GPUが拓く地理情報分析の新たな地平
～GPU版PostGISの実装と検証～

HeteroDB,Inc
Chief Architect & CEO
海外 浩平 <kaigai@heterodb.com>

自己紹介／HeteroDB社について

ヘテロジニアスコンピューティング技術を
データベース領域に適用し、
誰もが使いやすく、安価で高速なデータ解析基盤を提供する。

会社概要

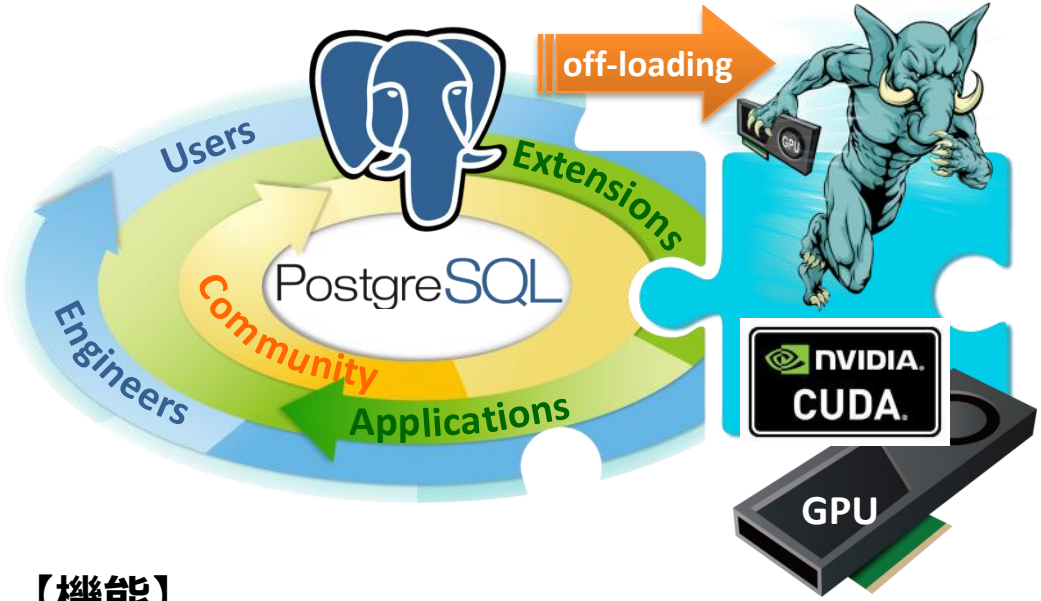
- 商号　ヘテロDB株式会社
- 創業　2017年7月4日
- 拠点　品川区北品川5-5-15
  大崎ブライトコア4F
- 事業内容　高速データベース製品の販売
  GPU＆DB領域の技術コンサルティング

代表者プロフィール

- 海外　浩平（KaiGai Kohei）
- OSS開発者コミュニティにおいて、PostgreSQLやLinux kernelの開発に10年以上従事。主にセキュリティ・FDW等の分野でアップストリームへの貢献。
- IPA未踏ソフト事業において“天才プログラマー”認定 (2006)
- GPU Technology Conference Japan 2017でInception Awardを受賞

PG-Stromとは？

PG-Strom: GPUとNVME/PMEMの能力を最大限に引き出し、
テラバイト級のデータを高速処理するPostgreSQL向け拡張モジュール



for IoT/Big-Data

- SSD-to-GPU Direct SQL
- Columnar Store (Arrow_Fdw)
- GPU Memory Store (Gstore_Fdw)
- Asymmetric Partition-wise JOIN/GROUP BY
- BRIN-Index Support
- NVME-over-Fabric Support

for ML/Analytics

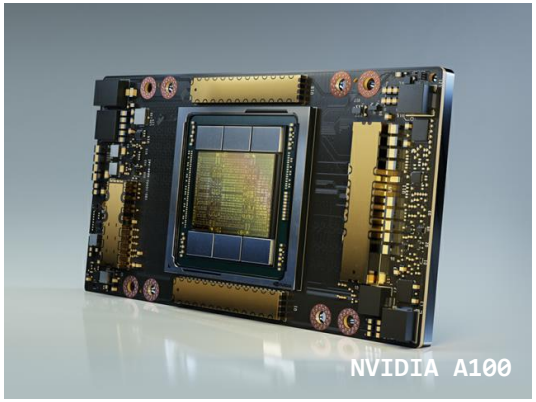
- Inter-process Data Frame for Python scripts
- Procedural Language for GPU native code (w/ cuPy)
- PostGIS Support

