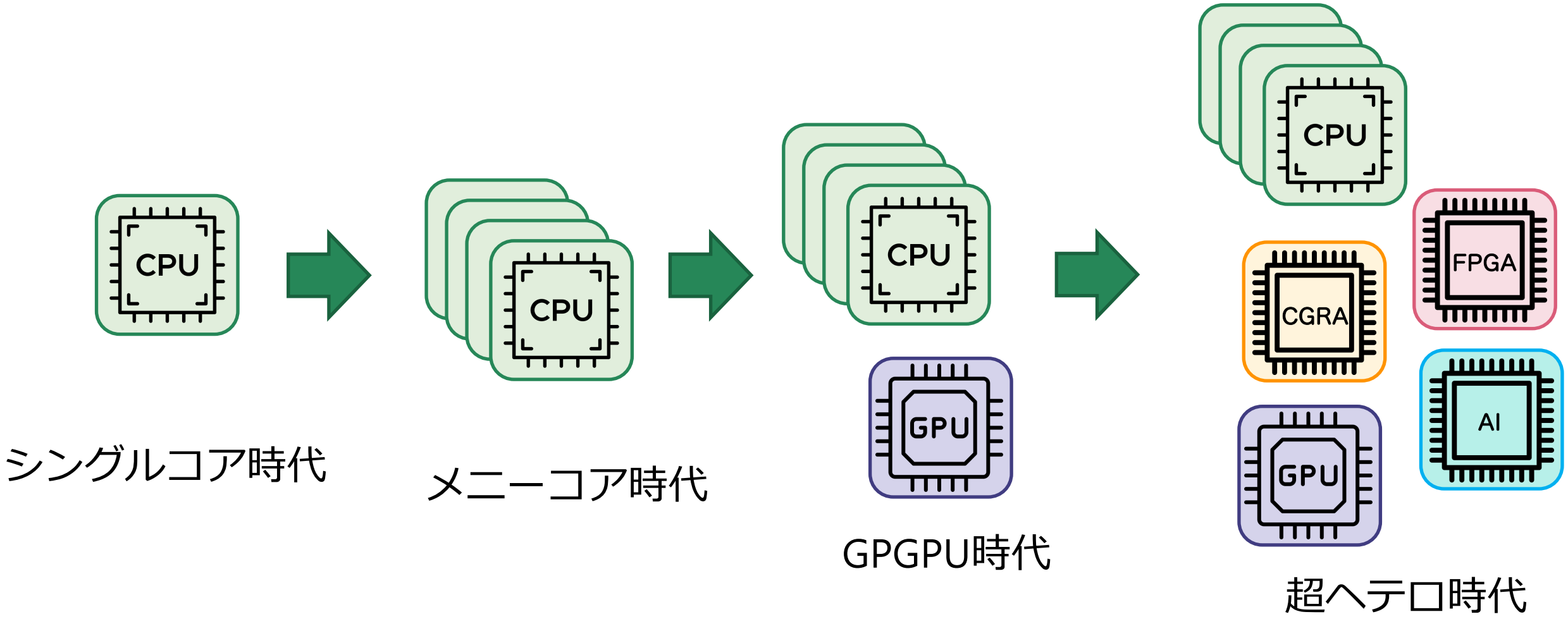
LLVMにおけるOpenMP GPUオフローディングの性能調査

小島 拓也
東京大学 情報理工学系研究科

ヘテロジニアス化する計算プラットフォーム

- 性能スケーリング、高エネルギー効率化を目指し、計算機システムはますます複雑化



ヘテロジニアスコンピューティングを取り巻くプログラミング環境

- 低レイヤAPIによるプログラミング
  - CUDA:　2007年 NVIDA社
  - OpenCL: 2008年 Khronos Group
  - ソースコード移植性の低さが課題
- シングルソースのプログラミング環境が登場
  - SYCL: Khronos Groupが策定する抽象化レイヤ
    - 実装: Intel oneAPIのDPC++(2020), 東北大 neoSYCL(2022)など
  - OpenACC: directiveベースのアクセラレータ用API
    - 2011年 version 1.0の仕様策定
  - OpenMP: 共有メモリ型の並列プログラミングAPI
    - 元々はマルチコアCPU向け
    - 2013年 version 4.0よりアクセラレータオフローディングをサポート

本研究による報告

- OpenMPによるGPUオフローディングの性能調査
  - コンパイラ基盤LLVMにおける実装の把握
  - NVIDIA社、AMD社のGPUを用いて評価
  - CUDAおよびOpenCLとの比較を実施
- 以降の流れ
  - OpenMPによるGPUオフローディング
  - LLVMにおける実装
  - 評価と比較
  - 結論

OpenMPによるGPUオフローディング

OpenMPの基本

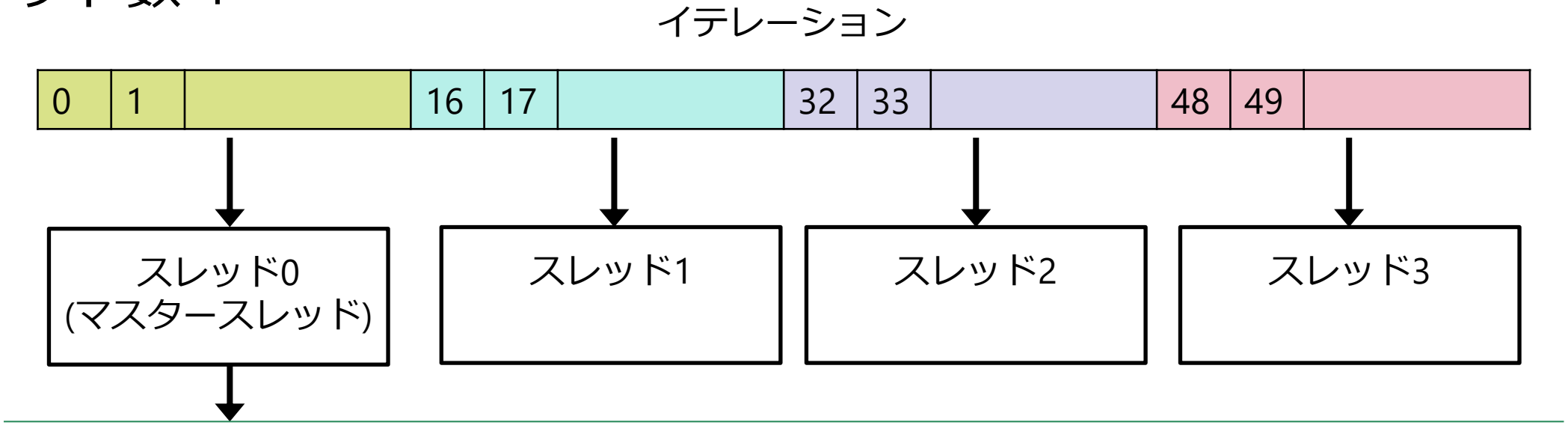
- OpenMPのディレクティブは全てompプラグマを使用
- Parallel構文により*team*(複数のスレッド)が生成
- ワークシェア構文を用いて構造化ブロックをスレッドで実行
  - For構文はforループを複数のスレッドで分割して実行

```
void saxpy(int N, const float A, float *X, float *Y)
{
  #pragma omp parallel for
  for (i = 1; i < N - 1; i++) {
    Y[i] = A * X[i] + Y[i];
  }
}
```

forループのマルチスレッド化の例

ワークシェアリングの例

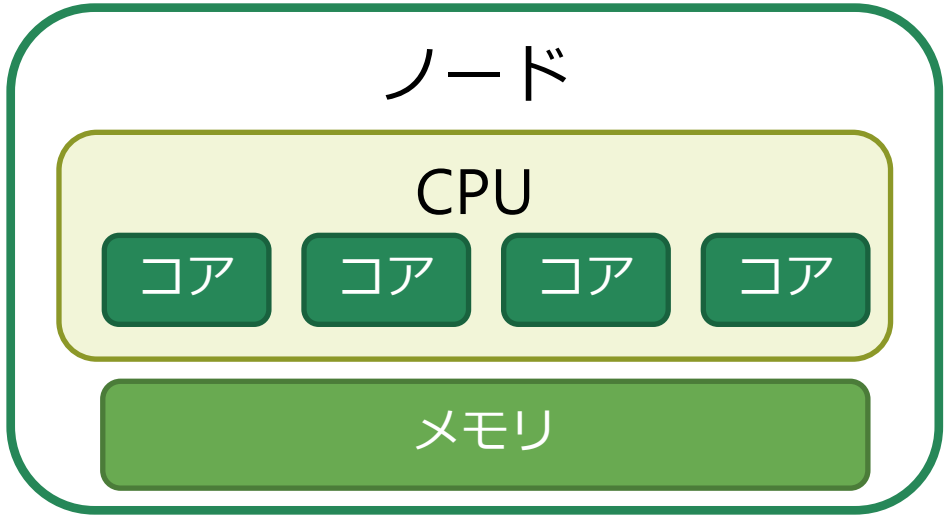
- ループサイズ N = 64
- スレッド数 4



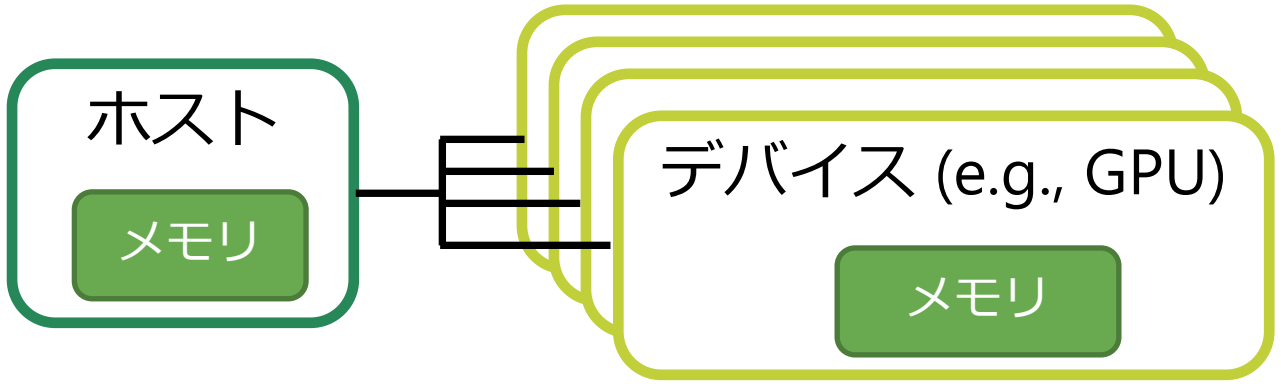
- デフォルトでは均等に分割、順番に割り付け
  - Chunk句でサイズ、schedule句で割り付け方法を変更可能

Open4.0以降のデバイスモデル

- OpenMP 3.1まで
  - 共有メモリ型のモデルゆえにノード内並列化が前提
- ターゲットデバイスのモデル
  - Ver 4.0以降で導入
  - 一つのホストと複数のデバイス
  - デバイスはホストと異なるアドレス空間のメモリを保有



OpenMP 3.1以前



OpenMP 4.0以降のホスト-デバイスモデル

オフローディングのためのTarget構文

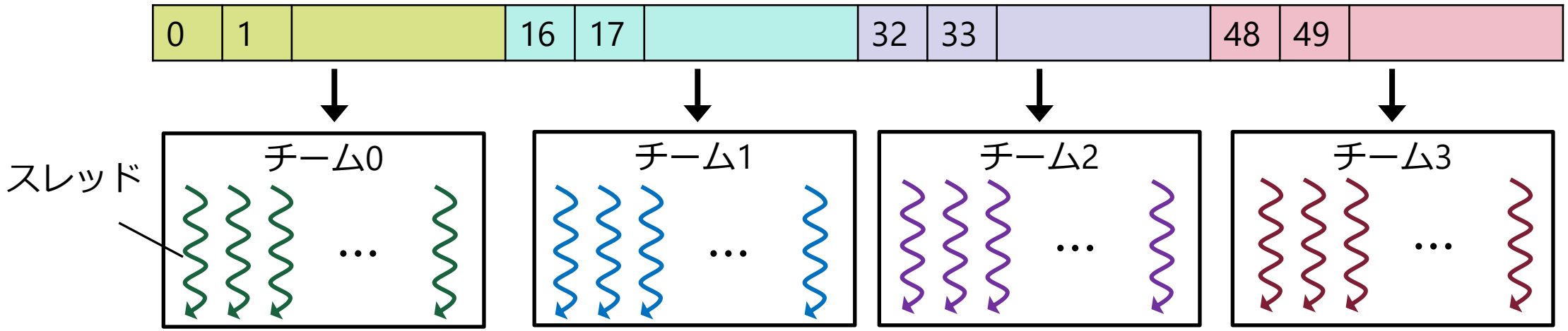
- デバイスオフローディングにはTarget構文を使う
- Target構文
  - ターゲット領域を作成
  - デバイス側で実行される

