池井　満

検討の動機　oneAPI

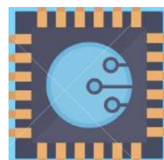
- C++ および SYCL* 標準ベースのクロスアーキテクチャー言語
- 主要なドメイン固有の関数を高速化するように設計された強力な API
- ベンダーにハードウェア抽象化レイヤーを提供する低水準ハードウェア・インターフェイス
- コミュニティーと業界のサポートを促進するオープンなスタンダード
- 複数のアーキテクチャーでコードを再利用可能

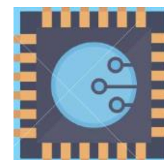
アプリケーション・ワークロード

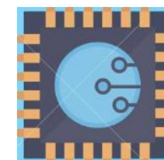
oneAPI 業界仕様

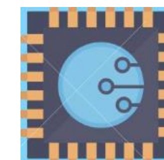
ダイレクト・プログラミング	API プログラム
C++ および SYCL* 標準	ライブラリー • 数値演算 (BLAS) • マルチスレッド • ビデオ • 暗号

低水準ハードウェア・インターフェイス


CPU


GPU


FPGA


その他の加速器

対象プログラム: OpenFDTD

- OpenFDTD は株式会社 EEM (2022年9月30日に解散) により開発 & メンテナンスされていたオープンソースの FDTD 法[1] を用いたシミュレーション・ソフト
- 2023年5月15日現在は、http://emoss.starfree.jp/ で公開されており、動作環境としては Windows* と Linux* が推奨され、両者で実行するための C 言語のソースコードが提供されている
- 実行速度の高速化のために、OpenMP*[2] や MPI[3] を用いた並列化のほかに CUDA*[4] を用いた CPU + GPU でのハイブリッド・プログラム版も提供されている
- そこで、Intel が oneAPI と呼ぶツール[5] で提供する SYCL* をベースとしたコンパイラーである DPC++[5] を用いたハイブリッド・プログラミングを試みた

SYCL*[6] とは

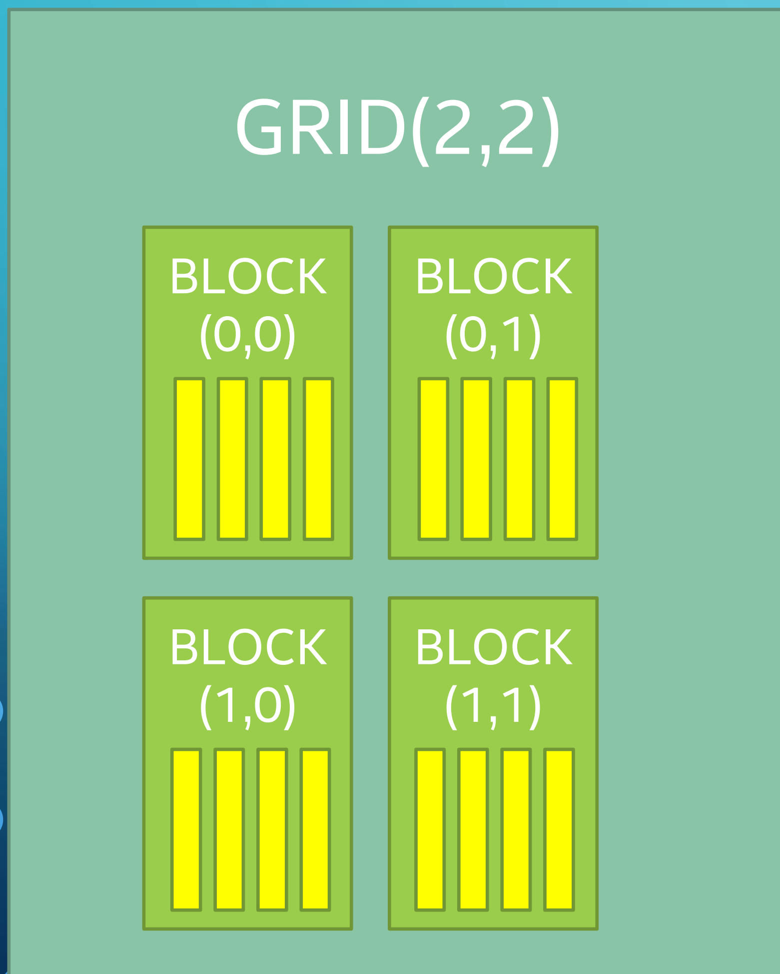
- CPU に加えて GPU のような演算加速器に演算をオフロードする、ヘテロジニアスな並列処理行うために提案されている抽象化レイヤー
- Khronos で標準化されているベース言語として C++ を用いた規格であり、これを実装する複数の処理系が存在している
- インテル社は新しい C++ コンパイラーでこれをサポートしており (-fsycl)、これを用いて C++ で CPU のほか GPU や FPGA といったさまざまな加速器用にプログラムをコンパイルすることができる
- インテル® oneAPI ツールキット[7] のコンパイラーとして、Linux* または Windows* 用のプログラム開発環境とともに提供されている
- コンパイラーのバックエンドとして LLVM/Clang [8] を採用しており、NVIDIA や AMD の GPU 向けのほか、将来の加速器への対応も考慮されている

SYCL* による GPU プログラミングの方針

- 加速器に渡す演算部分 (kernel) を C++ のラムダ式として渡す
- 対象としている OpenFDTD では CUDA* を用いて kernel を GPU 用に書き直したものが提供されているので、これを参考に C++ のラムダ式で記述することにした
- 加速器用のデータ並列 kernel の記述方法 (クラス) としては、GPU 向けに提供されている ND-Range を用いる
- メモリーモデルとしては、CPU と GPU 間で明示的にデータの転送を行わない USM (統合共有メモリー) を用いる
- このため、CPU と GPU 間の物理的なデータの移動は C++ のランタイムに任されることになる

GPU のプログラミング (例) ― Hello World

- GPU の上の 4 つの実行ユニットを 2X2 の 2 次元上に並べてそれぞれのユニットで 4 スレッドで「Hello World!」を表示するプログラムを実行する



Intel(r) oneAPI Tools

```
C:\Users\mikei\OneDrive\デスクトップ\GPU>icpx -fsycl hello2.cpp

C:\Users\mikei\OneDrive\デスクトップ\GPU>a
(0,1) Th 0 : Hello World!
(0,0) Th 0 : Hello World!
(0,0) Th 1 : Hello World!
(0,0) Th 2 : Hello World!
(0,0) Th 3 : Hello World!
(1,0) Th 0 : Hello World!
(1,0) Th 1 : Hello World!
(1,0) Th 2 : Hello World!
(1,0) Th 3 : Hello World!
(0,1) Th 1 : Hello World!
(0,1) Th 2 : Hello World!
(0,1) Th 3 : Hello World!
(1,1) Th 0 : Hello World!
(1,1) Th 1 : Hello World!
(1,1) Th 2 : Hello World!
(1,1) Th 3 : Hello World!

C:\Users\mikei\OneDrive\デスクトップ\GPU>
```

