的に割り当てられている場合、マップされる要素数を明示的に指定する必要があります

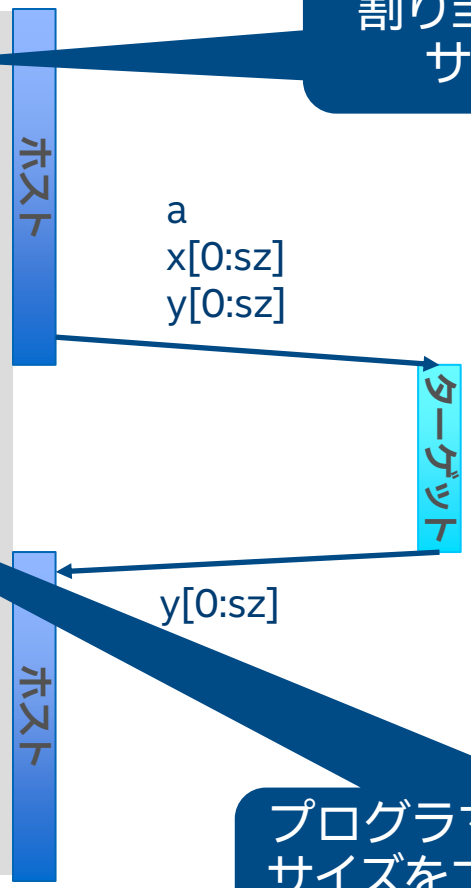
```
#pragma omp target map(to:array[start:length])
```

```
!$omp target map(to:array(start:end))
```

  - 部分配列を指定できます
  - 注: C/C++ (レングスを使用) 構文は Fortran (終了を使用) とは異なります

例: saxpy

```
void saxpy(float a, float* x, float* y,
           int sz) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x[0:sz]) ¥
                   map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
       y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf¥n", t);
}
```



コンパイラーは、ポインターに割り当てられているメモリーのサイズを特定できません

プログラマーは、転送するデータサイズをコンパイラーに知らせる必要があります

```
icx -fiopenmp -fopenmp-targets=spir64 -o saxpy saxpy.c
```

ターゲット領域間とのデータ転送を最小化

- **target data**、**target enter data**、および **target exit data** を使用してターゲット領域を形成し、ホストとデバイス間のデータ共有を最適化します
  - 変数をマップしてコードの実行をオフロードしません
  - 変数は、ターゲットデータ領域が存続する間デバイスに保持されます
  - **target update** 構造はホストとデバイス間の値をコピーできます

target data 構造の構文

- データスコープを作成し、ホストからデバイスにデータを転送して戻します
- 構文 (C/C++)

```
#pragma omp target data [節[[,] 節],…]
    構造化ブロック
```

- 構文 (Fortran)

```
!$omp target data [節[[,] 節],…]
    構造化ブロック
!$omp end target data
```

- 節

```
device(スカラー整数式)
map([{alloc | to | from | tofrom | release | delete}:] リスト)
if(スカラー式)
```

target data の例

- **target data** 構造を使用して、ターゲットデータ環境を作成します

```
#pragma omp target data map(tofrom: x)
{
        #pragma omp target map(to: y)
        {
                ...// 最初のターゲット領域、デバイスは x と y を操作
        }
        host_update(y);
        #pragma omp target map(to: y)
        {
                ...// 2 番目の target 領域、デバイスは x と y を操作
        }
}
```

デバイスデータ環境が作成され、配列 x がマップされます

y はホスト側で更新されるため、ターゲット領域ごとにマップする必要があります

target update 構造の構文

- 既存のデータデバイス環境との間でデータ転送を発行します
- 構文 (C/C++)

```
#pragma omp target update [節[[,] 節],…]
```