- 変数のマッピング
  - 明示的なデータ転送の指示が必要
  - map句でマッピングおよび転送方向の指定
  - 配列の場合は長さも指定

```
#pragma omp target data map(to:A[0:N]) map(tofrom:B[0:N])
{
  //構造化ブロック
}
```

Teams構文によるスレッドの階層化

- Parallel For構文では階層化されたスレッドができない
  - GPUなどの計算資源やメモリ階層化
  - CUDAでは階層化したスレッドモデル (grid, block, thread)
- Teams distribute構文
  - 複数の*team*をまとめた*League*を生成
  - チーム間はバリア同期しない



CUDAとOpenMPのコード比較

OpenMPオフローディング

```
void saxpy(int N, const float A, float *X,
           float *Y)
{
  #pragma omp target data map(to: X[0:N])
map(tofrom:Y[0:N])
  #pragma omp target teams distribute parallel
for thread_limit(256)
  for (i = 0; i < N; i++) {
    Y[i] = A * X[i] + Y[i];
  }
}
```

- thread_limit句でteam(block in CUDA)内のスレッド数上限を指定

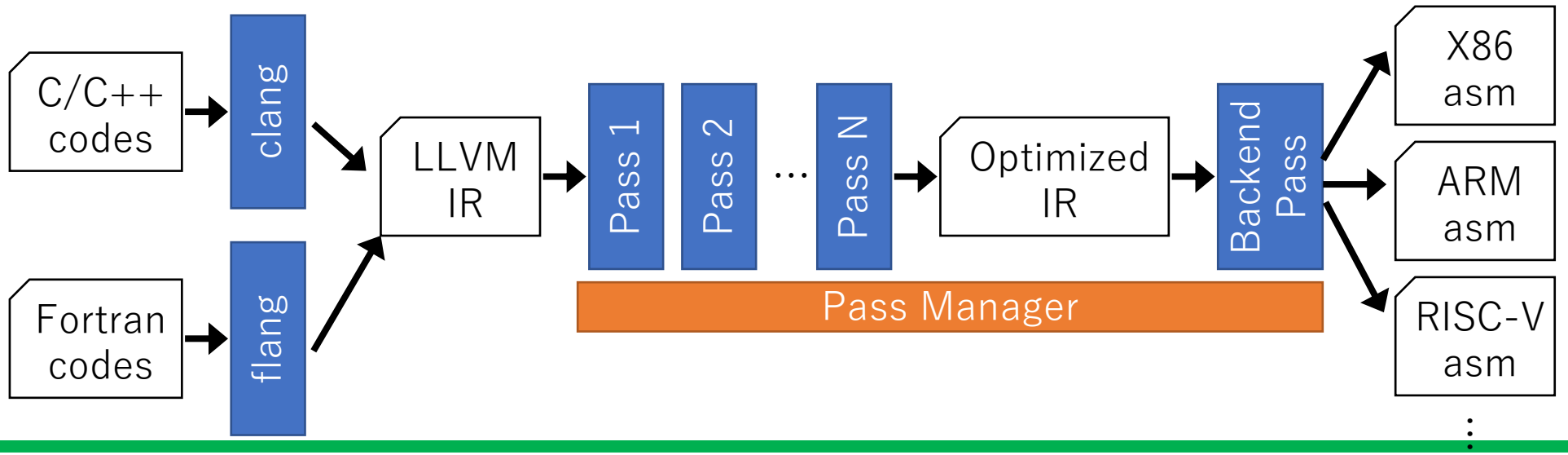
CUDA

```
__global__ void saxpy(float A, float *X,
                      float *Y)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  Y[i] = A * X[i] + Y[i];
}
void host(...)
{ //...省略: 配列の初期化など
  float *X_gpu, *Y_gpu;
  cudaMalloc(&X_gpu, N * sizeof(float));
  cudaMalloc(&Y_gpu, N * sizeof(float));
  cudaMemcpy(X_gpu, X, N * sizeof(float), …);
  cudaMemcpy(Y_gpu, Y, N * sizeof(float), …);
  saxpy<<<N/256,256>>>(N, A, X_gpu, Y_gpu);
  cudaMemcpy(Y, Y_gpu, N * sizeof(float), …);
  cudaFree(X_gpu);
  cudaFree(Y_gpu);
  //...省略
```

LLVMによるOpenMP実装および環境構築

コンパイラ基盤LLVM

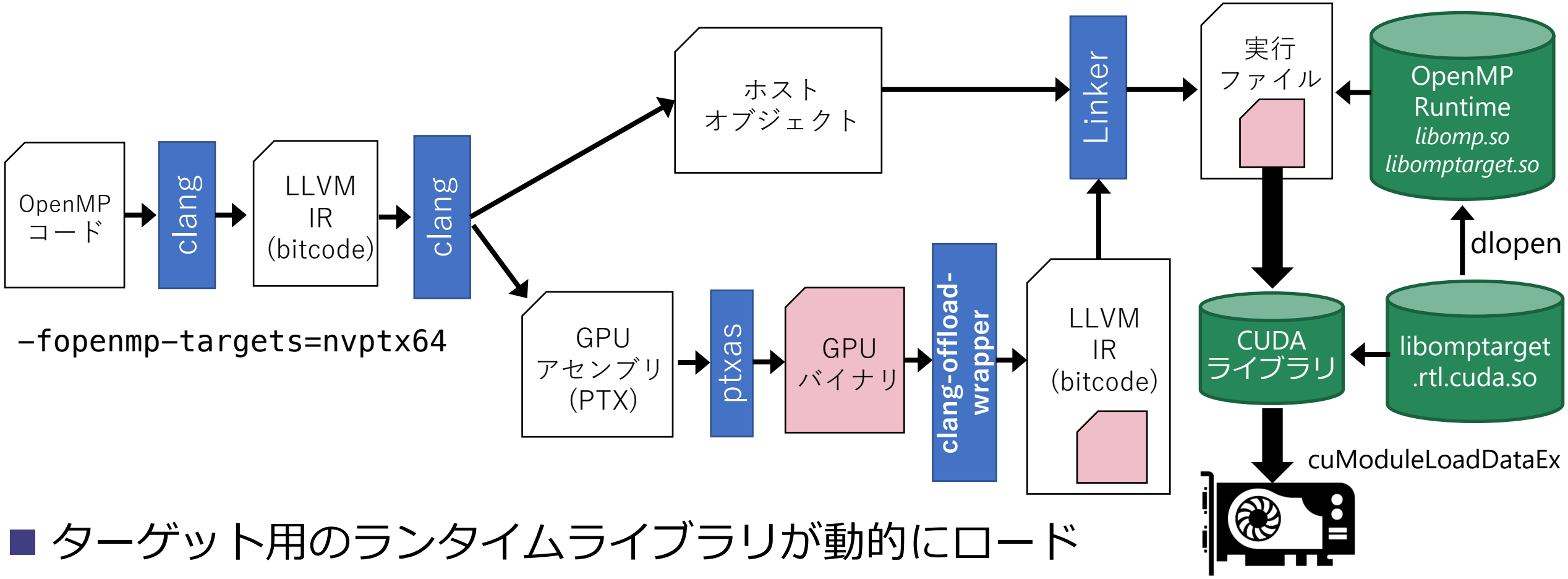
- LLVM: オープンソースコンパイラ基盤
  - マシン非依存な中間表現 LLVM-IR
  - コンパイラに共通の最適化、解析機能 (Pass)
  - さまざまな公式サブプロジェクト
    - Cフロントエンド Clang, FortranフロントエンドFlang, **OpenMP**, etc
  - 外部プロジェクトも多数



LLVMのOpenMPターゲットライブラリ

- LLVMにおけるランタイムライブラリは徐々にサポートを拡大
  - Version 5.0 in 2017
    - NVIDIA GPU用ライブラリ
  - Version 11.0 in 2020
    - NEC SX-Aurora用ライブラリ
  - Version 13.0 in 2021
    - AMD GPU用ライブラリ
- 余談
  - GCCによるAMD GPUへのオフローディングにはLLVMの実装をバックエンドに利用 (GCC 12時点)

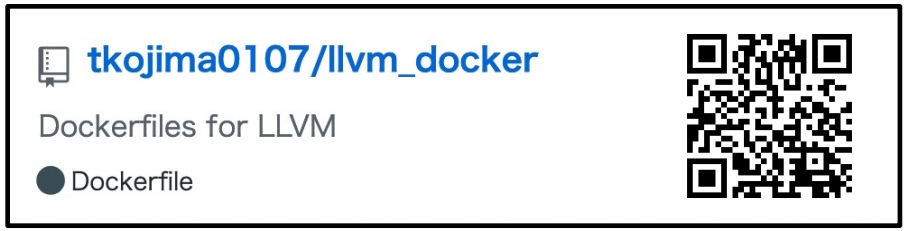
NVIDA GPU用のコンパイルフロー



- ターゲット用のランタイムライブラリが動的にロード
  - CUDA APIを呼び出すラッパー
  - AMD GPUの場合はHSAランタイムを呼び出す

実行環境のDockerコンテナ化

- LLVMのフルセット+OpenMPターゲットライブラリをビルド
- GithubおよびDockerHubで公開中



https://github.com/tkojima0107/llvm_docker/

- バリエーション
  - LLVMバージョン: 12.0.1, 13.0.1, 14.0.6
  - ベースイメージ:　CentOS8, Ubuntu20.04
  - ライブラリ群:
    1. LLVMのみ
    2. LLVM+CUDA+nvidia GPU向けターゲットランタイム
    3. LLVM+ROCmライブラリ+AMD GPU向けターゲットランタイム

使用例およびコンパイルオプション

```
[tkojima:~ @liver]$ docker run -it tkojima0107/llvm14-nvidia-centos
[root@876672432da8 ~]# vi veccopy.cpp
[root@876672432da8 ~]# clang++ -O3 -Xopenmp-target -march=sm_75 -fopenmp -fopenm
p-targets=nvptx64 --cuda-path=/usr/local/cuda -fopenmp-version=50 veccopy.cpp
[root@876672432da8 ~]# ./a.out
correct
[root@876672432da8 ~]#
```

- NVIDA GPU用の重要なオプション
  - -march=sm_*XX*: 対象のGPUアーキテクチャを指定
  - --cuda-path: CUDAのパス
- つまり、GPUの世代が変わると再コンパイルが必要

性能比較

評価用のベンチマークスイート

- PolyBenchを利用
  - 30種類の数値計算カーネル
  - さまざまな分野から抽出
    - 線形代数, 物理シミュレーション, 画像処理, 動的計画法, 統計など
  - 各ベンチマークで5つのサイズが異なるデータセット
    - MINI, SMALL, STANDARD, LARGE, EXTRALARGE
- PolyBench-ACC
  - https://github.com/cavazos-lab/PolyBench-ACC
  - 並列化版の実装
  - OpenMP, OpenCL, CUDA, HMPPによる実装
    - ただし、効果的な並列化が望めないカーネルは除く

選択したベンチマーク

- PolyBenchから7個のベンチマークを選択

名前	カテゴリー	内容	データセット
correlation	datamining	NxNの相関行列の計算	N∈{512,1024,2048,4096,8192}
2mm	linear algebra	D= α A x B x C + βDの行列計算(行列サイズNxN)	N∈{256,512,1024,2048,4096}
syr2k	linear algebra	NxNの対称行列に対するランク2kの更新	N∈{256,512,1024,2048,4096}
lu	linear algebra	NxN行列に対するLU分解	N∈{512,1024,2048,4096,8192}
fdtd-2d	stencils	平面NxN平面に対する2次元FDTD解析 (反復ステップ数 500)	N∈{512,1024,2048,4096,8192}
jacobi-2d	stencils	NxN行列に対するヤコビ法 (反復ステップ数 20)	N∈{256,512,1024,2048,4096}
convolution-3d	stencils	NxNxN行列に対する3x3x3カーネルによる畳み込み演算	N∈{64,128,256,384,512}