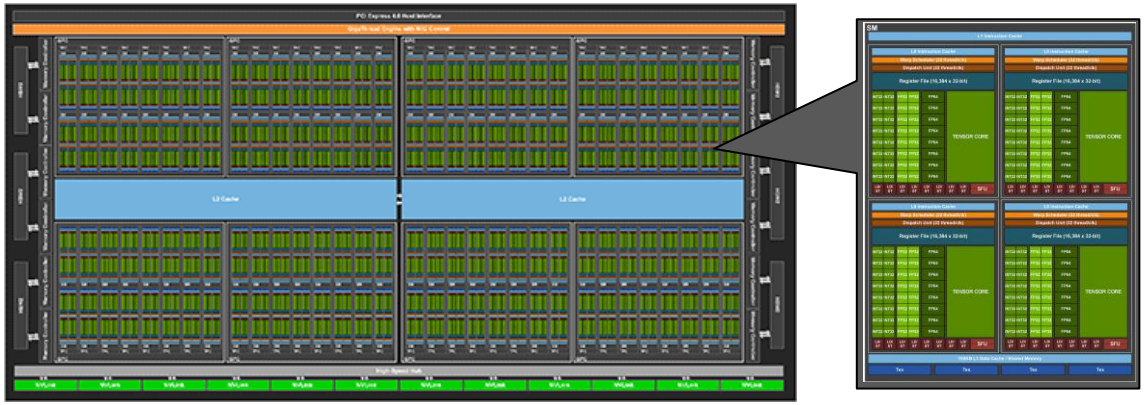
【機能】

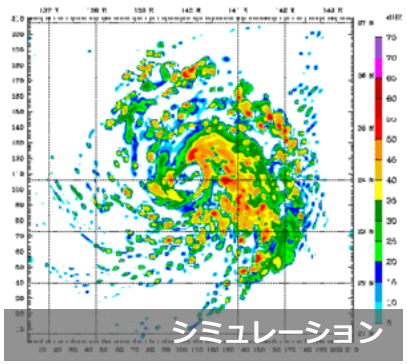
- 集計／解析ワークロードの透過的なGPU高速化
- SSD-to-GPU Direct SQLによるPCIeバスレベルの最適化
- Apache Arrow対応によるデータ交換、インポート時間をほぼゼロに
- GPUメモリにデータを常駐。OLTPワークロードとの共存 NEW
- PostGIS関数のサポート（一部）。位置情報分析を高速化 NEW