構文 (Fortran)

```
!$omp target update [節[[,] 節],…]
```

節

- device(スカラー整数式)
- to(リスト)
- from(リスト)
- if(整数式)

target enter/exit data および update の例

- **target enter/exit data** を使用してターゲットデータ領域間とマップを行います
- ホストとデバイス間で一貫性を保つため **target update** を使用します

```
#pragma omp target enter data map(to: y) map(alloc: x)
#pragma omp target
{...// 最初のターゲット領域、デバイスは x と y を操作
}
#pragma omp target update from(y)
host_update(y);
#pragma omp target update to(y)

#pragma omp target
{...// 2 番目の target 領域、デバイスは x と y を操作
}
#pragma omp target exit data map(from:x)
```

データ環境は複数の関数にまたがることができ非構造化マッピングと呼ばれます

y はホスト側で更新されるため、デバイス間と更新する必要があります

グローバル変数をデバイスにマップ

- プログラムの実行期間中、変数をデバイスにマップするには **declare target** 構造を使用します

```
#pragma omp declare target
int a[N]
#pragma omp end declare target
…
init(a);
#pragma omp target update to(a)
…
#pragma omp target teams ¥
distribute parallel for
for (int i=0; i<N; i++){
     result[i] = process(a[i]);
}
```

```
module my_arrays
!$omp declare target (a)
integer :: a(N)
end module
…
use my_arrays
integer :: i
call init(a);
!$omp target update to(a)
…
!$omp target teams distribute &
parallel do
do i=1,N
          result(i) = process(a(i));
end do
```

統合共有メモリー

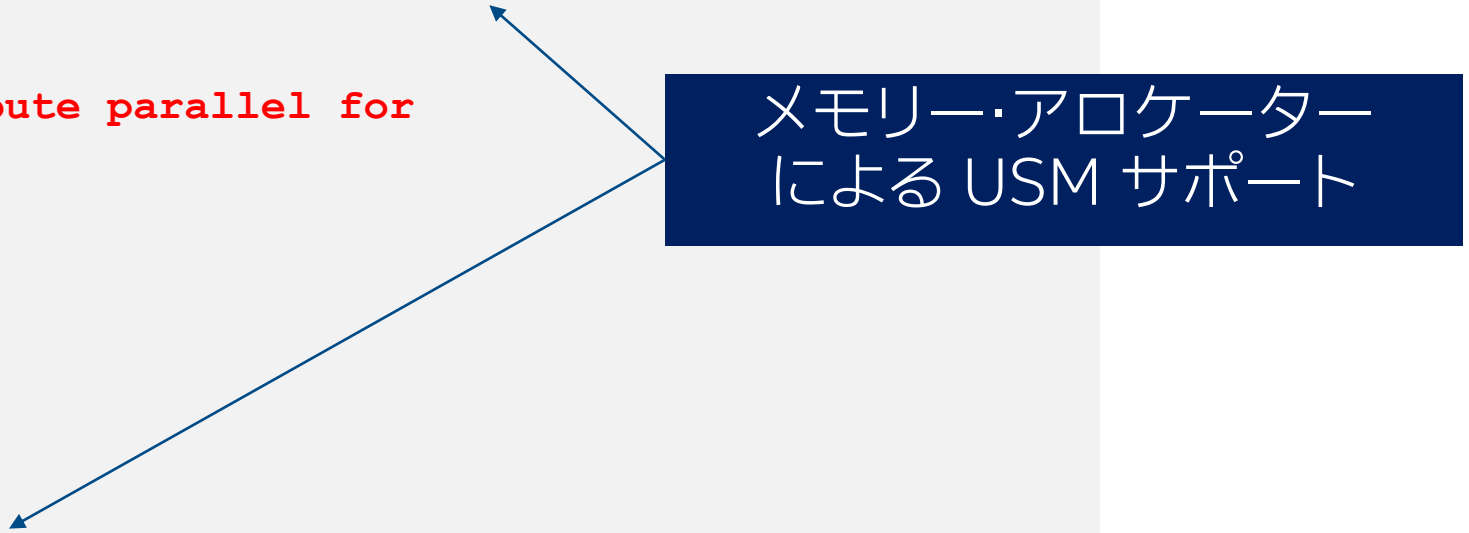
- CPU と GPU で 1 つのアドレス空間を持ちます
- アプリケーションに対し透過的な CPU と GPU 間のデータ移行
  - 明示的なデータのマップは必要ありません