- CPU向けOpenMP実装をGPUオフローディング向けに修正
  - プラグマtarget teams distributeへの変更およびdataマップを明示

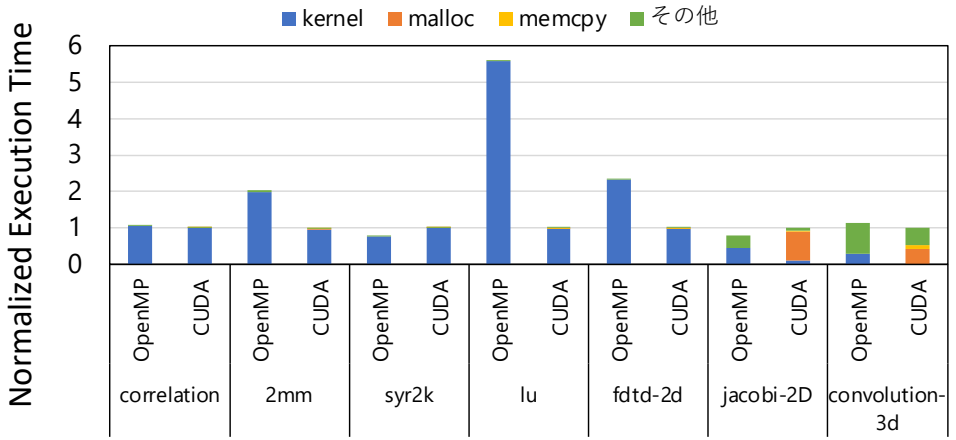
評価環境

- 5種類のベンダ、世代の異なるGPUを用いて評価

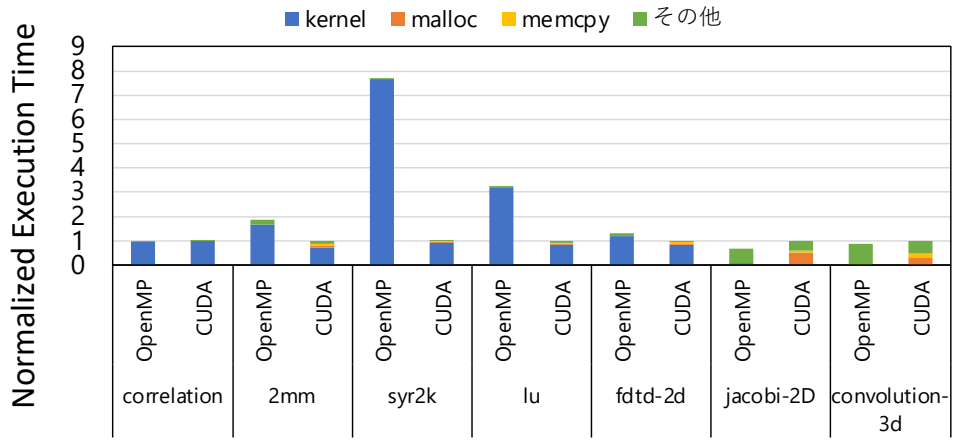
型番	マイクロアーキテクチャ	CUDAコア/SP数	定格クロック (MHz)	VRAM
Quadro P620	Pascal (sm_61)	640	1266	GDDR5 4GB
RTX-2080Ti	Turing (sm_75)	4352	1350	GDDR6 11GB
RTX-A4000	Ampere (sm_86)	6144	735 ≦	GDDR6 16 GB
Radeon RX 5700XT	RDNA (gfx1010)	2560	1605	GDDR6 8GB
Ryzen 5700G APU	第5世代GCN (gfx90c)	512	2000	DDR4 4GB (メインメモリの一部を割り当て)

- 各種ライブラリのバージョン
  - LLVM: 14.0.6 (執筆時最新版)
  - CUDA: 11.2
  - ROCm: 5.0.2

OpenMP vs CUDA (EXTRALARGE)

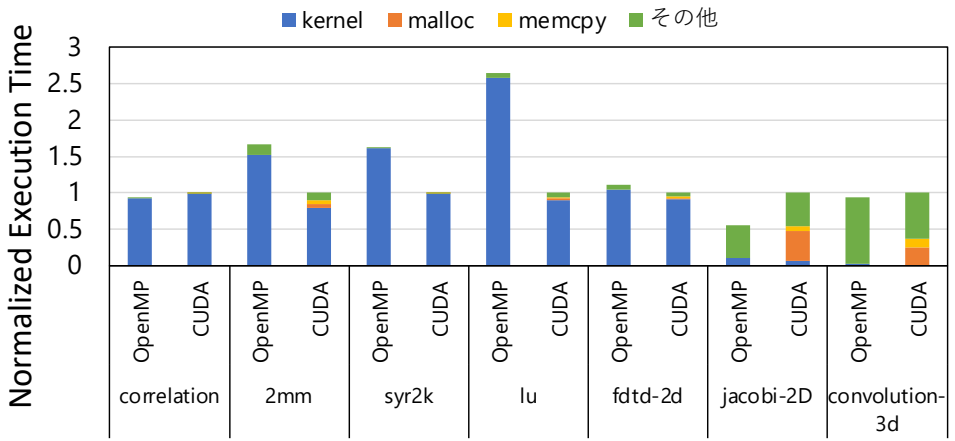


Quadro-P620



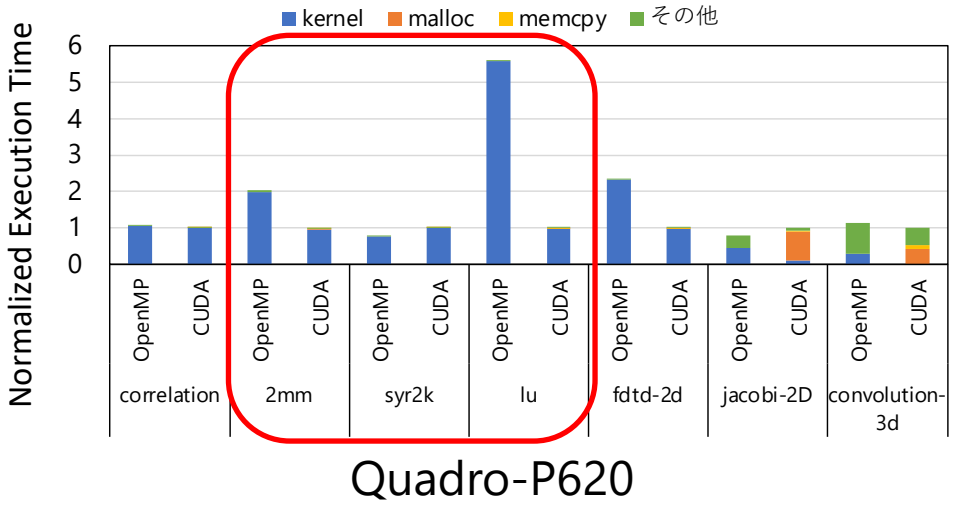
RTX-2080Ti

- 全てO3最適化
- 「linear algebra」カテゴリで差が顕著
  - *lu*ではどのGPUでも数倍遅い
- データサイズに対して計算量が少ないベンチマークでは計算カーネル以外の時間の影響が大きい
  - OpenMPでは総実行時間では短いケースがある
  - Jacobi-2D on A4000では56%の時間

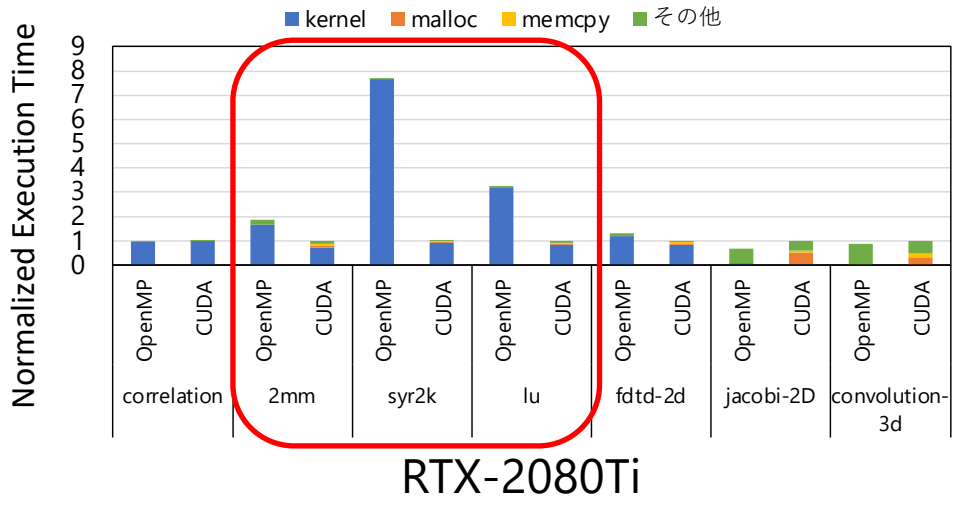


RTX-A4000

OpenMP vs CUDA (EXTRALARGE)

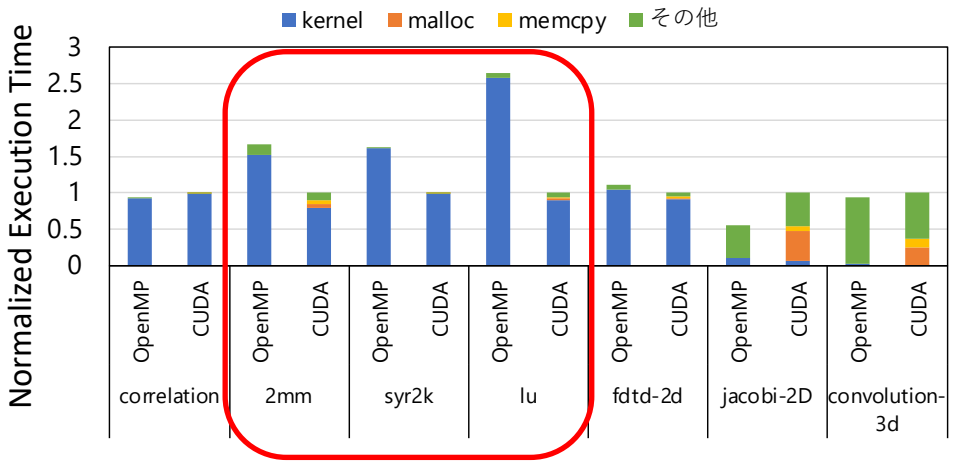


Quadro-P620



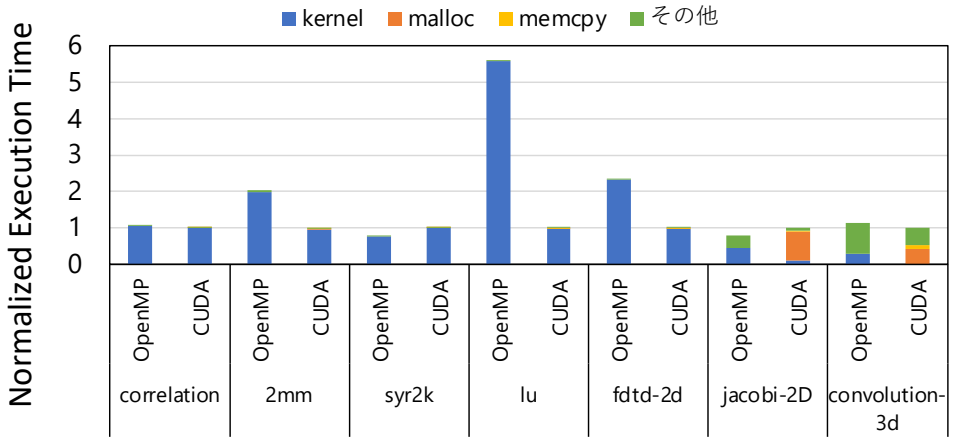
RTX-2080Ti

- 全てO3最適化
- 「linear algebra」カテゴリで差が顕著
  - *lu*ではどのGPUでも数倍遅い
- データサイズに対して計算量が少ないベンチマークでは計算カーネル以外の時間の影響が大きい
  - OpenMPでは総実行時間では短いケースがある
  - Jacobi-2D on A4000では56%の時間