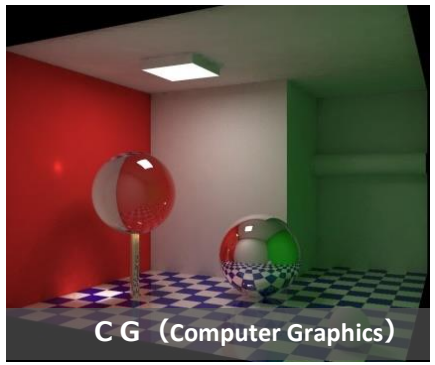
GPUとはどんなプロセッサなのか？

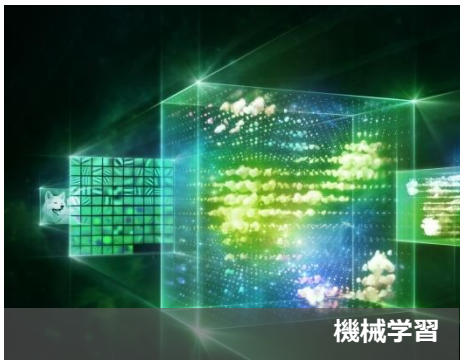
主にHPC分野で実績があり、機械学習用途で爆発的に普及





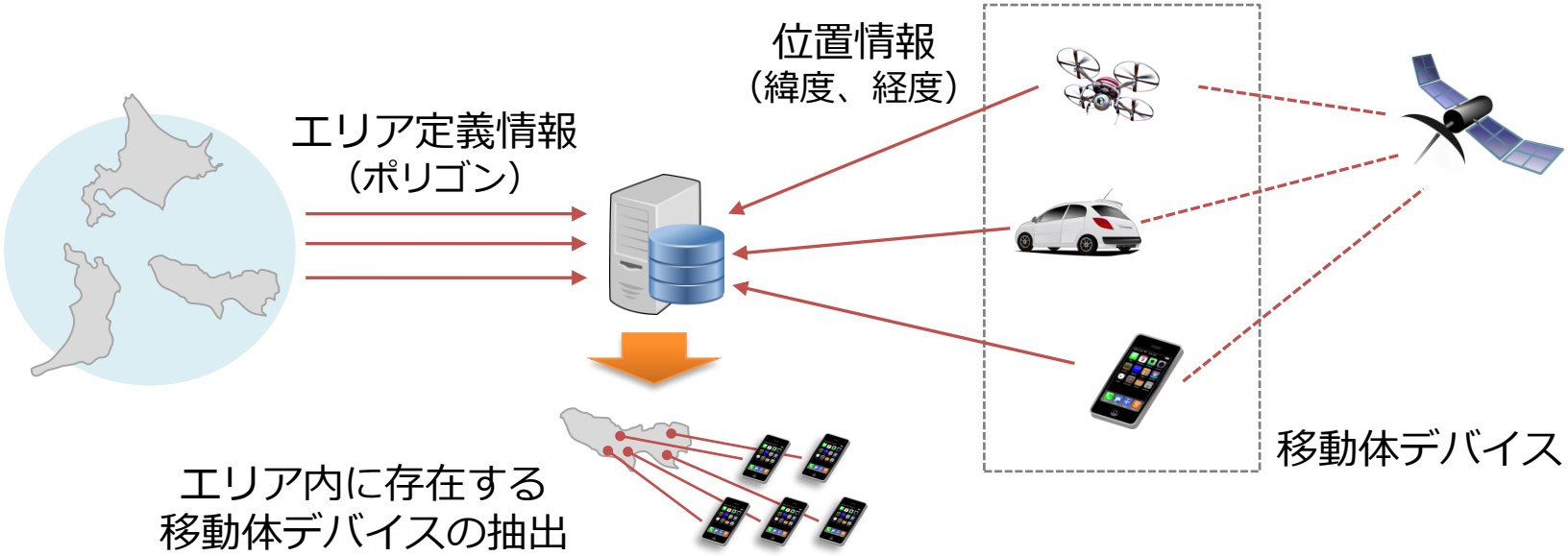






数千コアの並列処理ユニット、TB/sに達する広帯域メモリを
ワンチップに実装した半導体デバイス
➔ “同じ計算を大量のデータに並列実行” を最も得意とする

ターゲット：移動体デバイスのデータ検索



移動体デバイス

- 高頻度で位置（緯度、経度）の更新がかかる。
- （デバイスID、タイムスタンプ、位置、その他の属性）というデータ構造が多い

エリア定義情報

- 比較的件数は少なく、ほぼ静的なデータだが、エリア定義情報（多角形）が非常に複雑な構造を持ち、当たり判定が“重い”

用途

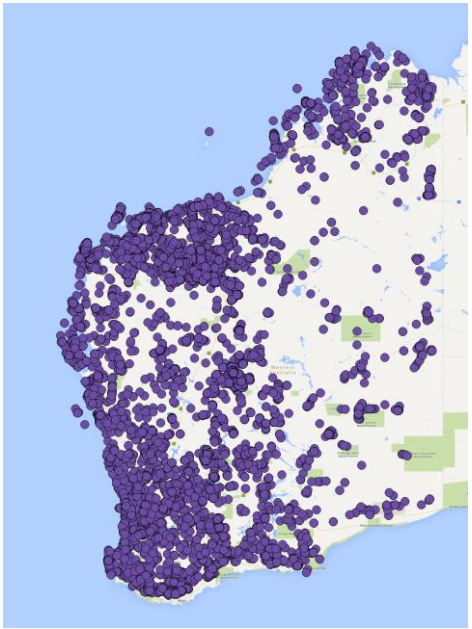
- 商圏分析、物流分析、広告配信、アラート通知、etc...