タイプ	場所	アクセス可能な場所	割り当てルーチン
ホスト	ホスト	ホストまたはデバイス	`omp_target_alloc_host(size, device_num)`
デバイス	デバイス	デバイス	`omp_target_alloc_device(size, device_num)`
共有	ホストまたはデバイス	ホストまたはデバイス	`omp_target_alloc_shared(size, device_num)`

- 暗黙的なデータ移動に共有またはホストメモリーを使用して、コーディングを簡単にします
- 最大のパフォーマンスを達成するため、明示的なデータ移動にデバイスメモリーを使用します

統合共有メモリー (暗黙的) の例

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define SIZE 1024
#pragma omp requires unified_shared_memory
int main() {
  int deviceId = (omp_get_num_devices() > 0) ?
       omp_get_default_device() : omp_get_initial_device();
 int *a = (int *)omp_target_alloc_shared(SIZE * sizeof(int) , deviceId);
 int *b = (int *)omp_target_alloc_shared(SIZE * sizeof(int) , deviceId);
  for (int i = 0; i < SIZE; i++) {
    a[i] = i;     b[i] = SIZE - i;
  }
#pragma omp target teams distribute parallel for
  for (int i = 0; i < SIZE; i++) {
    a[i] += b[i];
  }

  for (int i = 0; i < SIZE; i++) {
    if (a[i] != SIZE) {
      printf("%s failed¥n", __func__);
      return EXIT_FAILURE;
    }
  }
 omp_target_free(a, deviceId);
 omp_target_free(b, deviceId);
  printf("%s passed¥n", __func__);
  return EXIT_SUCCESS;
}
```



統合共有メモリー (明示的) の例

```
…
int main() {
  int deviceId = (omp_get_num_devices() > 0) ? omp_get_default_device() : omp_get_initial_device();
  int *a = (int *)malloc(SIZE * sizeof(int));  int *b = (int *)malloc(SIZE * sizeof(int));
  for (int i = 0; i < SIZE; i++) {
    a[i] = i;     b[i] = SIZE - i;
  }
 int *a_dev = (int *)omp_target_alloc_device(SIZE * sizeof(int) , deviceId);
 int *b_dev = (int *)omp_target_alloc_device(SIZE * sizeof(int) , deviceId);
 int error=omp_target_memcpy(a_dev, a, SIZE*sizeof(int), 0, 0, deviceId, 0);
 error=omp_target_memcpy(b_dev, b, SIZE*sizeof(int), 0, 0, deviceId, 0);
 #pragma omp target teams distribute parallel for
  for (int i = 0; i < SIZE; i++) {
    a_dev[i] += b_dev[i];
  }
 error=omp_target_memcpy(a, a_dev, SIZE*sizeof(int), 0, 0, 0, deviceId);
 error=omp_target_memcpy(b, b_dev, SIZE*sizeof(int), 0, 0, 0, deviceId);