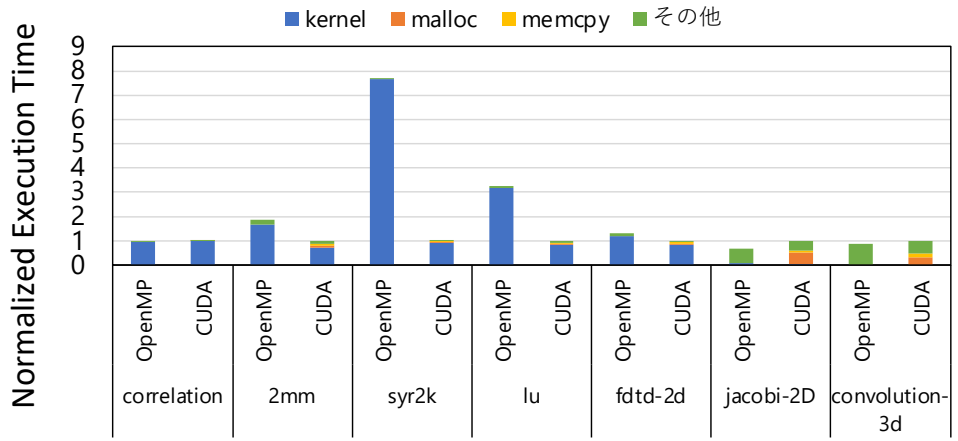


RTX-A4000

OpenMP vs CUDA (EXTRALARGE)

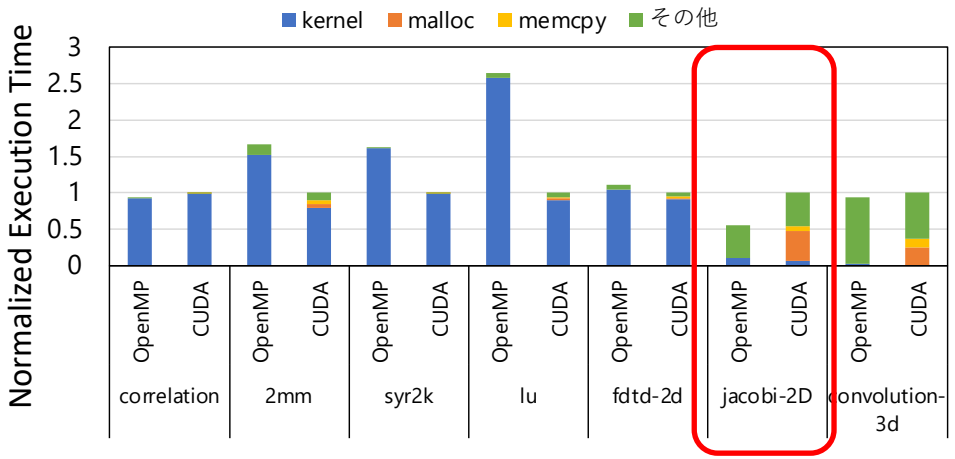


Quadro-P620



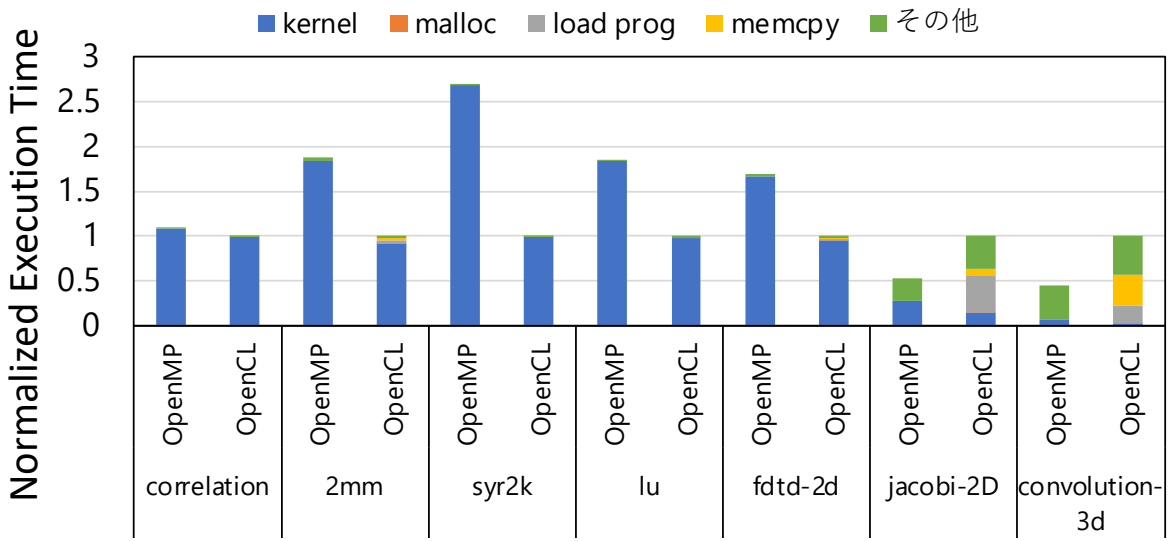
RTX-2080Ti

- 全てO3最適化
- 「linear algebra」カテゴリで差が顕著
  - *lu*ではどのGPUでも数倍遅い
- データサイズに対して計算量が少ないベンチマークでは計算カーネル以外の時間の影響が大きい
  - OpenMPでは総実行時間では短いケースがある
  - Jacobi-2D on A4000では56%の時間

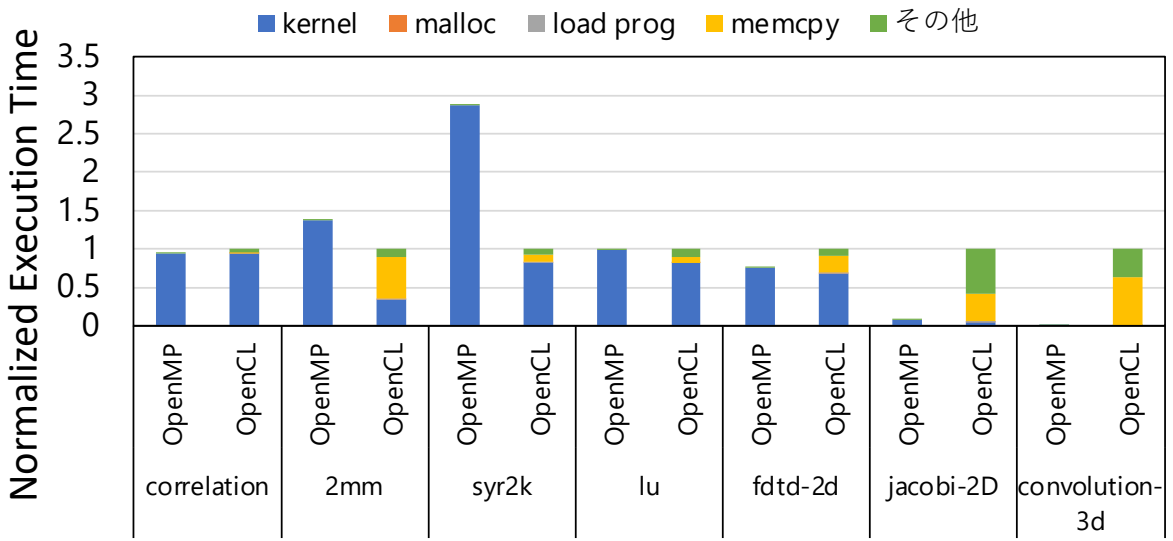


RTX-A4000

OpenMP vs OpenCL (EXTRALARGE)



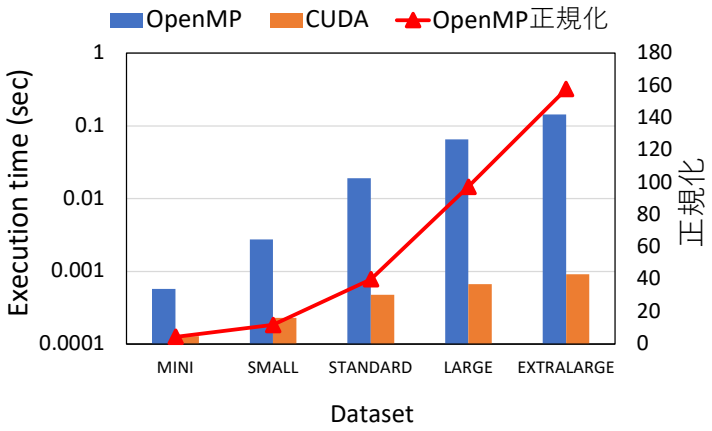
Radeon RX 5700 XT



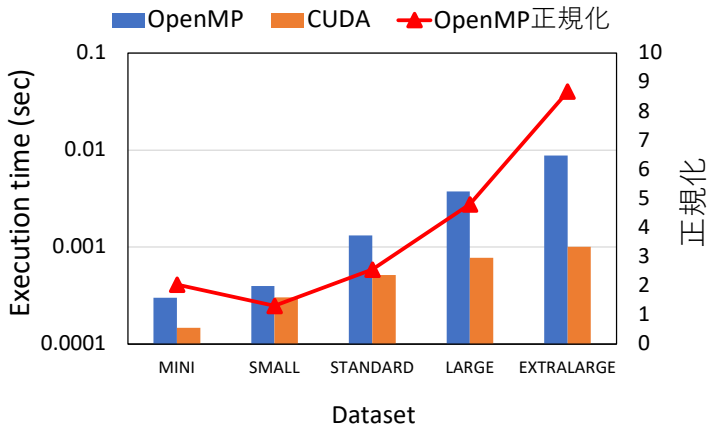
Ryzen 5700G APU

- NVIDIA GPUの場合と同様に「linear algebra」カテゴリで性能差
- Ryzen 5700G APUではデータ転送(clEnqueueWriteBuffer, clEnqueueReadBuffer)が異様に遅いケースが見受けられた