GPU版PostGISについて

PostGISって？（1/2）

概要

- 点（Point）、線（LineString）、多角形（Polygon）等のジオメトリ要素、およびそれらの間の演算を PostgreSQL で実行するための拡張モジュール
- 演算の例　…　包含（Contains）、交差（Crosses）、接合（Touch）など
- GiSTインデックスに対応して高速な検索を実現
- 2005年に PostGIS 1.0 がリリース。以降、15年にわたって継続的開発。



© GAIA RESOURCES

PostGIS関数・演算子の例

- st_distance(geometry g1, geometry g2)
  - ✓ ジオメトリ間の距離を計算する
- st_contains(geometry g1, geometry g2)
  - ✓ ジオメトリ g2 が g1 に包含されていれば真
- st_crosses(geometry g1, geometry g2)
  - ✓ ジオメトリ g2 が g1 を横切っていれば真

…など

PostGISって？（2/2）

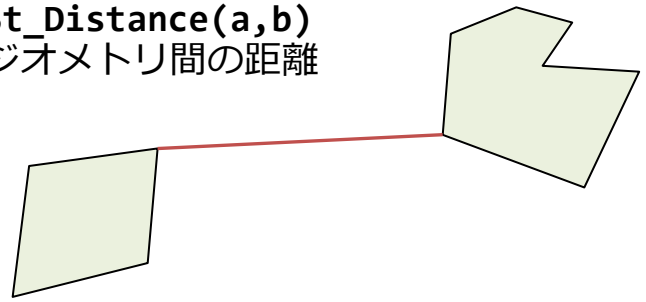
Geometry primitives (2D)

Type	凡例	
Point		POINT (30 10)
LineString		LINESTRING (30 10, 10 30, 40 40)
Polygon		POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))
		POLYGON ((35 10, 45 45, 15 40, 10 20, 35 10), (20 30, 35 35, 30 20, 20 30))

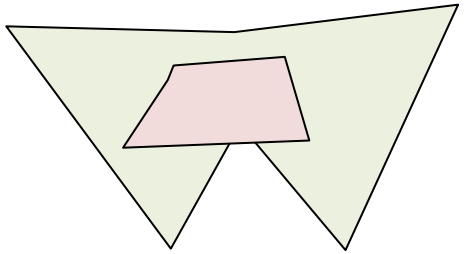
Multipart geometries (2D)

Type	凡例	
MultiPoint		MULTIPOINT ((10 40), (40 30), (20 20), (30 10))
		MULTIPOINT (10 40, 40 30, 20 20, 30 10)
MultiLineString		MULTILINESTRING ((10 10, 20 20, 10 40), (40 40, 30 30, 40 20, 30 10))
MultiPolygon		MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10, 15 5)))
		MULTIPOLYGON (((40 40, 20 45, 45 30, 40 40)), ((20 35, 10 30, 10 10, 30 5, 45 20, 20 35), (30 20, 20 15, 20 25, 30 20)))