  for (int i = 0; i < SIZE; i++) {
    if (a[i] != SIZE) { printf("%s failed¥n", __func__);  return EXIT_FAILURE; }}
 omp_target_free(a_dev, deviceId);
 omp_target_free(b_dev, deviceId);
  free(a);  free(b);
  printf("%s passed¥n", __func__);
  return EXIT_SUCCESS;
}
```

ホストからデバイスへの
明示的なデータ移動

デバイスからホストへの
明示的なデータ移動

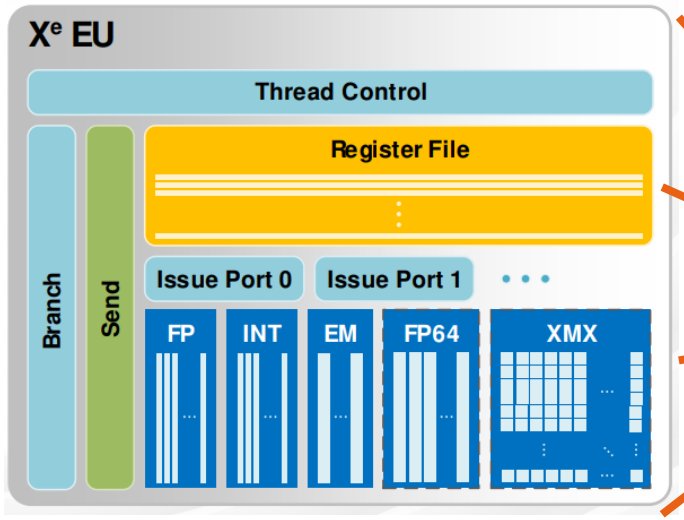
並列処理

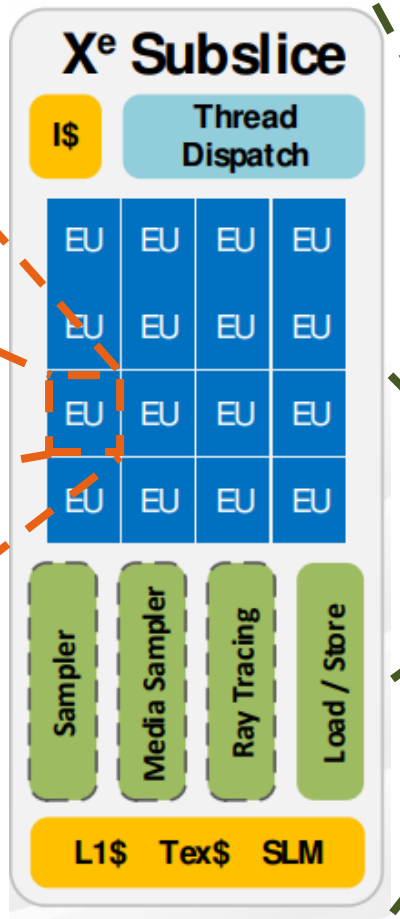
ターゲットデバイスで並列処理を行う

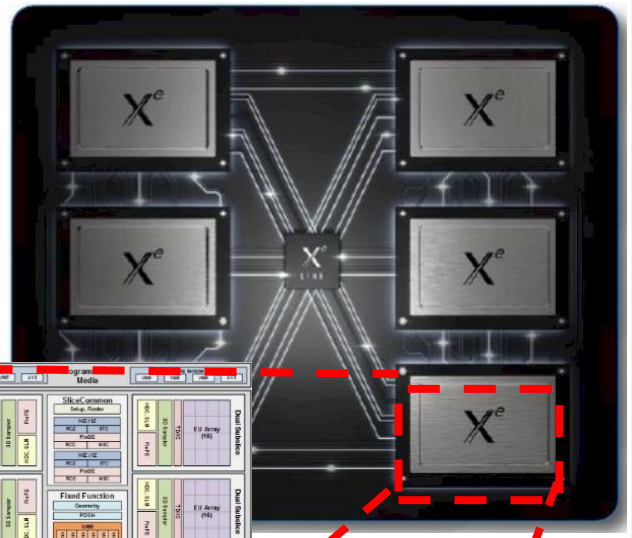
- **`target`** 構造は、制御フローをターゲットデバイスへ転送します
  - 制御の転送はシーケンシャルで同期されます

- OpenMP* はオフロードと並列処理を分離します
  - ターゲットデバイス上で並列処理を明示的に生成する必要があります
  - 理論的には、すべての OpenMP* 構造と組み合わせることができます
  - 実際には、ターゲットデバイス向けの有用な OpenMP* サブセットしかありません

GPU アーキテクチャー







OpenMP* GPU オフロードと OpenMP* 構造

- OpenMP* GPU オフロードは、“通常の” OpenMP* 構造をすべてサポートします
  - 例: **`parallel`**、**`for/do`**、**`barrier`**、**`sections`**、**`tasks`** など
  - すべての構造が有益なわけではありません

- 単一 GPU サブスライス以外のスレッド化モデルには対応していません
  - サブスライス間には同期はありません
  - サブスライスの L1 キャッシュ間のコヒーレンスとメモリーフェンスはありません

例: saxpy

- デバイスでは、**parallel** 構造で **1 つの**サブスライスまたはストリーム・マルチプロセッサーで実行されるスレッドのチームが生成されます