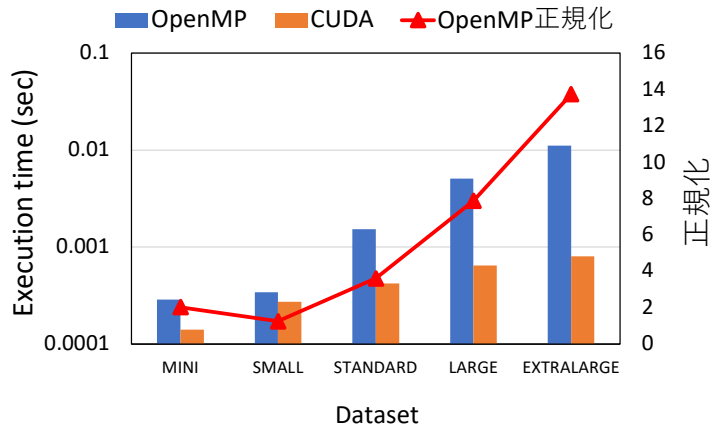
カーネル実行時間 – *convolution-3d*



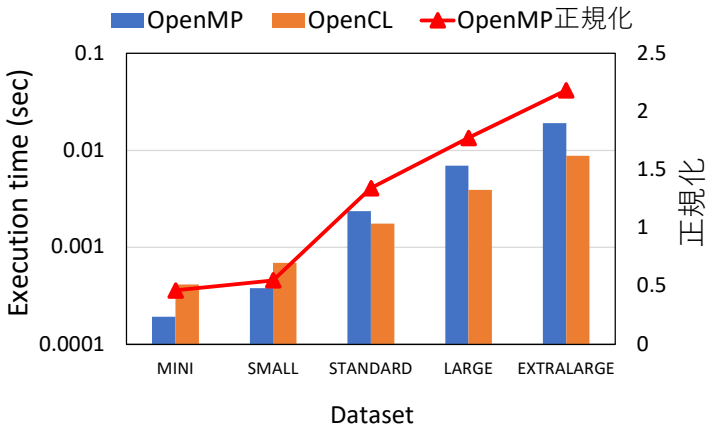
Quadro-P620



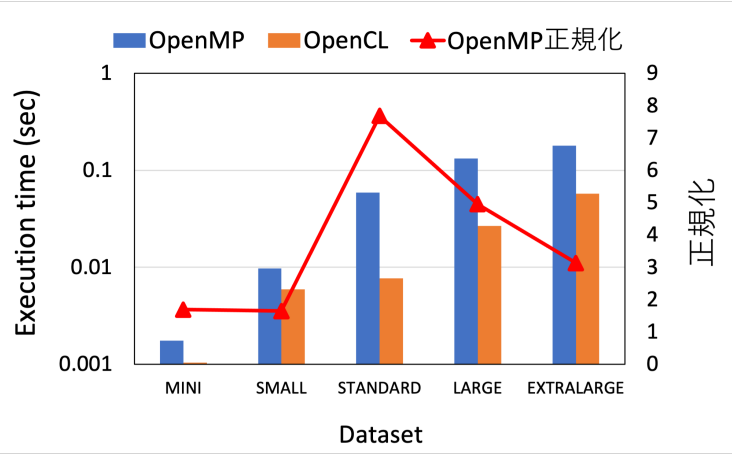
RTX-2080Ti



RTX-A4000



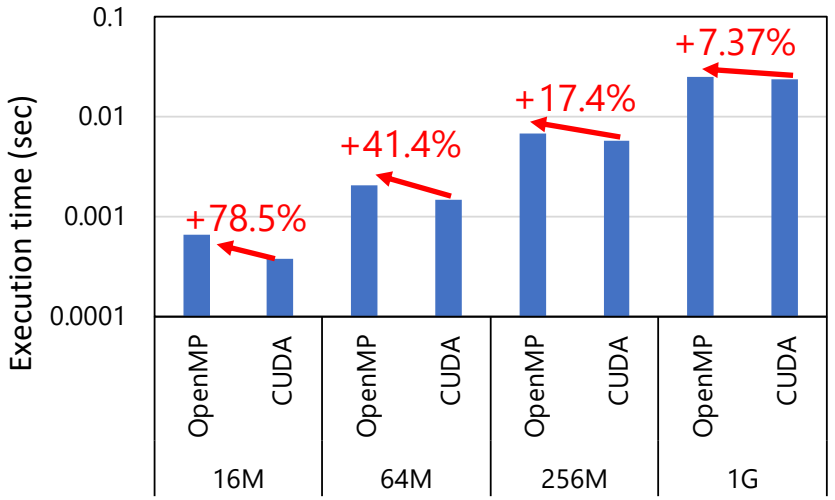
Radeon RX 5700 XT



Ryzen 5700G APU

詳細な検証

- SAXPYのコードを利用
  - GPU: RTX-2080Ti
  - データサイズ: 16M-1G
  - カーネル実行部分の時間を計測
    - cuLaunchKernelの時間を計測できる様にOpenMPラインタイムに細工
    - OpenMPライタイム部の実行時間は50usec程度
    - 計測精度: gettimeofday



計算カーネル実行時間

Nsight Computによるプロファイル (抜粋)

	OpenMP	CUDA
Registers Per Thread	34	16
Static Shared Memory Per Block	1.58 KB/block	0 B
Block Limit Registers	6	16
Block Limit Shared Mem	18	N/A
Achieved Occupancy (%)	87.96	89.89
SASS アセンブリの命令数	272	16

PTXの比較

CUDAが生成するPTX

```
.visible .entry _Z9saxpy_gpufPfS_(
.param .f32 _Z9saxpy_gpufPfS__param_0,
.param .u64 _Z9saxpy_gpufPfS__param_1,
.param .u64 _Z9saxpy_gpufPfS__param_2
)
{
  .reg .f32 %f<5>;
  .reg .b32 %r<5>;
  .reg .b64 %rd<8>;

  ld.param.f32 %f1, [_Z9saxpy_gpufPfS__param_0];
  ld.param.u64 %rd1, [_Z9saxpy_gpufPfS__param_1];
  ld.param.u64 %rd2, [_Z9saxpy_gpufPfS__param_2];
  cvta.to.global.u64 %rd3, %rd2;
  cvta.to.global.u64 %rd4, %rd1;
  mov.u32 %r1, %ctaid.x;
  mov.u32 %r2, %ntid.x;
  mov.u32 %r3, %tid.x;
  mad.lo.s32 %r4, %r1, %r2, %r3;
  mul.wide.u32 %rd5, %r4, 4;
  add.s64 %rd6, %rd4, %rd5;
  ld.global.f32 %f2, [%rd6];
  add.s64 %rd7, %rd3, %rd5;
  ld.global.f32 %f3, [%rd7];
  fma.rn.f32 %f4, %f2, %f1, %f3;
  st.global.f32 [%rd7], %f4;
  ret;
}
```

OpenMPが生成するPTX (序盤のみ)

```
.weak .entry __omp_offloading_35_2aa284e__Z9saxpy_gpulfPfS__l59(
.param .u64 __omp_offloading_35_2aa284e__Z9saxpy_gpulfPfS__l59_param_0,
.param .u64 __omp_offloading_35_2aa284e__Z9saxpy_gpulfPfS__l59_param_1,
.param .u64 __omp_offloading_35_2aa284e__Z9saxpy_gpulfPfS__l59_param_2,
.param .u64 __omp_offloading_35_2aa284e__Z9saxpy_gpulfPfS__l59_param_3
)
{
  .reg .pred %p<28>;
  .reg .b16 %rs<6>;
  .reg .b32 %r<43>;
  .reg .f32 %f<8>;
  .reg .b64 %rd<106>;
  // demoted variable
  .shared .align 16 .b8 _ZN12_GLOBAL__N_122SharedMemorySmartStackE[1536];
  // demoted variable
  .shared .align 8 .b8 _ZN12_GLOBAL__N_19TeamStateE[40];
  ld.param.u64 %rd47, [__omp_offloading_35_2aa284e__Z9saxpy_gpulfPfS__l59_param_0];
  mov.u32 %r1, %tid.x;
  setp.ne.s32 %p1, %r1, 0;
  cvt.u64.u32 %rd51, %r1;
  mov.u64 %rd52, _ZN12_GLOBAL__N_122SharedMemorySmartStackE;
  add.s64 %rd53, %rd52, %rd51;
  mov.u16 %rs2, 0;
  st.shared.u8 [%rd53+512], %rs2;
  mov.u32 %r39, 0;
  mov.u32 %r40, 1;
  @%p1 bra LBB0_2;
  mov.u32 %r10, %ntid.x;
  st.shared.v2.u32 [_ZN12_GLOBAL__N_19TeamStateE], {%r10, %r39};
  st.shared.v2.u32 [_ZN12_GLOBAL__N_19TeamStateE+8], {%r39, %r40};
  st.shared.v2.u32 [_ZN12_GLOBAL__N_19TeamStateE+16], {%r40, %r40};
  st.shared.u32 [_ZN12_GLOBAL__N_19TeamStateE+24], %r40;
LBB0_2:
  bar.sync 0;
  setp.lt.s64 %p2, %rd47, 1;
   @%p2 bra LBB0_31;
```

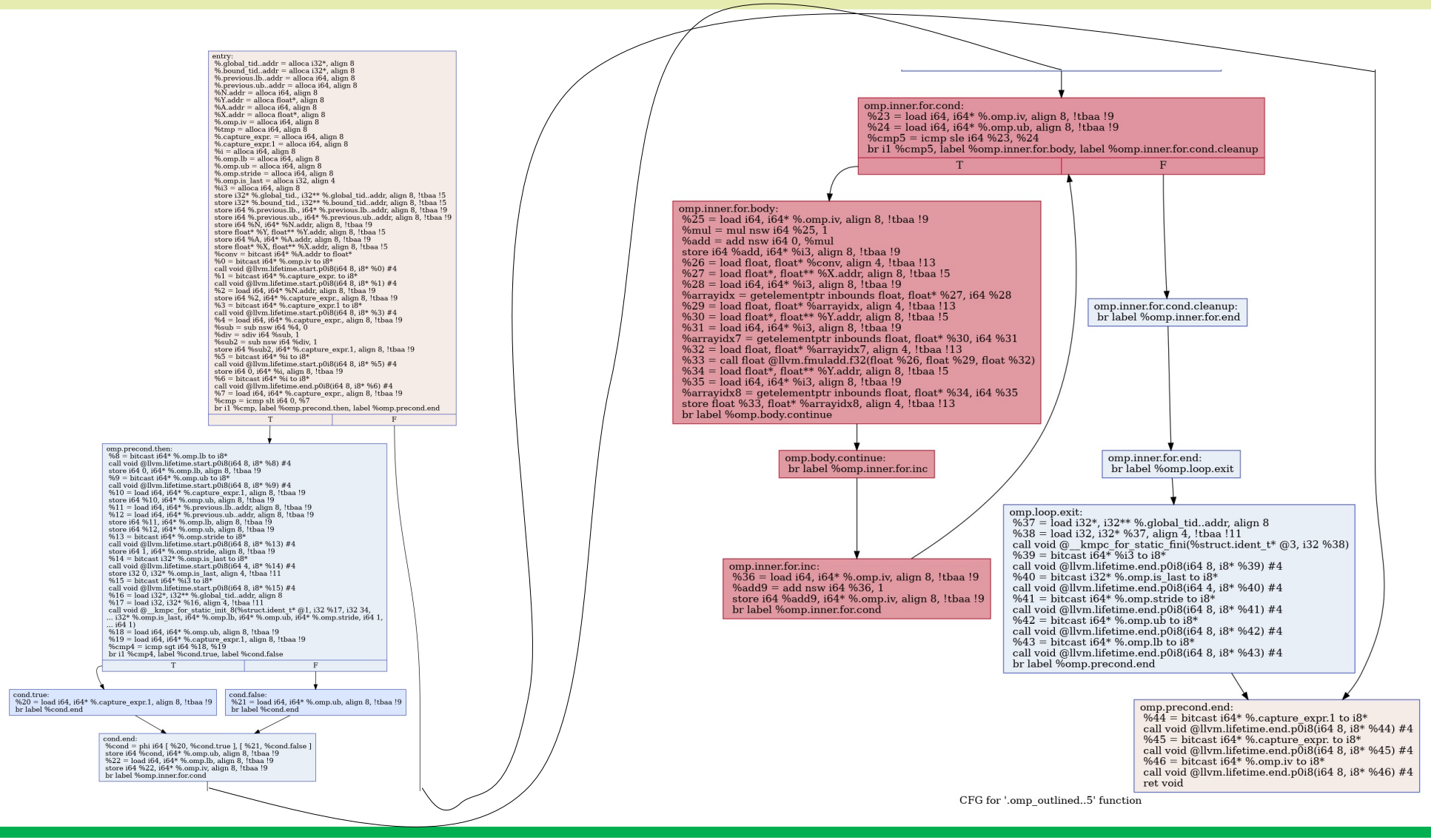
計算カーネルのLLVM IRを覗く

```
; Function Attrs: alwaysinline norecurse nounwind uwtable
define internal void @__omp_offloading_35_2aa284e__Z9saxpy_gpulfPfS__l59(i64 noundef %N, float* noundef %Y, i64 noundef %A, float*
noundef %X) #9 {
entry:
  %N.addr = alloca i64, align 8
  %Y.addr = alloca float*, align 8
  %A.addr = alloca i64, align 8
  %X.addr = alloca float*, align 8
  %N.casted = alloca i64, align 8
  %A.casted = alloca i64, align 8
  %0 = call i32 @__kmpc_global_thread_num(%struct.ident_t* @2)
  store i64 %N, i64* %N.addr, align 8, !tbaa !25
  store float* %Y, float** %Y.addr, align 8, !tbaa !9
  store i64 %A, i64* %A.addr, align 8, !tbaa !25
  store float* %X, float** %X.addr, align 8, !tbaa !9
  %conv = bitcast i64* %A.addr to float*
  call void @__kmpc_push_num_teams(%struct.ident_t* @2, i32 %0, i32 0, i32 256)
  %1 = load i64, i64* %N.addr, align 8, !tbaa !25
  store i64 %1, i64* %N.casted, align 8, !tbaa !25
  %2 = load i64, i64* %N.casted, align 8, !tbaa !25
  %3 = load float*, float** %Y.addr, align 8, !tbaa !9
  %4 = load float, float* %conv, align 4, !tbaa !23
  %conv1 = bitcast i64* %A.casted to float*
  store float %4, float* %conv1, align 4, !tbaa !23
  %5 = load i64, i64* %A.casted, align 8, !tbaa !25
  %6 = load float*, float** %X.addr, align 8, !tbaa !9
  call void (%struct.ident_t*, i32, void (i32*, i32*, ...)*, ...) @__kmpc_fork_teams(%struct.ident_t* @2, i32 4, void (i32*,
i32*, ...)* bitcast (void (i32*, i32*, i64, float*, i64, float*)* @.omp_outlined..4 to void (i32*, i32*, ...)*), i64 %2, float* %3,
i64 %5, float* %6)
  ret void
}
```