CUDA* と SYCL* のプログラミング比較 (例)

```
/* hello.cu */

#include <stdio.h>


__global__ void hello(){
  int t = threadIdx.x;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  printf("(%d,%d) Th %d : Hello World!\n",bx,by,t);
}

int main() {
  dim3 grid(2,2);
  dim3 block(4);

  hello<<< grid, block >>>();
  cudaDeviceSynchronize();
  return 0;
}
```

```
/* hello.cpp */

#include <CL/sycl.hpp>
using namespace sycl;

void hello(nd_item<2> i)
{
  int t = i.get_local_id(1);
  int bx = i.get_group(1);
  int by = i.get_group(0);
  sycl::ext::oneapi::experimental::
    printf("(%d,%d) Th %d : Hello World!\n",bx,by,t);
}

int main() {
  queue myQ;
  range<2> whole = {2,8};
  range<2> block = {1,4};

  myQ.submit([&](handler& h){
    h.parallel_for(nd_range<2>(whole, block),
      [=](nd_item<2> i){
      hello(i);
    });
  });
}
myQ.wait();
```

CUDA* と SYCL* のベース言語

(1) ベース言語が異なる

- CUDA* は C (++) をベースに独自の言語拡張をしている
- SYCL* は C++ 上で Khronos がクラスなどを拡張したものである

```
/* hello.cu */

#include <stdio.h>


__global__ void hello(){
  int t = threadIdx.x;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  printf("(%d,%d) Th %d : Hello World!\n",bx,by,t);
}

int main() {
  dim3 grid(2,2);
  dim3 block(4);

  hello<<< grid, block >>>();
  cudaDeviceSynchronize();
  return 0;
}
```

```
/* hello.cpp */

#include <CL/sycl.hpp>
using namespace sycl;

void hello(nd_item<2> i)
{
  int t = i.get_local_id(1);
  int bx = i.get_group(1);
  int by = i.get_group(0);
  sycl::ext::oneapi::experimental::
    printf("(%d,%d) Th %d : Hello World!\n",bx,by,t);
}

int main() {
  queue myQ;
  range<2> whole = {2,8};
  range<2> block = {1,4};

  myQ.submit([&](handler& h){
    h.parallel_for(nd_range<2>(whole, block),
      [=](nd_item<2> i){
      hello(i);
    });
  });
}
myQ.wait();
```

CUDA* と SYCL* の加速器側で実行する関数

(2) 加速器側で実行するプログラムの指定方法
- CUDA* ではホストから呼び出される加速器側の関数を「__global__」で関数を修飾する
- SYCL* ではホストからラムダ関数で加速器のプログラムを呼び出すので、特別な修飾はない

```
/* hello.cu */

#include <stdio.h>


__global__ void hello(){
  int t = threadIdx.x;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  printf("(%d,%d) Th %d : Hello World!\n",bx,by,t);
}

int main() {
  dim3 grid(2,2);
  dim3 block(4);

  hello<<< grid, block >>>();
  cudaDeviceSynchronize();
  return 0;
}
```

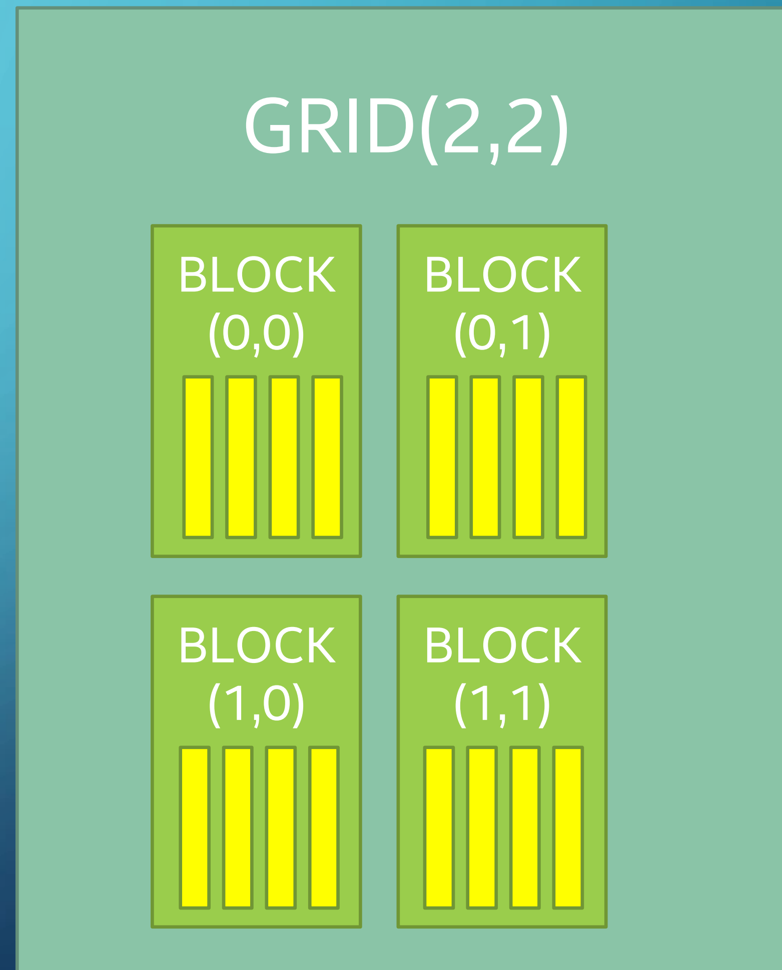
```
/* hello.cpp */

#include <CL/sycl.hpp>
using namespace sycl;

void hello(nd_item<2> i)
{
  int t = i.get_local_id(1);
  int bx = i.get_group(1);
  int by = i.get_group(0);
  sycl::ext::oneapi::experimental::
    printf("(%d,%d) Th %d : Hello World!\n",bx,by,t);
}

int main() {
  queue myQ;
  range<2> whole = {2,8};
  range<2> block = {1,4};

  myQ.submit([&](handler& h){
    h.parallel_for(nd_range<2>(whole, block),
      [=](nd_item<2> i){
      hello(i);
    });
  });
}
myQ.wait();
```

CUDA* での GPU (加速器) へのスレッドの割り当て

(3) 加速器側でのスレッドの割り当て方法

- CUDA* では、「関数名<<< GRID, BLOCK >>>」の形でグローバル関数をホスト側から呼び出す

```
/* hello.cu */

#include <stdio.h>


__global__ void hello(){
  int t = threadIdx.x;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  printf("(%d,%d) Th %d : Hello World!\n",bx,by,t);
}

int main() {
  dim3 grid(2,2);
  dim3 block(4);

  hello<<< grid, block >>>();
  cudaDeviceSynchronize();
  return 0;
}
```