```
void saxpy(float a, float* x, float* y,
           int sz) {
#pragma omp target map(to:x[0:sz]) ¥
                   map(tofrom(y[0:sz])
#pragma omp parallel for simd
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

ホスト

ターゲット

GPU は複数レベルデバイスです (SIMD、スレッド、スレッドブロック)

ループを並列に実行するスレッドのチームを生成して SIMD 化します。GPU サブスライスが 1 つしか利用されず、GPU の利用効率は著しく低くなります

```
icx -fiopenmp -fopenmp-targets=spir64 -o saxpy saxpy.c
```

teams 構造

- 複数レベルの並列デバイスをサポートします
- 構文 (C/C++):

```
#pragma omp teams [節[[,] 節],…]
    構造化ブロック
```

- 構文 (Fortran):

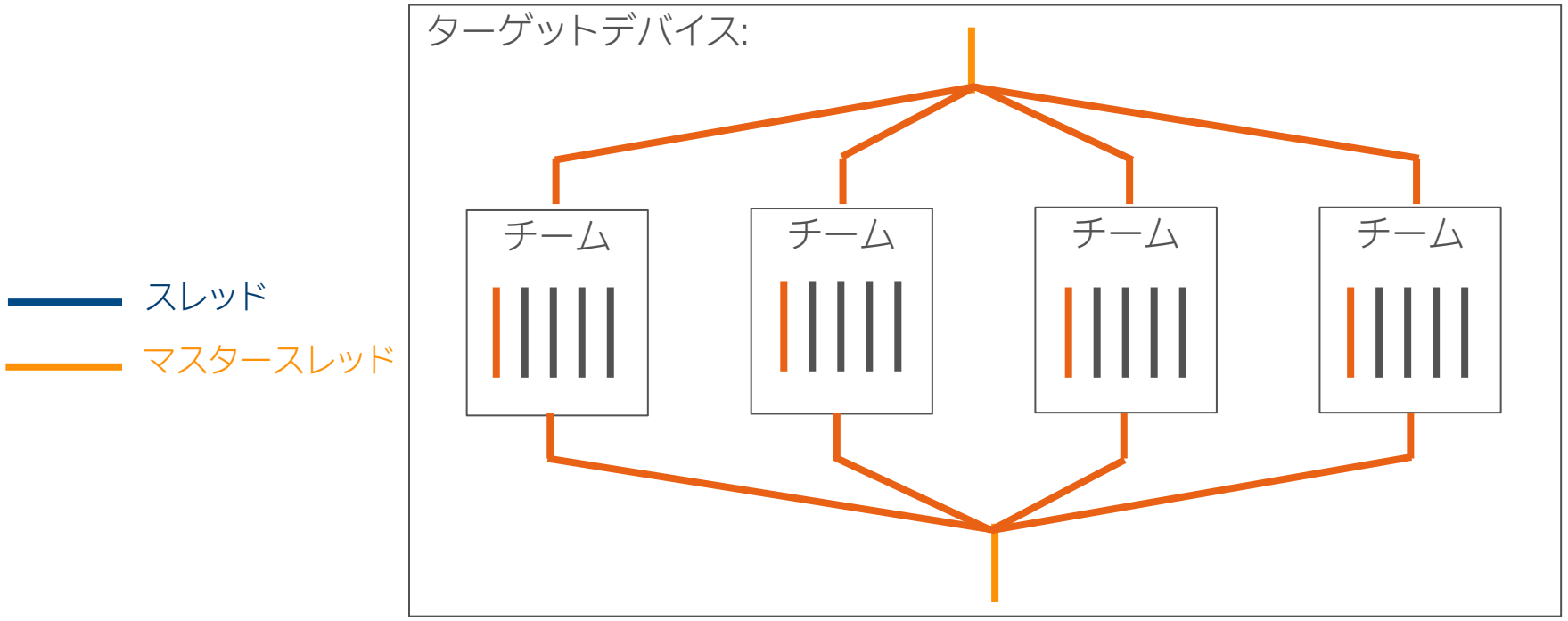
```
!$omp teams [節[[,] 節],…]
    構造化ブロック
```