計算カーネルのLLVM IRを覗く

```
; Function Attrs: alwaysinline norecurse nounwind uwtable
define internal void @.omp_outlined..5(i32* noalias noundef %.global_tid., i32* noalias noundef %.bound_tid., i64
noundef %.previous.lb., i64 noundef %.previous.ub., i64 noundef %N, float* noundef %Y, i64 noundef %A, float* noundef %X) #3 {
entry:
  %.global_tid..addr = alloca i32*, align 8
  %.bound_tid..addr = alloca i32*, align 8
  %.previous.lb..addr = alloca i64, align 8
  %.previous.ub..addr = alloca i64, align 8
  %N.addr = alloca i64, align 8
  %Y.addr = alloca float*, align 8
  %A.addr = alloca i64, align 8
  %X.addr = alloca float*, align 8
  %.omp.iv = alloca i64, align 8
  %tmp = alloca i64, align 8
  %.capture_expr. = alloca i64, align 8
  %.capture_expr.1 = alloca i64, align 8
  %i = alloca i64, align 8
  %.omp.lb = alloca i64, align 8
  %.omp.ub = alloca i64, align 8
  %.omp.stride = alloca i64, align 8
  %.omp.is_last = alloca i32, align 4
  %i3 = alloca i64, align 8
  store i32* %.global_tid., i32** %.global_tid..addr, align 8, !tbaa !9
  store i32* %.bound_tid., i32** %.bound_tid..addr, align 8, !tbaa !9
  store i64 %.previous.lb., i64* %.previous.lb..addr, align 8, !tbaa !25
  store i64 %.previous.ub., i64* %.previous.ub..addr, align 8, !tbaa !25
  store i64 %N, i64* %N.addr, align 8, !tbaa !25
  store float* %Y, float** %Y.addr, align 8, !tbaa !9
  store i64 %A, i64* %A.addr, align 8, !tbaa !25
  store float* %X, float** %X.addr, align 8, !tbaa !9
```

各teamが担当するループのIR



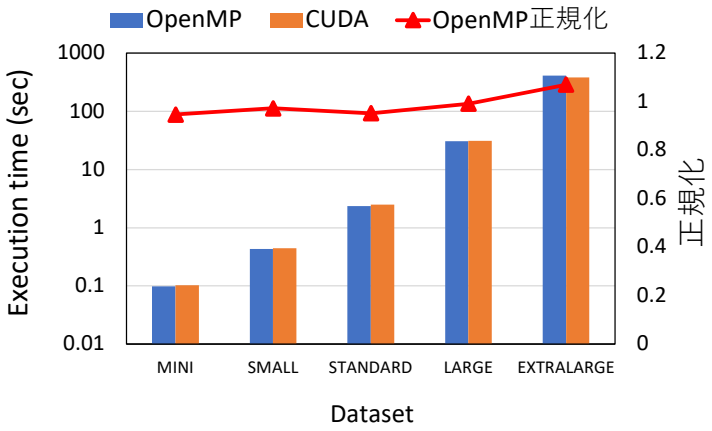
CFG for '.omp_outlined..5' function

まとめ

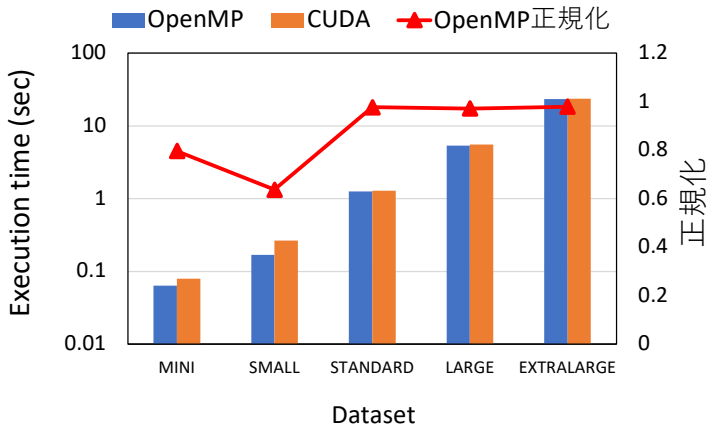
- OpenMP version4.0からアクセラレータオフローディングがサポート
- LLVMにおいてはNVIDIA GPUやAMD GPU向けの実装が公開
- PolyBenchを用いてOpenMP実装およびCUDA, OpenCL実装を比較
- 「linear algebra」カテゴリのベンチマークではOpenMPが数倍遅い
- 一方で、計算カーネルの実行時間が支配的ではないベンチマークでは総実行時間が短いものもあり
- OpenMPの仕組み上、メモリを介してパラメータを渡すため、最適化が困難であると考えられる
  - NVIDIA GPUでは共有メモリを利用
  - より詳細な解析にはGPUアセンブリ生成のバックエンドを理解する必要がある