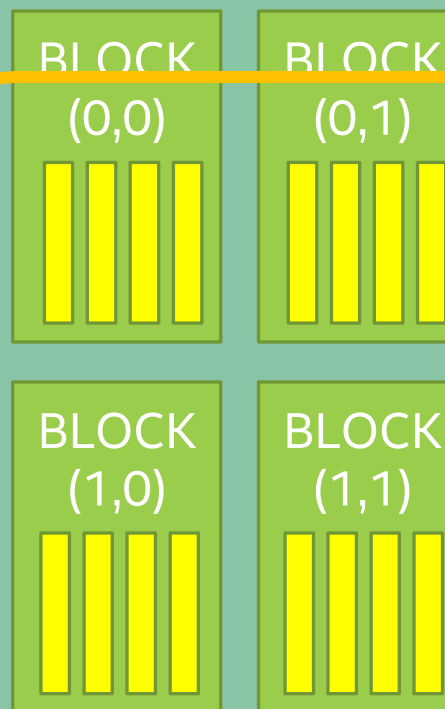


SYCL* (nd_range) での GPU へのスレッド割り当て

(3) 加速器側でのスレッドの割り当て方法

- DPC++ では、「parallel_for(nd_range<>(whole,block), …」の形で指定して、加速器側のプログラムはラムダ式

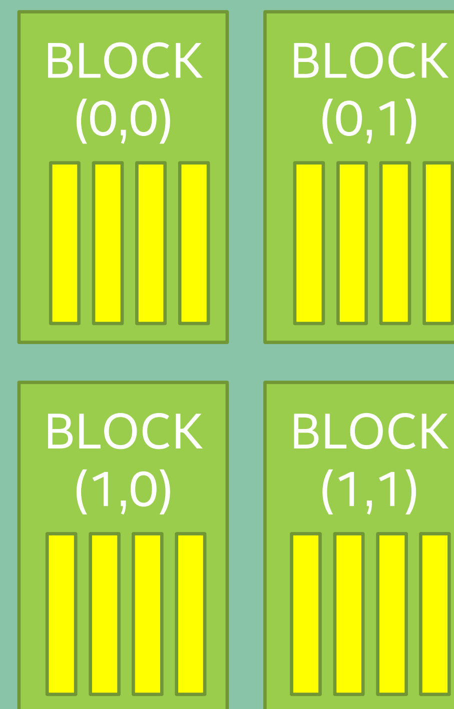
```
/* hello.cpp */

#include <CL/sycl.hpp>
using namespace sycl;

void hello(nd_item<2> i)
{
  int t = i.get_local_id(1);
  int bx = i.get_group(1);
  int by = i.get_group(0);
  sycl::ext::oneapi::experimental::
    printf("(%d,%d) Th %d : Hello World!\n",bx,by,t);
}

int main() {
  queue myQ;
  range<2> whole = {2,8};
  range<2> block = {1,4};

  myQ.submit([&](handler& h){
    h.parallel_for(nd_range<2>(whole, block),
      [=](nd_item<2> i){
      hello(i);
    });
  });
}
myQ.wait();
```



SYCL* (nd_range) での GPU 側プログラムの直接記述

(3) 加速器側でのスレッドの割り当て方法
- SYCL* では、GPU 側プログラムを別関数として記述する必要はない

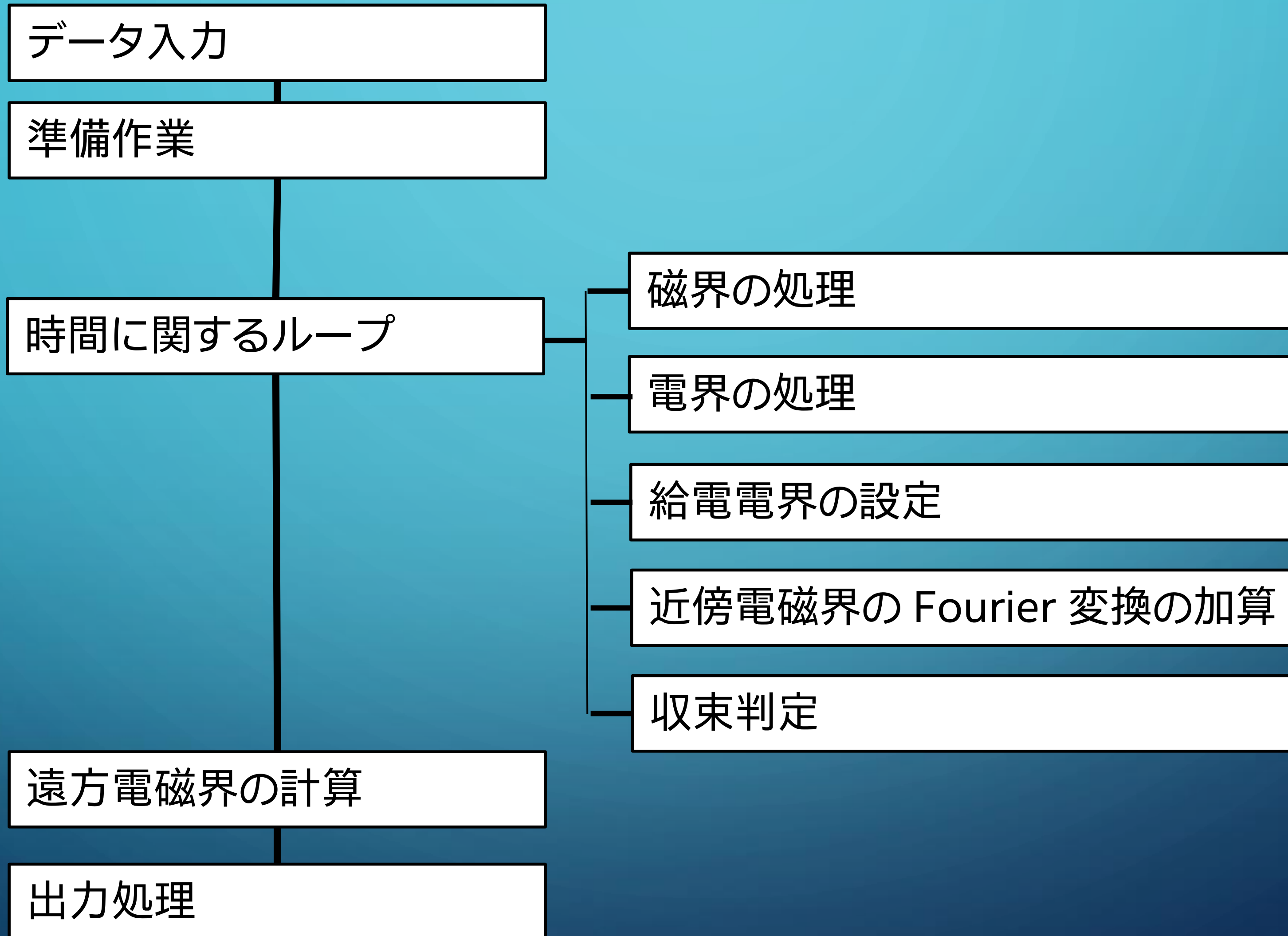
```
/* hello.cpp */

#include <CL/sycl.hpp>
using namespace sycl;

int main()
{
  queue myQ;
  range<2> whole = {2,8};
  range<2> block = {1,4};

  myQ.submit([&](handler& h){
    stream out = stream(1024, 768, h);
    h.parallel_for(nd_range<2>(whole, block),[=](nd_item<2> i){
     int t = i.get_local_id(1);
     int bx = i.get_group(1);
     int by = i.get_group(0);
     sycl::ext::oneapi::experimental::\
      printf("(%d,%d) Th %d : Hello World!\n",bx,by,t);});
  });
  myQ.wait();
}
```