- 節

```
num_teams(整数式)、thread_limit(整数式)
default(shared | firstprivate | private none)
private(リスト)、firstprivate(リスト)、shared(リスト)、reduction(演算子:リスト)
```

teams 構造

- 複数のマススレッドを生成し、スレッドのチーム (リーグ) のセットを効率良く作成します
- チーム間で同期は適用されません



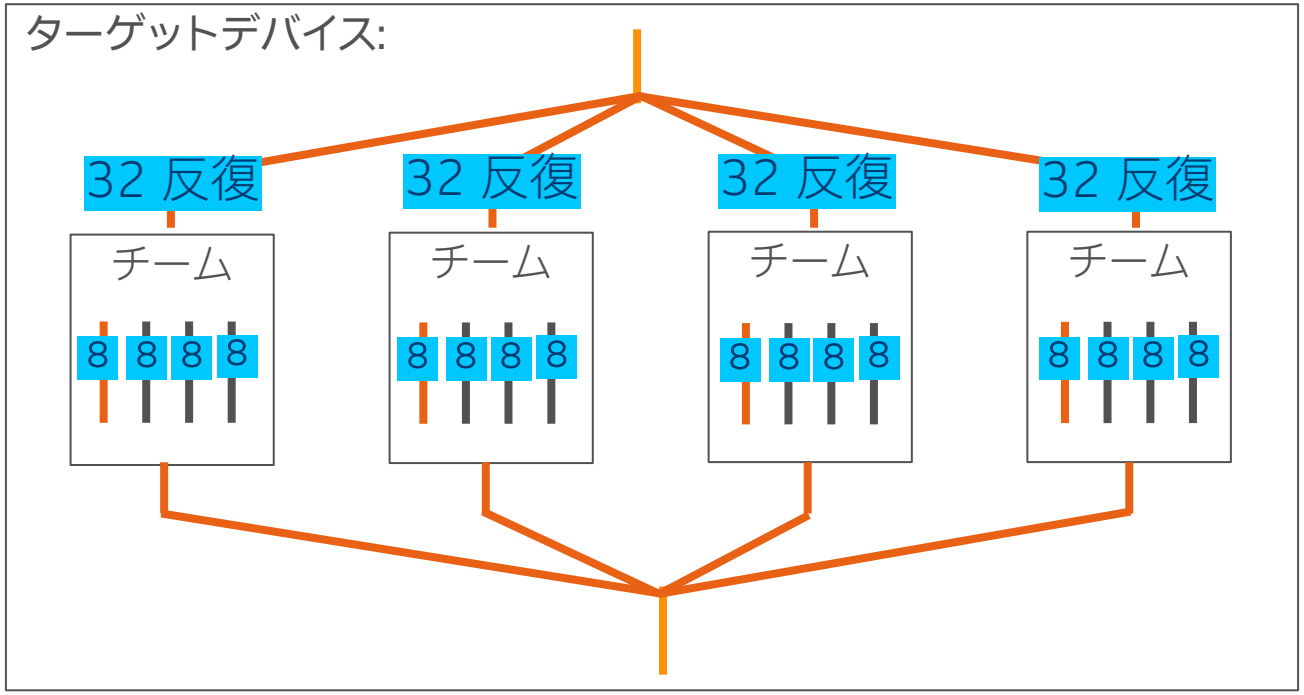
distribute 構造

- **distribute** 構造は、ループの反復を異なるチームに分散します
  - リーグ内でワークシェアを行います
  - **teams** 領域内で入れ子になります
  - 分散のスケジュールを指定できます
  - **parallel** 領域の **for/do** 構造に似ています
  - 構文
    - **#pragma omp distribute** [節[[,] 節]…]
    - **!$omp distribute** [節[[,] 節]…]