EXTRA SLIDES

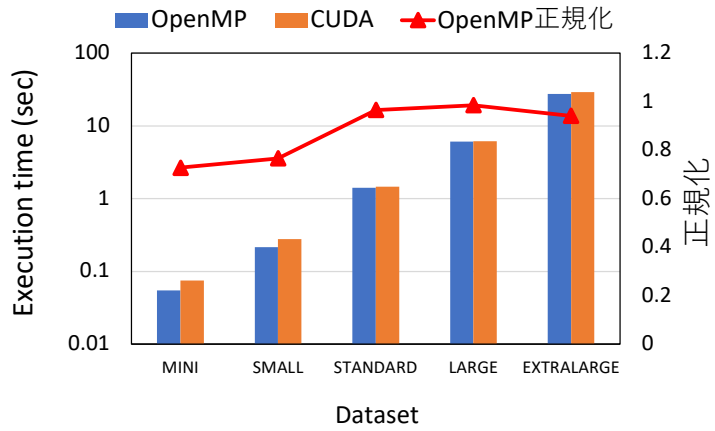
カーネル実行時間 - *correlation*



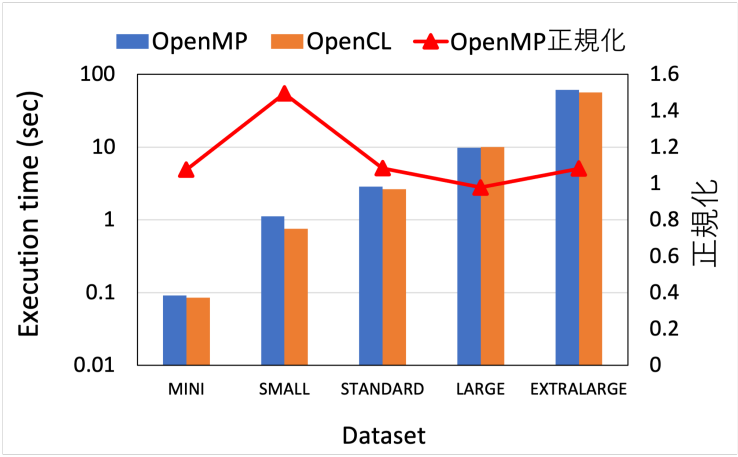
Quadro-P620



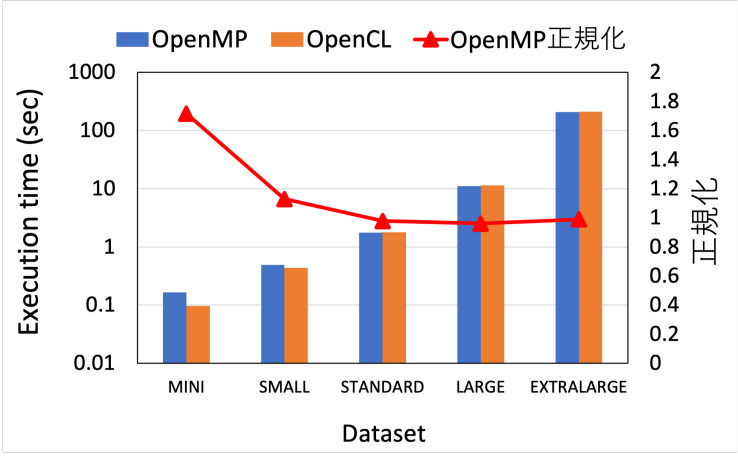
RTX-2080Ti



RTX-A4000

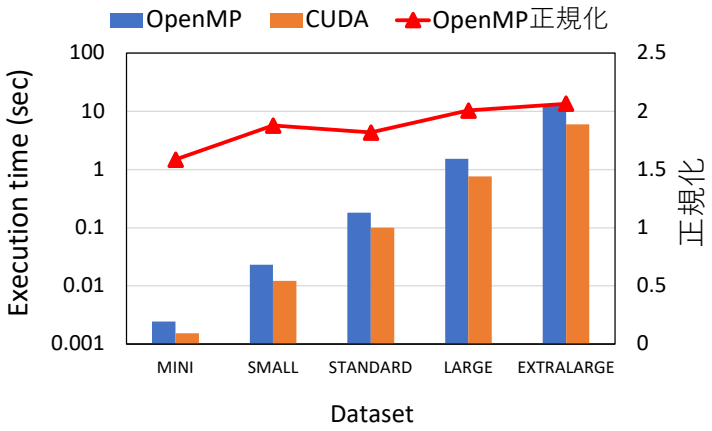


Radeon RX 5700 XT

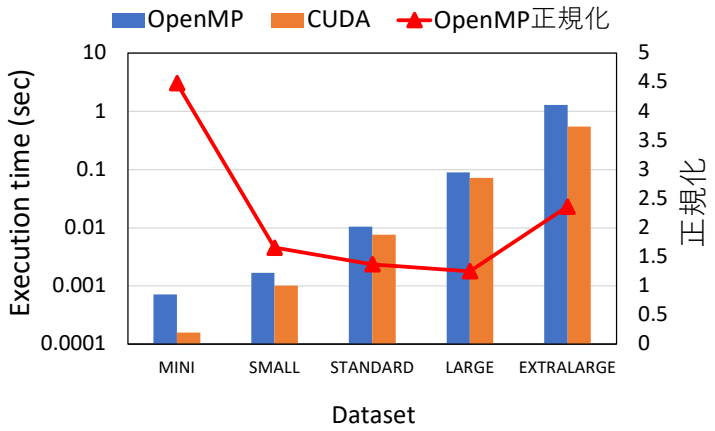


Ryzen 5700G APU

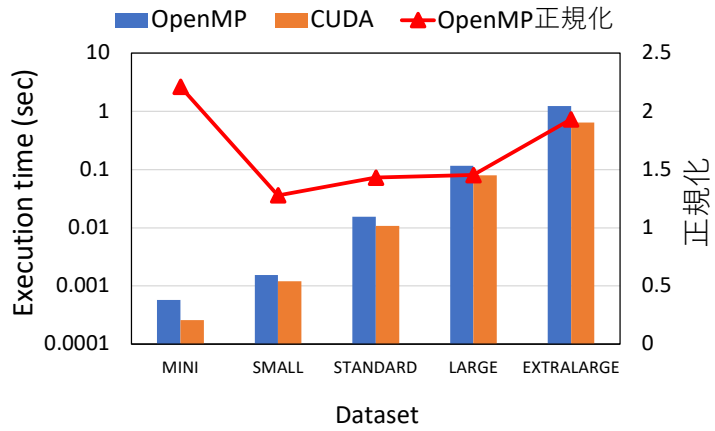
カーネル実行時間 - *2mm*



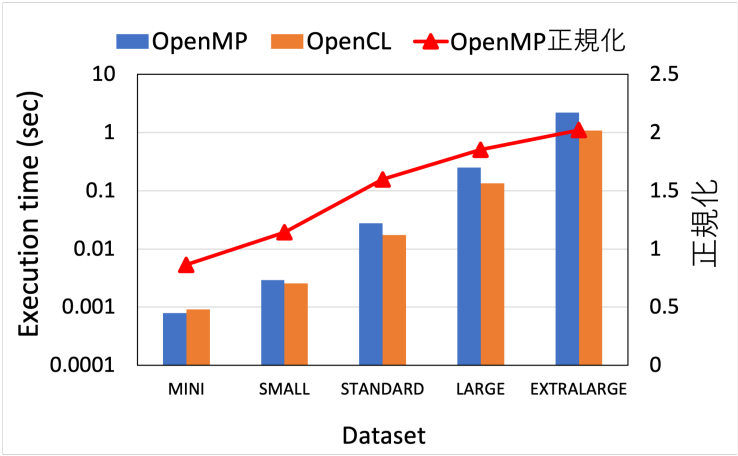
Quadro-P620



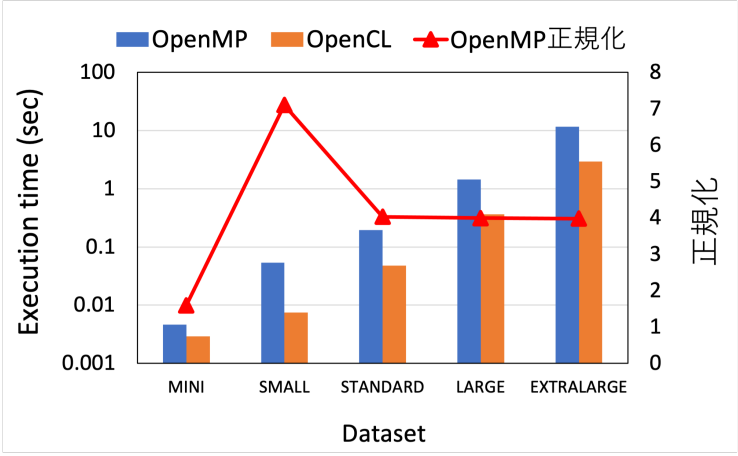
RTX-2080Ti



RTX-A4000

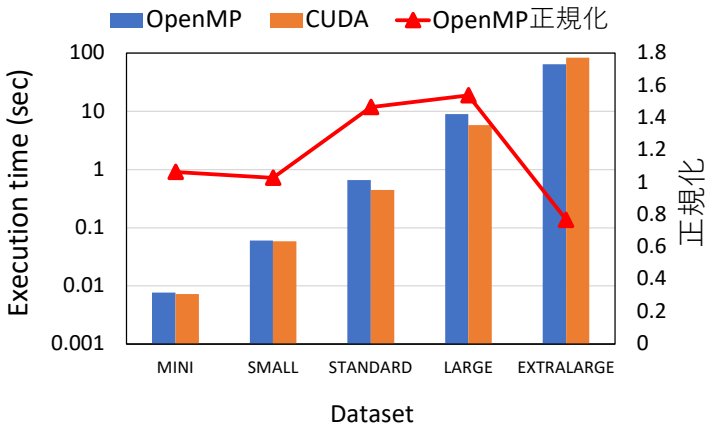


Radeon RX 5700 XT

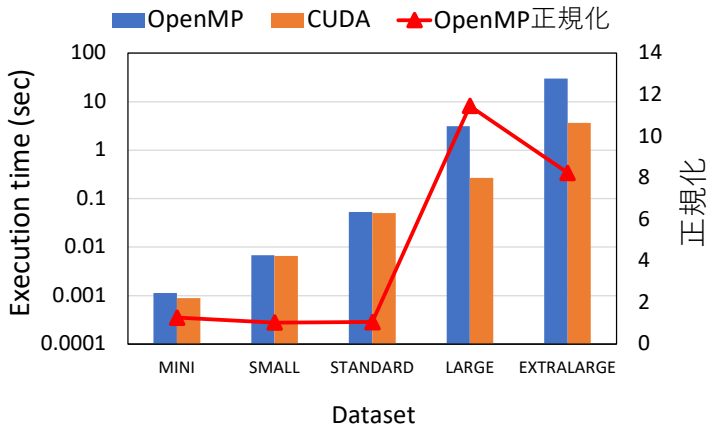


Ryzen 5700G APU

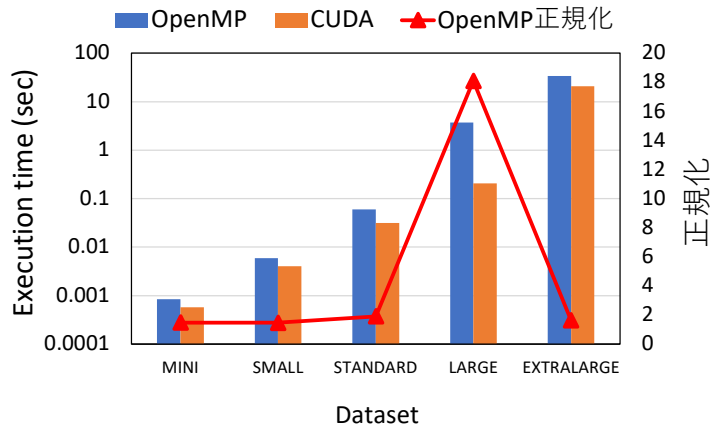
カーネル実行時間 - *syr2k*



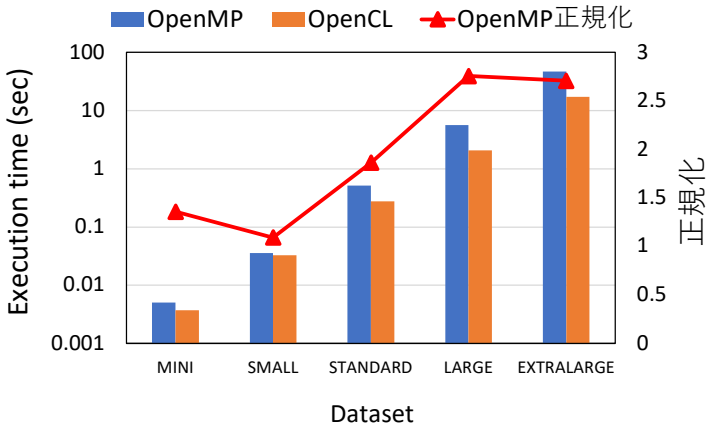
Quadro-P620



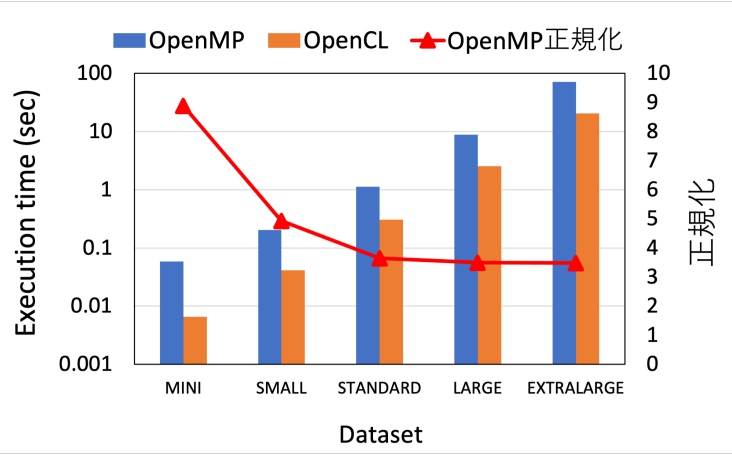
RTX-2080Ti



RTX-A4000

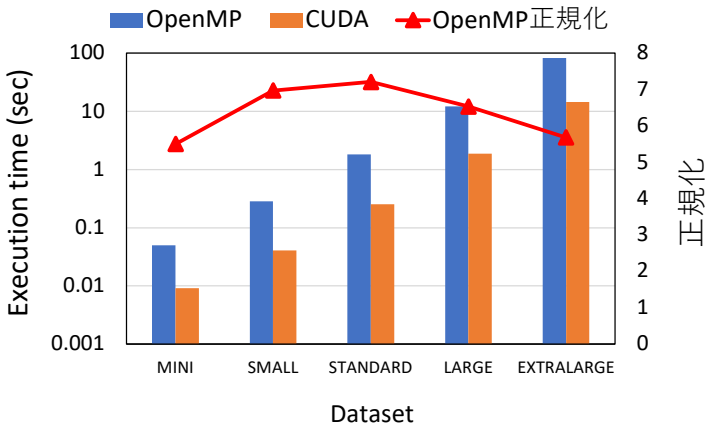


Radeon RX 5700 XT