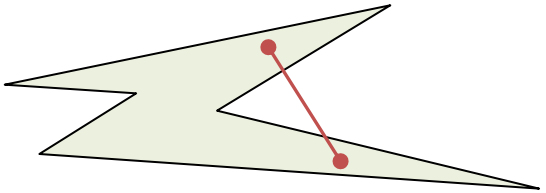
St_Distance(a,b)
ジオメトリ間の距離



St_Contains(a,b)
ジオメトリの包含判定



St_Crosses(a,b)
ジオメトリの交差判定



高速化手法（1/2）- Bounding Box

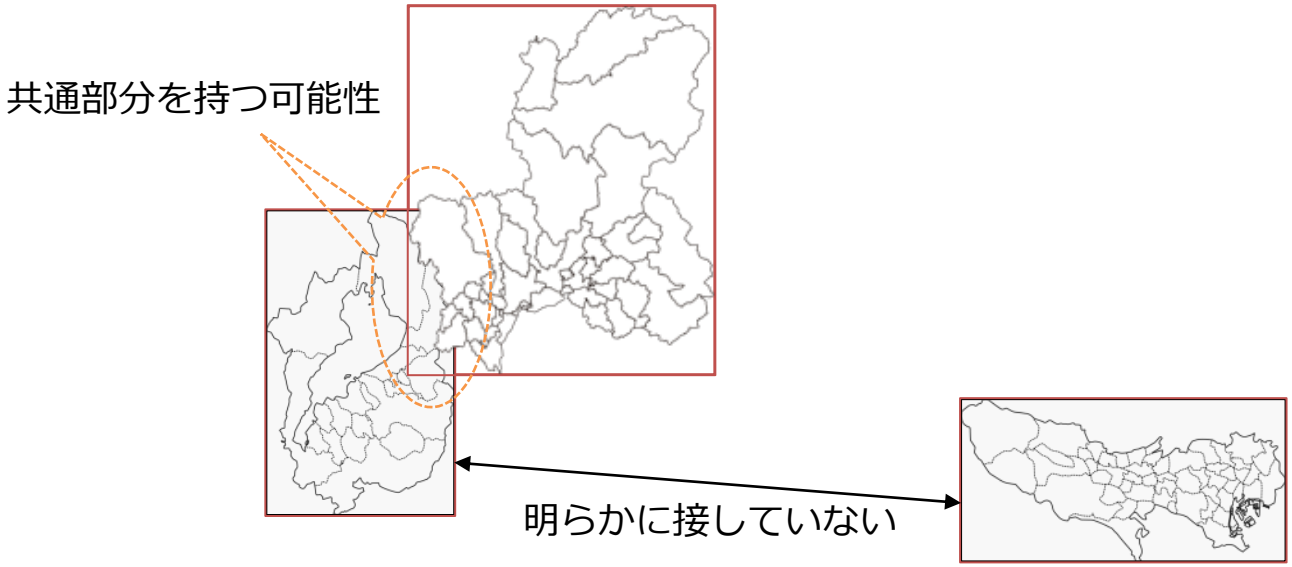
Bounding Boxとは

- 複雑な形状のポリゴンを包摂する最も小さな矩形領域
- (x_1, y_1) - (x_2, y_2) で表現できるのでデータ量が少ない
- PostgreSQL / PostGISが geometry 型を保存する時に自動的に生成される。

Bounding Boxの効果

- 空間関係演算を行う前に、『明らかに接していない』ものを識別できる。

➔ 重なりを含むジオメトリのみ判定を行う事で、計算リソースを節約。



高速化手法（2/2）- GiSTインデックス

GiSTインデックスとは

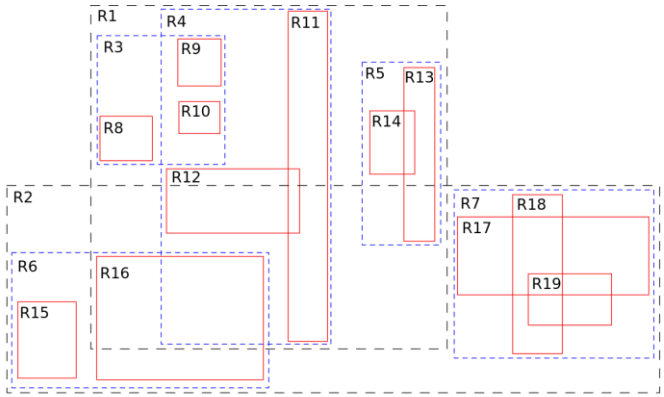
- PostgreSQLのGiST（Generalized Search Tree）は、ツリー構造を持つインデックスを一般化したフレームワーク。
- PostGISのGeometry型は、GiST上にR木を実装している。

GiSTのR木でできること

- 包含（@演算子）や重なり（&&演算子）判定で、インデックスを用いた絞込み
- 特に多数のポリゴン×ポイントの重なり判定で計算量が増大しやすい

➔ 事前の絞り込みで計算量を抑え込む

```
# 国土地理院の市町村形状データをインポート
$ shp2pgsql N03-20_200101.shp | psql gistest
gistest=# ¥d+
                        List of relations
 Schema |    Name     |   Type   | Owner  |    Size    |
--------+-------------+----------+--------+------------+
 public | geo_japan   | table    | kaigai | 243 MB     |

gistest=# ¥di+
                                    List of relations
 Schema |         Name         | Type  | Owner  |   Table    |  Size   |
--------+----------------------+-------+--------+------------+---------+
 public | geo_japan_pkey       | index | kaigai | geo_japan  | 2616 kB |
 public | geo_japan_geom_idx   | index | kaigai | geo_japan  | 14 MB   |
```