分散の様子

スレッド

マスタースレッド



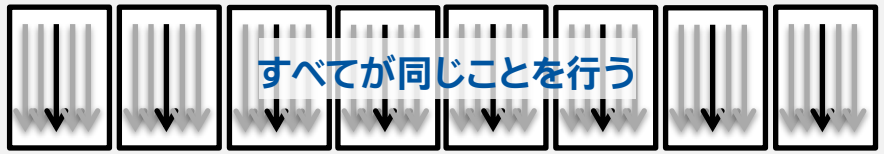

```
omp target

omp teams
omp distribute

omp parallel
omp for/do

omp simd
```

複数レベルの並列処理: saxpy

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target map(to:x[0:sz]) map(tofrom(y[0:sz])
    {

        {




            for (ib = 0; ib < sz; ib += num_blocks) {




                for (int i = ib; i < ib + num_blocks; i++) {



                    y[i] = a * x[i] + y[i];
}   }   }   }   }
```

複数レベルの並列処理: saxpy

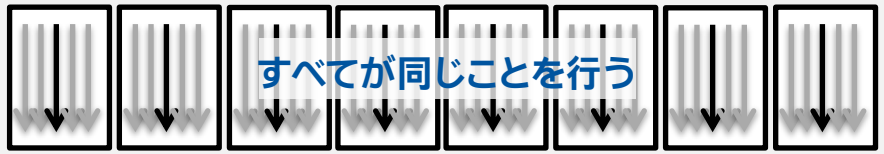
```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target map(to:x[0:sz]) map(tofrom(y[0:sz])
    {
        #pragma omp teams num_teams(num_blocks)
        {
```

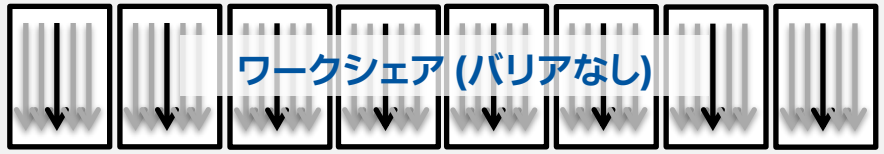



```
            for (ib = 0; ib < sz; ib += num_blocks) {


                for (int i = ib; i < ib + num_blocks; i++) {


                    y[i] = a * x[i] + y[i];
}   }   }   }   }
```

複数レベルの並列処理: saxpy

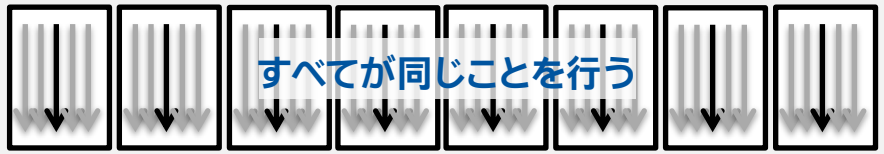
```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target map(to:x[0:sz]) map(tofrom(y[0:sz])
    {
        #pragma omp teams num_teams(num_blocks)
        {
            #pragma omp distribute
            for (ib = 0; ib < sz; ib += num_blocks) {
                for (int i = ib; i < ib + num_blocks; i++) {
                    y[i] = a * x[i] + y[i];
}   }   }   }   }
```

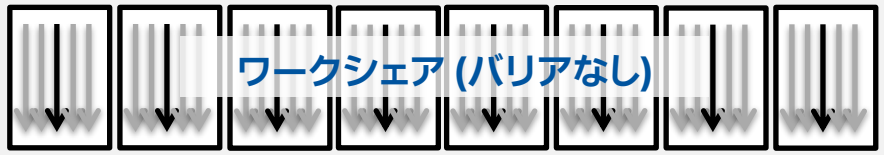


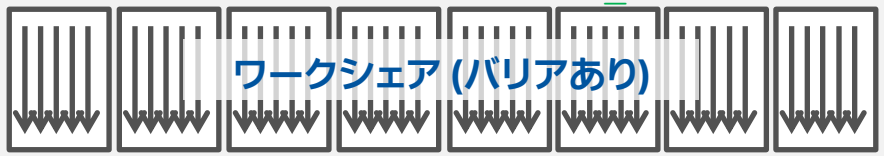


複数レベルの並列処理: saxpy

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target map(to:x[0:sz]) map(tofrom(y[0:sz])
    {
        #pragma omp teams num_teams(num_blocks)
        {
            #pragma omp distribute
            for (ib = 0; ib < sz; ib += num_blocks) {
                #pragma omp parallel for simd
                for (int i = ib; i < ib + num_blocks; i++) {
                    y[i] = a * x[i] + y[i];
}   }   }   }   }
```