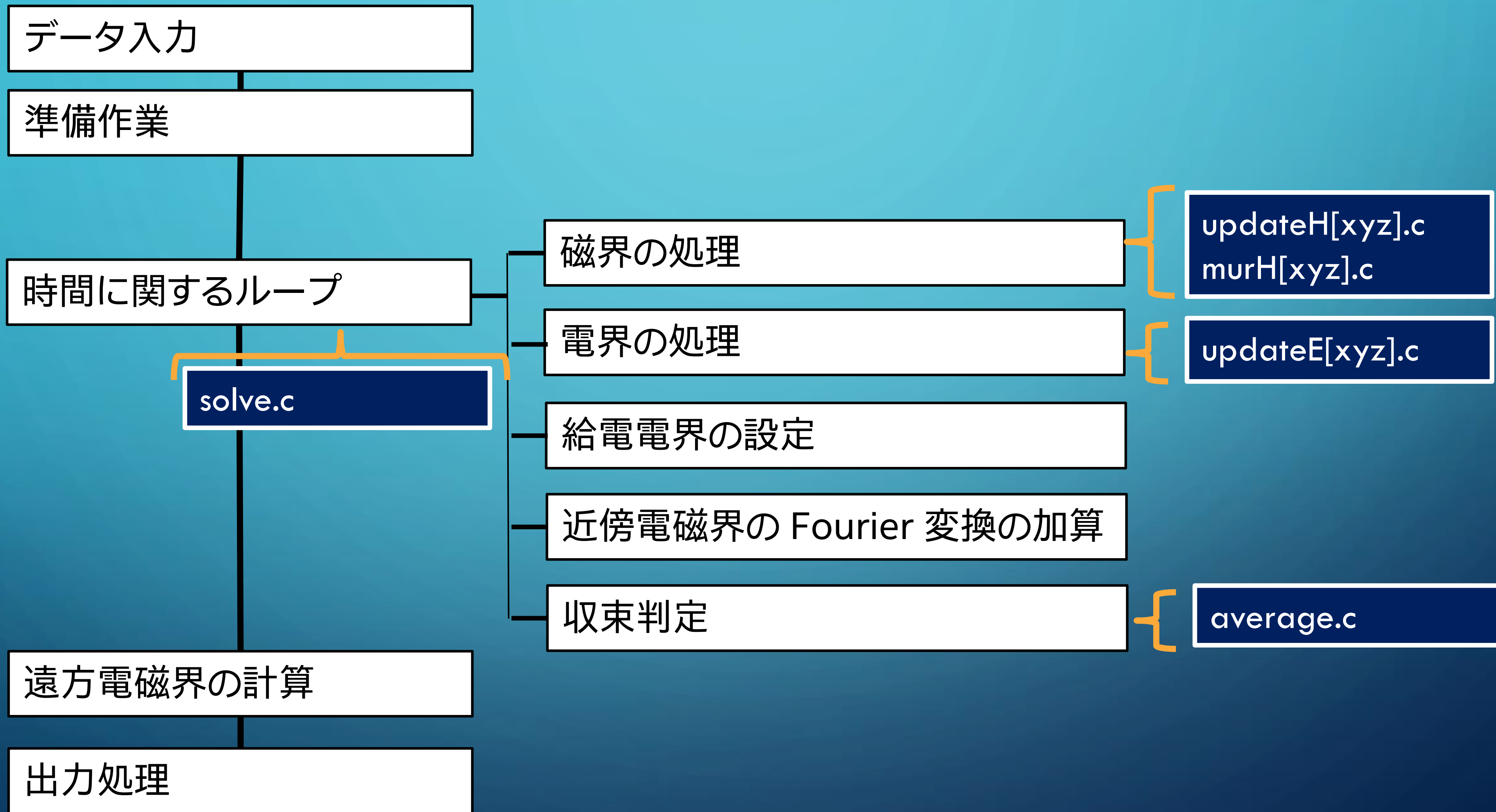
OpenFDTD を GPU にオフロードする際の方針

- OpenFDTD の C プログラム群をそのまま C++ のプログラムとしてコンパイル
- オフロードする関数は CUDA* 版でオフロードされているものが対象
- SYCL* でラムダ式に渡せないグローバル変数はローカル変数にコピーして、代わりにこれを渡す
- OpenFDTD で作成された 3 次元の計算モデルがマップされた 1 次元のメモリー空間を共有メモリー空間にして、GPU ではこれを参照する
- GPU にオフロードするプログラムは、オリジナルの計算ループの隣に挿入し、コンパイル時に if_def などでオフロードするかどうかを選択する