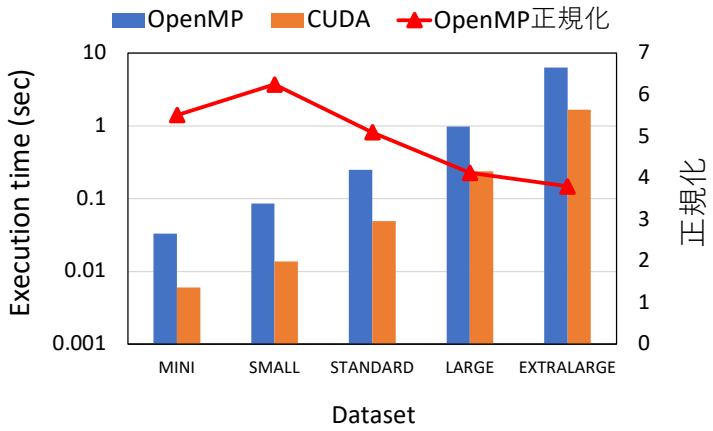


Ryzen 5700G APU

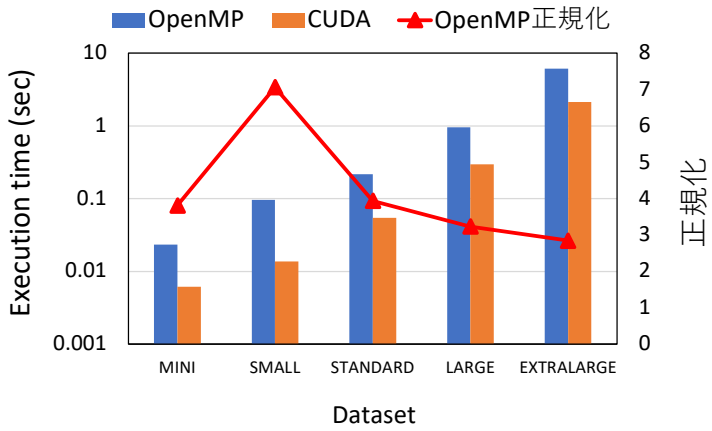
カーネル実行時間 - *lu*



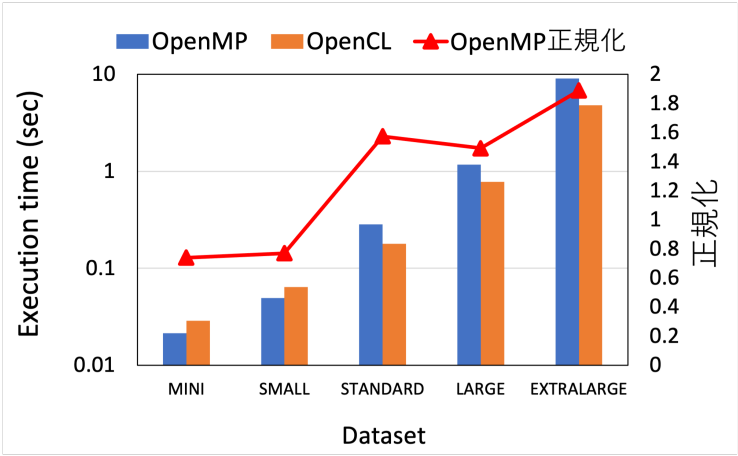
Quadro-P620



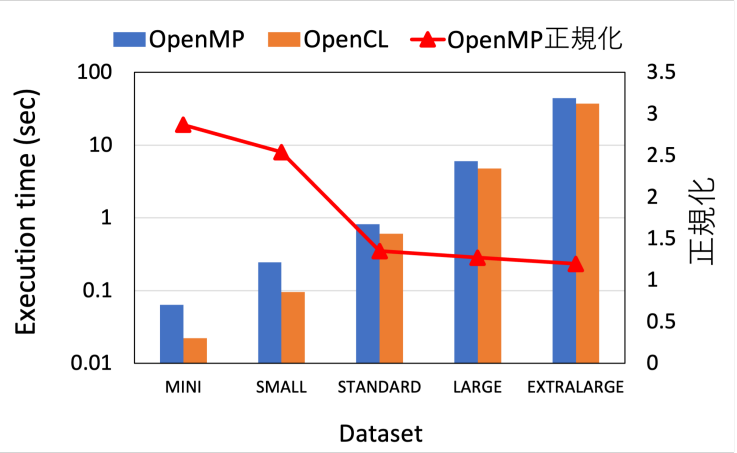
RTX-2080Ti



RTX-A4000

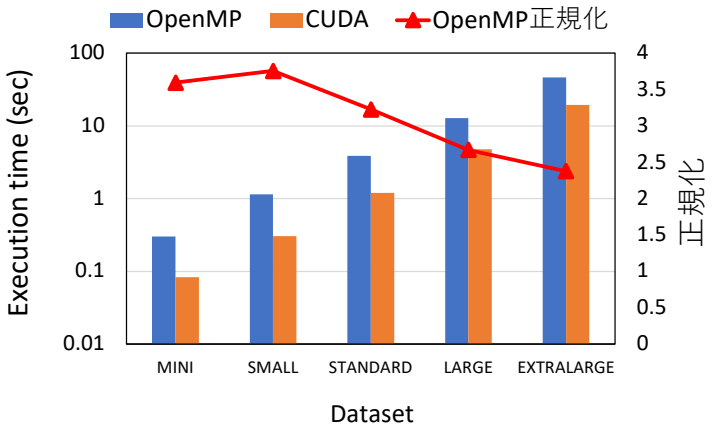


Radeon RX 5700 XT

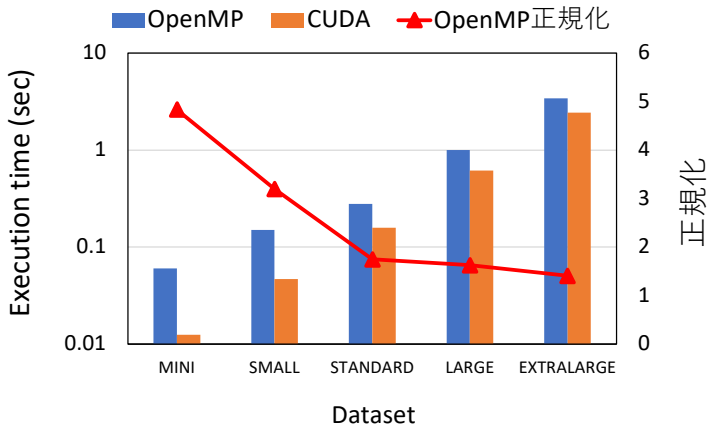


Ryzen 5700G APU

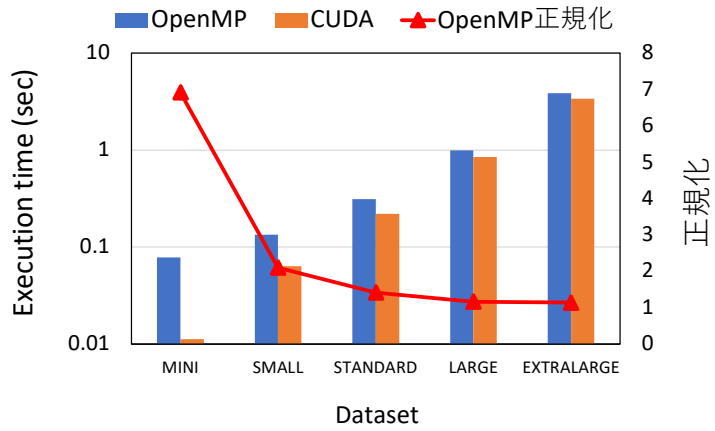
カーネル実行時間 – *fdtd-2d*



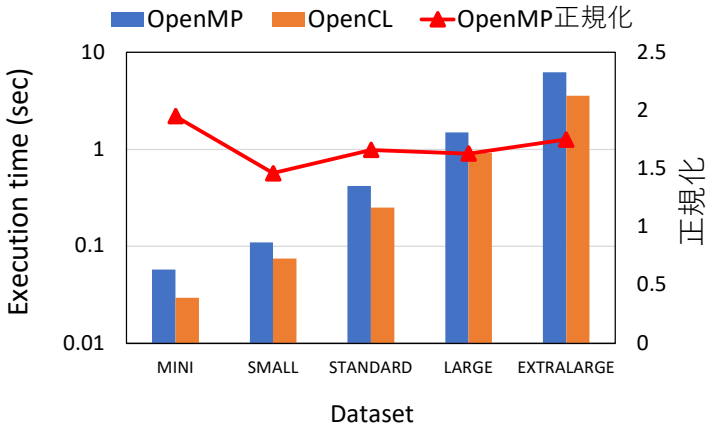
Quadro-P620



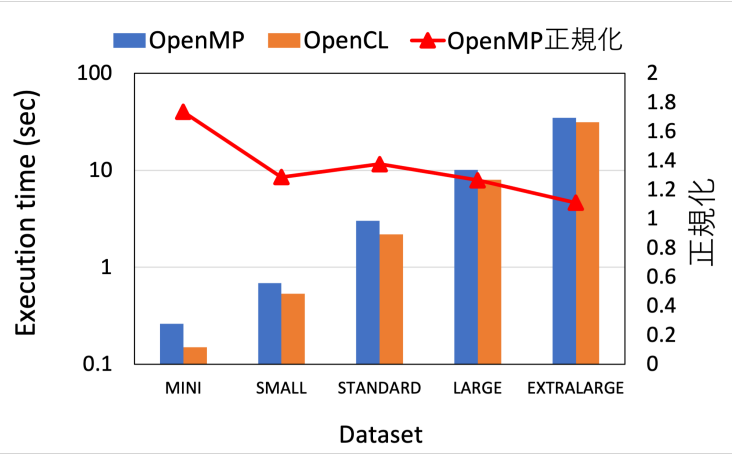
RTX-2080Ti



RTX-A4000

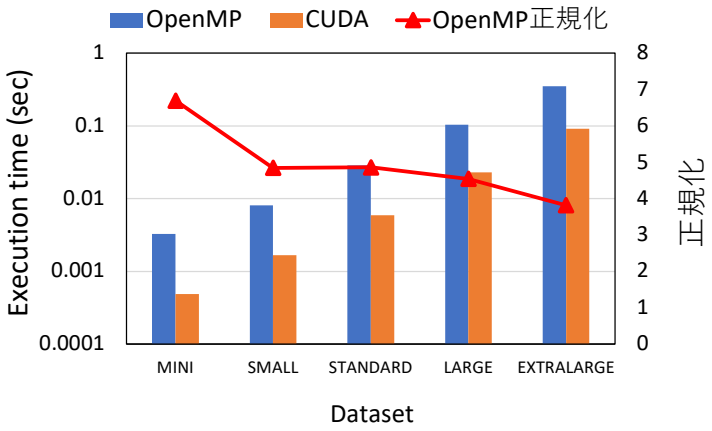


Radeon RX 5700 XT

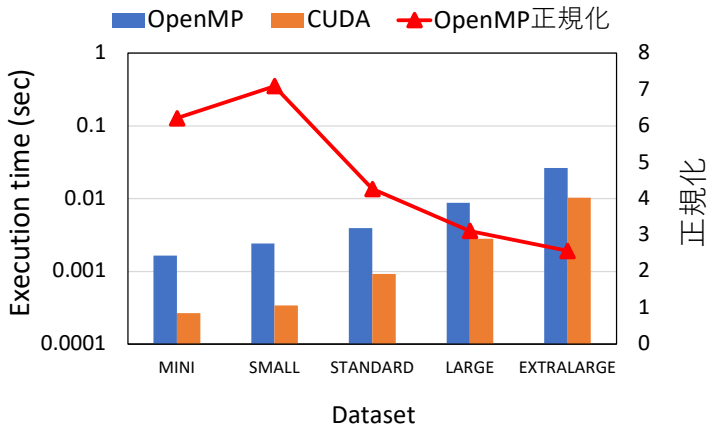


Ryzen 5700G APU

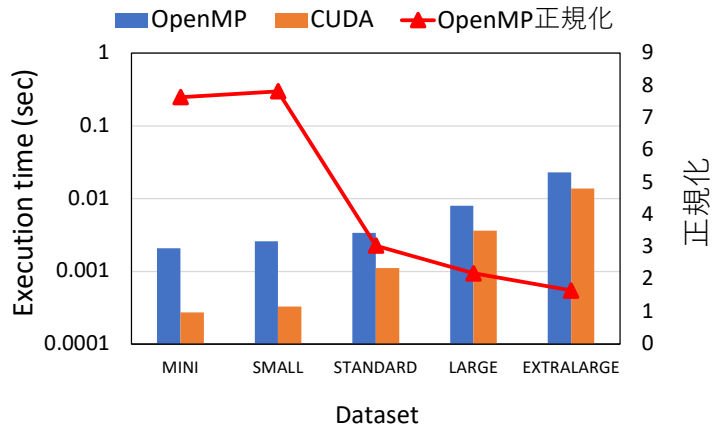
カーネル実行時間 – *jacobi-2d*



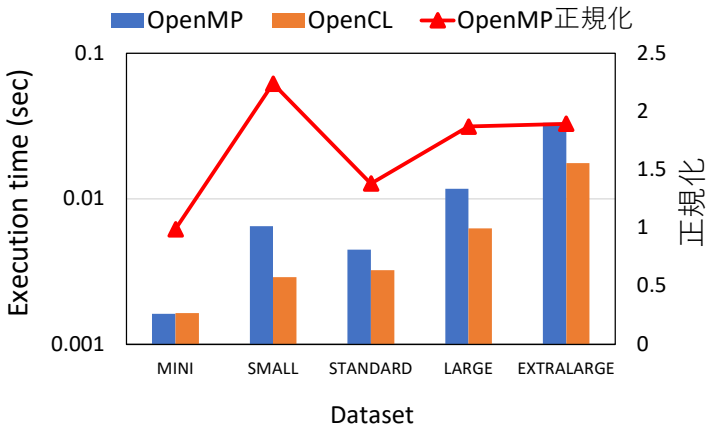
Quadro-P620



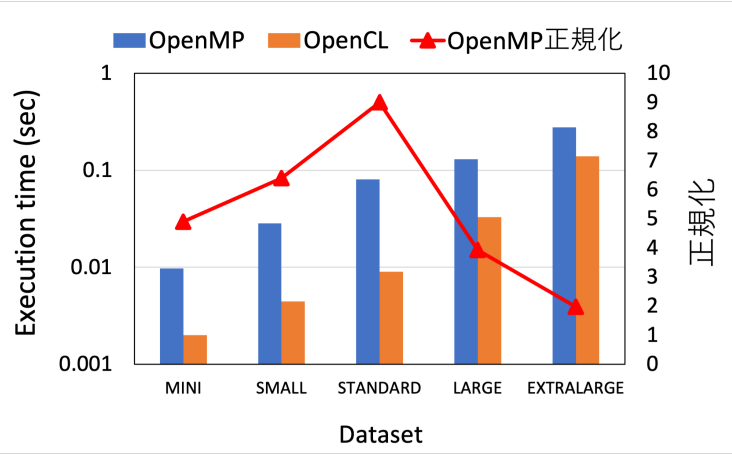
RTX-2080Ti



RTX-A4000



Radeon RX 5700 XT



Ryzen 5700G APU