複数レベルの並列処理: saxpy

- 利便性のため、OpenMP* では複合構造を定義します

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams distribute parallel for simd ¥
            num_teams(num_blocks) map(to:x[0:sz]) map(tofrom(y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

```
subroutine saxpy(a, x, y, n)
    ! 宣言は省略
!$omp omp target teams distribute parallel do simd &
            num_teams(num_blocks) map(to:x) map(tofrom(y)
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target teams distribute parallel do simd
end subroutine
```

まとめ

まとめ

- OpenMP* オフロードは、インテル® oneAPI HPC ツールキットに含まれるインテル® C++ コンパイラーおよびインテル® Fortran コンパイラーによってサポートされます
- オフロードには **`target`** ディレクティブを使用します
- **`target`**、**`target data`**、**`target enter/exit data`** ディレクティブと **`map`** 節を使用してデータ転送の効率を改善します
- **`teams/distribute`** ディレクティブを使用して複数の GPU サブスライスを完全に活用します
- **`parallel/for/do`** ディレクティブを使用して、GPU サブスライス内でスレッドを利用します
- **`simd`** ディレクティブを使用して、GPU 実行ユニットで simd 命令を実行します

その他のトピック

- インテル® Advisor を使用: オフロードのモデル化でオフロードに有益なコード領域を特定します
  - アクセラレーターでの性能の高速化を予測します
- インテル® Advisor を使用: CPU と GPU ハードウェアのパフォーマンスの上限を視覚化するルーフライン解析を利用できます
  - ボトルネックと最適化の手順に関する情報を提供します

インテル® DevCloud のセッション紹介

法務上の注意書きと最適化に関する注意事項

本資料には、開発中の製品、サービスおよびプロセスについての情報が含まれています。本資料に含まれる情報は予告なく変更されることがあります。 インテル® テクノロジーの機能と利点はシステム構成によって異なり、対応するハードウェアやソフトウェア、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、各システムメーカー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。

上記のベンチマーク結果は、追加のテストによって変更が必要になる可能性があります。結果は、テストに使用される特定のプラットフォーム構成や作業負荷に依存します。個々のユーザーのコンポーネント、コンピューター・システム、作業負荷では同様の結果が得られない可能性があります。結果は、必ずしもほかのベンチマークを代表するものではなく、ほかのベンチマークでは、結果が異なることがあります。

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。詳細については、www.intel.com/benchmarks (英語) を参照してください。

本資料に掲載されている情報は現状のまま提供され、いかなる保証もいたしません。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を許諾するためのものではありません。製品に付属の売買契約書『Intel's Terms and Conditions of Sale』に規定されている場合を除き、インテルはいかなる責任を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証(特定目的への適合性、商品性に関する保証、第三者の特許権、著作権、その他、知的財産権の侵害への保証を含む)をするものではありません。

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。

改訂 #20110804


intel