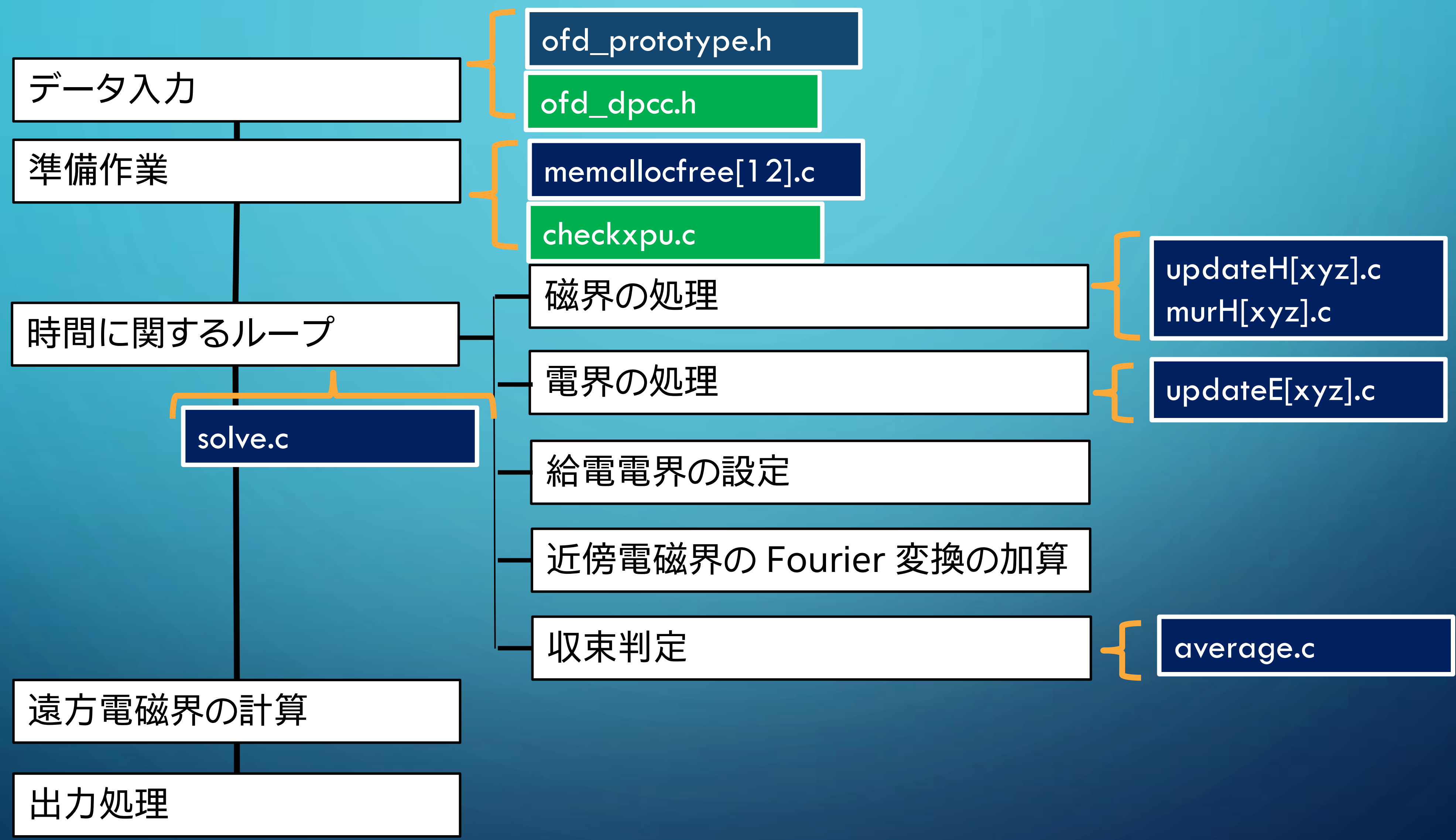
OpenFDTD プログラム計算処理流れ



GPU にオフロードした計算部分



SYCL* を用いるために変更や付加したプログラム

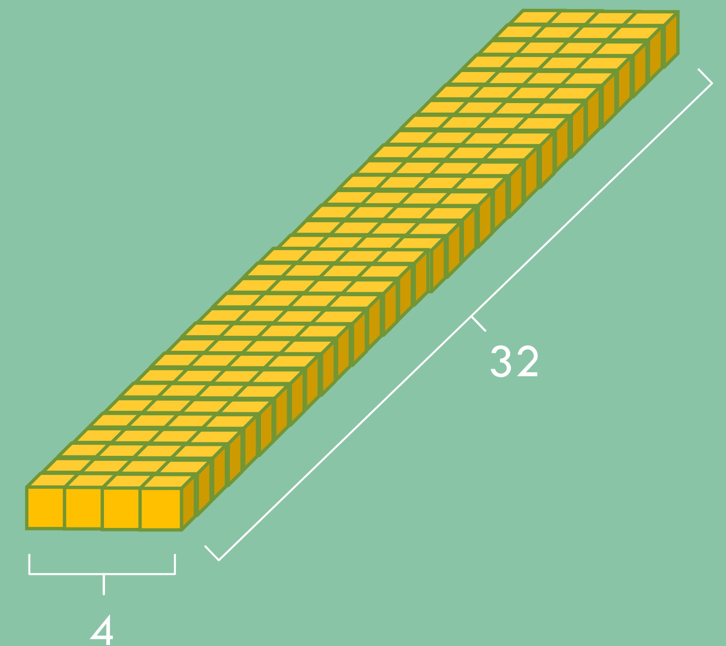


磁界の処理 – updateHx.c

(1) nd_range<3> を用いた分割方法の指定

- CUDA* プログラムでの分割評価の結果を参考に、(1,4,32) を演算グループとして用いることにした

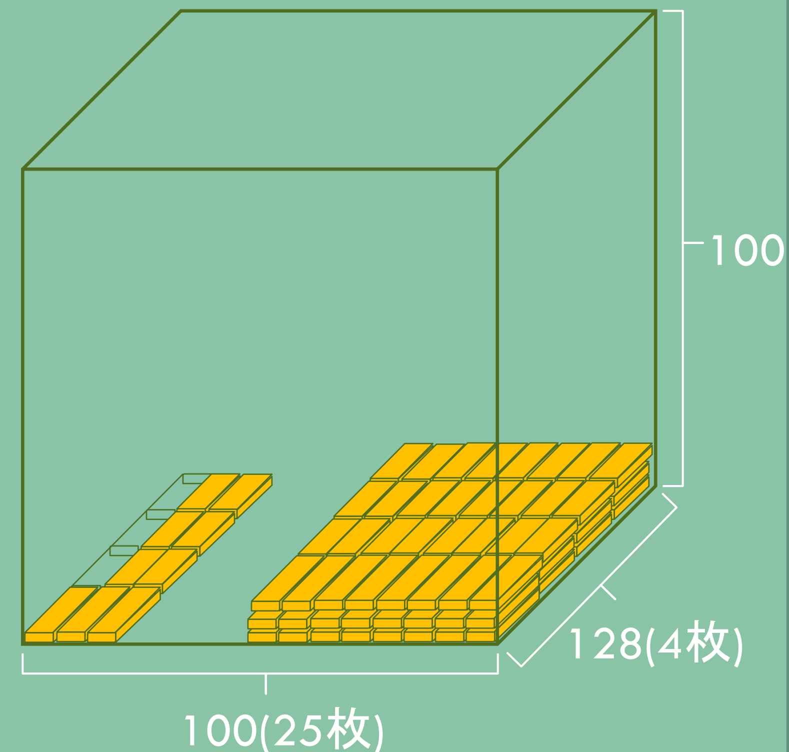
```
static void updateHx_f2(void)
{
    assert(Nk == 1);

    updateBlock = sycl::range<3>(1, 4, 32)
    sycl::range<3> grid(CEIL(iMax - iMin + 1, updateBlock[0]),
        CEIL(jMax - jMin + 0, updateBlock[1]),
        CEIL(kMax - kMin + 0, updateBlock[2]));
    sycl::range<3> all_grid = grid * updateBlock;

    myQ.submit([&](sycl::handler& cgh) {
        auto iMin_l = iMin;
        auto jMin_l = jMin;
        auto kMin_l = kMin;
        auto iMax_l = iMax;
        auto jMax_l = jMax;
        auto kMax_l = kMax;
        auto N0_l = N0;
        auto Ni_l = Ni;
        auto Nj_l = Nj;
        auto Nk_l = Nk;
        auto Hx_l = Hx;
        …
```



磁界の処理 – updateHx.c

(1) nd_range<3> を用いた分割方法の指定
- 全 grid のブロックは、grid (100,25,4) となるので、CUDA* では <<< grid, updateBlock >>>

```
static void updateHx_f2(void)
{
    assert(Nk == 1);

    updateBlock = sycl::range<3>(1, 4, 32);
    sycl::range<3> grid(CEIL(iMax - iMin + 1, updateBlock[0]),
        CEIL(jMax - jMin + 0, updateBlock[1]),
        CEIL(kMax - kMin + 0, updateBlock[2]));
    sycl::range<3> all_grid = grid * updateBlock;

    myQ.submit([&](sycl::handler& cgh) {
        auto iMin_l = iMin;
        auto jMin_l = jMin;
        auto kMin_l = kMin;
        auto iMax_l = iMax;
        auto jMax_l = jMax;
        auto kMax_l = kMax;
        auto N0_l = N0;
        auto Ni_l = Ni;
        auto Nj_l = Nj;
        auto Nk_l = Nk;
        auto Hx_l = Hx;
        …
```



磁界の処理 – updateHx.c

(2) グローバル変数のローカル変数への代入

- DPC++ では、グローバル変数をそのまま加速器側へのサブミットに使用できないので、ローカル変数にコピー

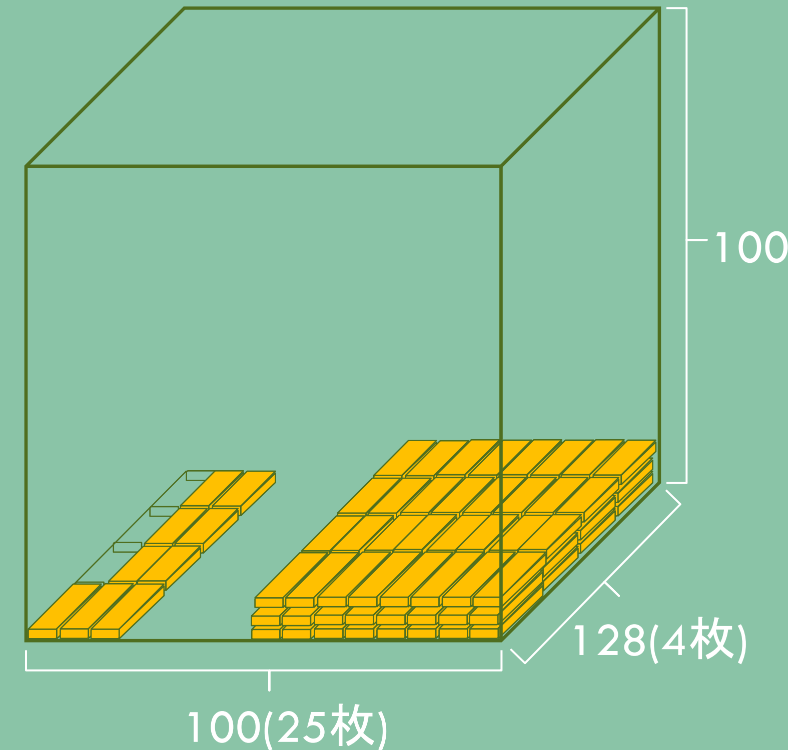
```
…
sycl::range<3> all_grid = grid * updateBlock;

myQ.submit([&](sycl::handler& cgh) {
    auto iMin_l = iMin;
    auto jMin_l = jMin;
    auto kMin_l = kMin;
    auto iMax_l = iMax;
    auto jMax_l = jMax;
    auto kMax_l = kMax;
    auto N0_l = N0;
    auto Ni_l = Ni;
    auto Nj_l = Nj;
    auto Nk_l = Nk;
    auto Hx_l = Hx;
    auto Ey_l = Ey;
    auto Ez_l = Ez;
    auto iHx_l = iHx;
    auto d1 = D1;
    auto d2 = D2;
    auto ryc = RYc;
    auto rzc = RZc;
    cgh.parallel_for(
```

磁界の処理 – updateHx.c

(3) parallel_for(nd_range<3>(all_grid, update_block), … で並列処理を記述

- USM を用いればオリジナルの for_loop を書き換えるだけで並列処理を行える

```
  …
  auto rzc = RZc;
  cgh.parallel_for(
    sycl::nd_range<3>(all_grid, updateBlock),
    [=](sycl::nd_item<3> item_ct1) {
        auto i = iMin_l + item_ct1.get_global_id(0);
        auto j = jMin_l + item_ct1.get_global_id(1);
        auto k = kMin_l + item_ct1.get_global_id(2);
        if ((i <= iMax_l) &&
            (j < jMax_l) &&
            (k < kMax_l)) {
            const int64_t n = NA2(i, j, k);
            const id_t m = iHx_l[n];
            Hx_l[n] = d1[m] * Hx_l[n]
                - d2[m] * (ryc[j] * (Ez_l[n + Nj_l] - Ez_l[n])
                    - rzc[k] * (Ey_l[n + Nk_l] - Ey_l[n]));
        }
    });
  });
    myQ.wait();
}
```



磁界の処理 – updateHx.c

(3) parallel_for(nd_range<3>(all_grid, update_block)、… で並列処理を記述

- USM を用いればオリジナルの for_loop を書き換えるだけで並列処理を行える

```
  …
  auto rzc = RZc;
  cgh.parallel_for(
    sycl::nd_range<3>(all_grid, updateBlock),
    [=](sycl::nd_item<3> item_ct1) {
        auto i = iMin_l + item_ct1.get_global_id(0);
        auto j = jMin_l + item_ct1.get_global_id(1);
        auto k = kMin_l + item_ct1.get_global_id(2);
        if ((i <= iMax_l) &&
            (j < jMax_l) &&
            (k < kMax_l)) {
            const int64_t n = NA2(i, j, k);
            const id_t m = iHx_l[n];
            Hx_l[n] = d1[m] * Hx_l[n]
                - d2[m] * (ryc[j] * (Ez_l[n + Nj_l] - Ez_l[n])
                    - rzc[k] * (Ey_l[n + Nk_l] - Ey_l[n]));
        }
    });
  });
    myQ.wait();
}
```

update_block(1,4,32)
all_grid(100,100,128)

<オリジナルの for_loop>

```
for (     i = iMin; i <= iMax; i++) {
  for (int j = jMin; j <  jMax; j++) {
    int64_t n = NA(i, j, kMin);
    int64_t n1 = n + Nj;
    int64_t n2 = n + Nk;
    for (int k = kMin; k <  kMax; k++) {
      Hx[n] = D1[iHx[n]] * Hx[n]
       - D2[iHx[n]] * (RYc[j] * (Ez[n1] - Ez[n])
       - RZc[k] * (Ey[n2] - Ey[n]));
      n++;
      n1++;
      n2++;
    }
  }
}
```

収束判定 – average.c

nd_range<3> を用いた分割方法 (1, 8, 32)
(1) parallel_for の work_group 別に加算を行うために電界および磁界用共有メモリーと G 内共有メモリーを確保

```
sumBlock = sycl::range<3>(1, 8, 32);
sumGrid=sycl::range<3>(CEIL(Nx,sumBlock[0]),CEIL(Ny,sumBlock[1]),CEIL(Nz,sumBlock[2]));
sycl::range<3> all_grid = sumGrid * sumBlock;
const int wgroup_size = sumBlock[2] * sumBlock[1] * sumBlock[0];
const int wgroup_num = sumGrid[2] * sumGrid[1] * sumGrid[0];
float* sumH = static_cast<float*> malloc_shm(wgroup_num * sizeof(float));
Float* sumE = static_cast<float*> malloc_shm(wgroup_num * sizeof(float));

myQ.submit([&](sycl::handler& cgh) {
auto iMin_l = iMin;
auto iMax_l = iMax;
auto jMax_l = jMax;
auto kMax_l = kMax;
auto N0_l = N0;
auto Ni_l = Ni;
auto Nj_l = Nj;
…
sycl::local_accessor<real_t> se(wgroup_size, cgh);
sycl::local_accessor<real_t> sh(wgroup_size, cgh);
```

収束判定 – average.c

nd_range<3> を用いた分割方法 (1, 8, 32)
(1) parallel_for の work_group 別に加算を行うために電界および磁界用共有メモリーと G 内共有メモリーを確保

```
sumBlock = sycl::range<3>(1, 8, 32);
sumGrid=sycl::range<3>(CEIL(Nx,sumBlock[0]),CEIL(Ny,sumBlock[1]),CEIL(Nz,sumBlock[2]));
sycl::range<3> all_grid = sumGrid * sumBlock;
const int wgroup_size = sumBlock[2] * sumBlock[1] * sumBlock[0];
const int wgroup num = sumGrid[2] * sumGrid[1] * sumGrid[0];
float* sumH = static_cast<float*> malloc_shm(wgroup_num * sizeof(float));
Float* sumE = static_cast<float*> malloc_shm(wgroup_num * sizeof(float));

myQ.submit([&](sycl::handler& cgh) {
auto iMin_l = iMin;
auto iMax_l = iMax;
auto jMax_l = jMax;
auto kMax_l = kMax;
auto N0_l = N0;
auto Ni_l = Ni;
auto Nj_l = Nj;
…
sycl::local_accessor<real_t> se(wgroup_size, cgh);
sycl::local accessor<real t> sh(wgroup size, cgh);
```

収束判定 – average.c

parallel_for(nd_range<3>, ... の work_group 内の共有メモリー (local_memory) を使用して reduction

```
cgh.parallel_for( sycl::nd_range<3>(all_grid, sumBlock),
[=](sycl::nd_item<3> item_ct1) {
const int i = item_ct1.get_global_id(0);
const int j = item_ct1.get_global_id(1);
const int k = item_ct1.get_global_id(2);
const int tid = item_ct1.get_local_linear_id();
const int bid = item_ct1.get_group_linear_id();

if ((i < iMax_l) && (j < jMax_l) && (k < kMax_l)) {
  se[tid] = fabs(EX2(i, j, k) + EX2(i, j+1, k) + EX2(i, j, k+1) + EX2(i, j + 1, k+1))
    + fabs(EY2(i, j, k) + EY2(i+1, j, k) + EY2(i, j, k + 1) + EY2(i+1, j, k + 1))
    + fabs(EZ2(i, j, k) + EZ2(i, j + 1, k) + EZ2(i+1, j, k) + EZ2(i+1, j + 1, k));
  sh[tid] = fabs(HX2(i, j, k) + HX2(i + 1, j, k)) + fabs(HY2(i, j, k) + HY2(i, j + 1, k))
    + fabs(HZ2(i, j, k) + HZ2(i, j, k + 1));
}
else {
  se[tid] = 0;
  sh[tid] = 0;
}
```

収束判定 – average.c

parallel_for(nd_range<3>, ... の work_group 内の共有メモリー (local_memory) を使用してスレッド毎に演算

```
cgh.parallel_for( sycl::nd_range<3>(all_grid, sumBlock),
[=](sycl::nd_item<3> item_ct1) {
const int i = item_ct1.get_global_id(0);
const int j = item_ct1.get_global_id(1);
const int k = item_ct1.get_global_id(2);
const int tid = item_ct1.get_local_linear_id();
const int bid = item_ct1.get_group_linear_id();

if ((i < iMax_l)
  se[tid] = fabs(
    + fabs(EY2(i,
    + fabs(EZ2(i,
  sh[tid] = fabs(
    + fabs(HZ2(i,
}
else {
  se[tid] = 0;
  sh[tid] = 0;
}
```

<オリジナルの for_loop>

```
for (     i = iMin; i < iMax; i++) {
  for (int j = jMin; j < jMax; j++) {
    for (int k = kMin; k < kMax; k++) {
      se += fabs( EX(i, j, k) + EX(i, j+1, k) + EX(i, j, k + 1) + EX(i, j+1, k+1))
        + fabs( EY(i, j, k) + EY(i, j, k+1) + EY(i+1, j, k) + EY(i+1, j, k+1))
        + fabs( EZ(i, j, k) + EZ(i+1, j, k) + EZ(i, j+1, k) + EZ(i+1, j+1, k));
      sh += fabs( HX(i, j, k) + HX(i+1, j, k))+ fabs( HY(i, j, k)   + HY(i, j+1, k))
        + fabs( HZ(i, j, k) + HZ(i, j, k + 1));
    }
  }
}
  fsum[0] = se / (4.0 * Nx * Ny * Nz);
  fsum[1] = sh / (2.0 * Nx * Ny * Nz);
}
```

収束判定 – average.c

ホスト側でUSMの sumE[ ] と sumH[ ] を使用して逐次的に加算してreduction

```
…
else {
    se[tid] = 0;
    sh[tid] = 0;
}

sumE[bid] = joint_reduce(item_ct1.get_group(), &se[0], &se[wgroup_size], sycl::plus<>());
sumH[bid] = joint_reduce(item_ct1.get_group(), &sh[0], &sh[wgroup_size], sycl::plus<>());
    });
});
myQ.wait();

for (int i = 0; i < wgroup num; i++) {
 se += sumE[i];
 sh += sumH[i];
}
```

収束判定 – average.c

parallel_for(nd_range<3>, ... の work_group 内の local_memory se[ ]､sh[ ] を使用して reduction

```
…
else {
    se[tid] = 0;
    sh[tid] = 0;
}

sumE[bid] = joint_reduce(item_ct1.get_group(), &se[0], &se[wgroup_size], sycl::plus<>());
sumH[bid] = joint_reduce(item_ct1.get_group(), &sh[0], &sh[wgroup_size], sycl::plus<>());
    });
});
myQ.wait();

for (int i = 0; i < wgroup num; i++) {
 se += sumE[i];
 sh += sumH[i];
}
```

SYCL* を用いた GPU オフロードプログラム

- CUDA* 版は 2 種類のソースコードを使用する必要がある

項目	SYCL* 版	CUDA* 版プログラム	オリジナル
C/C++ ファイル数	109 (22)	62	105
CUDA* プログラム数	0	44 (22)	0
ヘッダー数	6	6	5
全ファイル数	115	112	110

- オフロードするための並列記述に必要なプログラム行数には大きな違いはないが、CUDA* 版ではメモリー管理により多くの行数を必要としている

プログラム名	SYCL* 版の増加行数	CUDA* 版プログラムの行数	オリジナルの行数
updateHx	144	122	163
average	77	102	44
memallocfree2	60	149	61

オフロードの性能比較 (100x100x100)

- OpenFDTD に付属のベンチマーク 100 の実行時間の比較

実行ハードウェア	プログラム	実行時間
インテル® Core™ i3-1200 プロセッサー (3.30GHz P-core x 4) インテル® UHD グラフィックス 730 (1.40GHz MHz EU x24)	OpenMP*	16.0 秒
	SYCL*	13.6 秒
インテル® Core™ i7-11370H プロセッサー (3.30GHz WC x 4) インテル® Xe グラフィックス (1.30GHz EU x 96)	OpenMP*	21.3 秒
	SYCL*	11.5 秒

- CUDA* 版は DFT 部分もオフロードしている

実行ハードウェア	プログラム	実行時間
インテル® Core™ i3-1200 プロセッサー (3.30GHz P-core x 4) NVIDIA* GeForce* GT 1030 (1.25GHz コア x 384 GDDR5 2GB)	CUDA*	8.5 秒

オフロードの性能比較 (200x200x200)

- OpenFDTD に付属のベンチマーク 200 の実行時間の比較

実行ハードウェア	プログラム	実行時間
インテル® Core™ i3-1200 プロセッサー (3.30GHz P-core x 4) インテル® UHD グラフィックス 730 (1.40GHz MHz EU x24)	OpenMP*	115 秒
	SYCL*	83.1 秒
インテル® Core™ i7-11370H プロセッサー (3.30GHz WC x 4) インテル® X^e グラフィックス (1.30GHz EU x 96)	OpenMP*	139 秒
	SYCL*	54.0 秒

- CUDA* 版は DFT 部分もオフロードしている

実行ハードウェア	プログラム	実行時間
インテル® Core™ i3-1200 プロセッサー (3.30GHz P-core x 4) NVIDIA* GeForce* GT 1030 (1.25GHz コア x 384 GDDR5 2GB)	CUDA*	62.1 秒

SYCL* を用いた GPU オフロードプログラムまとめ

FDTD 法のシミュレーション･プログラム OpenFDTD で SYCL* による演算プログラムの GPU オフロードの有用性を確認した

- オフロードには nd_range を用い、CUDA* を用いたオフロードと同等なことを単一の C++ のソースコードで行える
- USM を採用して並列化対象ループから比較的容易に、演算データの管理部分を最小限にしたオフロードプログラムを生成できる
- リダクション演算においては、演算グループ内の共用メモリーを用いて高速な演算が可能
- 内蔵 GPU を持つ CPU では、CPU 単独の 2 倍以上の性能を実現できた。この性能は外付け GPU を用いて CUDA* 版のプログラムでオフロードしたものと比較して同等程度

今後の課題

- OpenFDTD の CUDA* 版のオフロードプログラムのうち、まだ SYCL* で実装できていないものがあり、これらの実装
- 性能評価に使用した GPU は、このような目的のオフロードに使用するには最小限のものばかりであり、より高性能なシステムにおける性能評価

参考文献

[1] 宇野亨, “FDTD法による電磁界およびアンテナ解析”, コロナ社, 1998

[2] OpenMP, Home – OpenMP (英語)

[3] Message Passing Interface Forum, MPI Forum (mpi-forum.org) (英語)

[4] NVIDIA, “CUDA Zone”, https://developer.nvidia.com/cuda-zone/ (英語)

[5] IA Software User Society, “oneAPI DPC++ 導入ガイド”, oneAPI DPC++ 導入ガイド | iSUS

[6] Khronos, “Sycl”, SYCL Overview - The Khronos Group Inc (英語)

[7] Intel, “oneAPI”, oneAPI: A New Era of Heterogeneous Computing (intel.com) (英語)

[8] LLVM “The LLVM Compiler Infrastructure”, Clang C Language Family Frontend for LLVM (英語)

Backup

磁界の処理 – murHx.c

同様に USM を用いればオリジナルの for_loop を書き換えるだけで並列処理を行える

```
const int murBlock = 256;
updateBlock = sycl::range<1>(murBlock);
sycl::range<1> grid(CEIL(numMurHx, murBlock));
sycl::range<1> all_grid = grid * updateBlock;
```

<オリジナルの for_loop>

```
int64_t n;
for (n = 0; n < numMurHx; n++) {
  const int i  = fMurHx[n].i;
  const int j  = fMurHx[n].j;
  const int k  = fMurHx[n].k;
  const int i1 = fMurHx[n].i1;
  const int j1 = fMurHx[n].j1;
  const int k1 = fMurHx[n].k1;
  HX(i, j, k) = fMurHx[n].f
  + fMurHx[n].g * (HX(i1, j1, k1) - HX(i, j, k));
  fMurHx[n].f = HX(i1, j1, k1);
  }
}
```

```
myQ.submit([&](sycl::handler& cgh) {
  auto numMurHx_l = numMurHx;
  auto fMurHx_l = fMurHx;
  auto Hx_l = Hx;
  auto Ni_l = Ni;
  auto Nj_l = Nj;
  auto Nk_l = Nk;
  auto N0_l = N0;
  cgh.parallel_for(
    sycl::nd_range<1>(all_grid, updateBlock),
     [=](sycl::nd_item<1> item_ct1) {
    const int64_t n = item_ct1.get_global_id(0);
      if (n < numMurHx_l) {
      const int i = fMurHx_l[n].i;
      const int j = fMurHx_l[n].j;
      const int k = fMurHx_l[n].k;
      const int i1 = fMurHx_l[n].i1;
      const int j1 = fMurHx_l[n].j1;
      const int k1 = fMurHx_l[n].k1;
      HX2(i, j, k) = fMurHx_l[n].f
        + fMurHx_l[n].g * (HX2(i1, j1, k1) - HX2(i, j, k));
        fMurHx_l[n].f = HX2(i1, j1, k1);
      }
  });
});
```