なぜGPUでSQLを高速化できるのか？

SQL処理の中でも、全件スキャンを伴うようなワークロードに強い

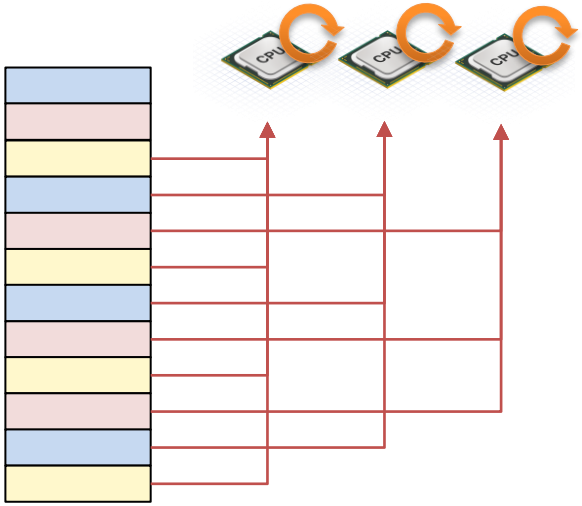
GPUの特徴

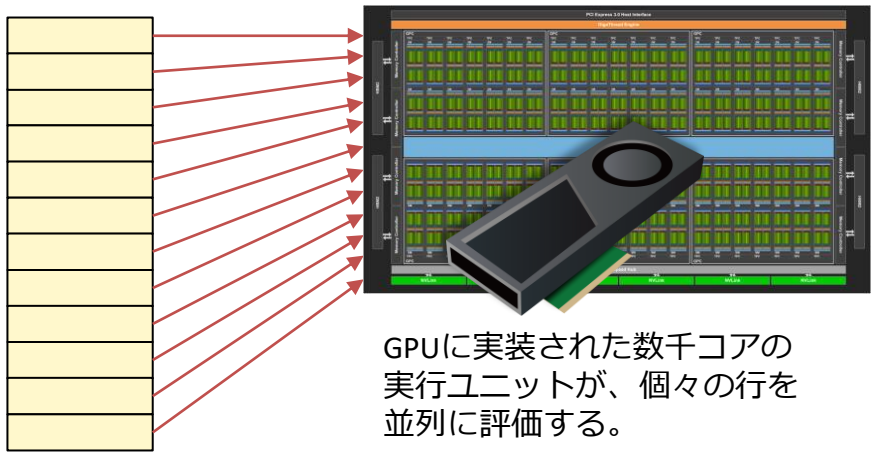
- 数千演算コアを搭載する計算能力
- 1.0TB/sに迫るメモリバンド

➔「同じ演算を多量のデータに対して実行する」ために作られたプロセッサ
（e.g 行列演算）

SQLワークロードの特性

- 「同じ処理を大量の行に対して実行する」
（例：WHERE句の評価、JOINやGROUP BY処理）





GPUに実装された数千コアの
実行ユニットが、個々の行を
並列に評価する。

GPUでPostGIS関数を実行する（1/2）

```
postgres=# ¥d _gistest
                            Table "public._gistest"
 Column |   Type    | Collation | Nullable |               Default
--------+-----------+-----------+----------+-------------------------------------
 id     | integer   |           | not null | nextval('_gistest_id_seq'::regclass)
 a      | geometry  |           |          |
 b      | geometry  |           |          |

postgres=# explain verbose select * from _gistest where st_contains(a,b);
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Custom Scan (GpuScan) on public._gistest  (cost=4251.50..4251.50 rows=1 width=196)
   Output: id, a, b
   GPU Filter: st_contains(_gistest.a, _gistest.b)
   GPU Preference: None
   Kernel Source: /var/lib/pgdata/pgsql_tmp/pgsql_tmp_strom_21028.4.gpu
   Kernel Binary: /var/lib/pgdata/pgsql_tmp/pgsql_tmp_strom_21028.5.ptx
(6 rows)
```

GPUでPostGIS関数を実行する（2/2）

```
$ less /var/lib/pgdata/pgsql_tmp/pgsql_tmp_strom_21028.4.gpu
        :
#include "cuda_postgis.h"
#include "cuda_gpuscan.h"

DEVICE_FUNCTION(cl_bool)
gpuscan_quals_eval(kern_context *kcxt,
                   kern_data_store *kds,
                   ItemPointerData *t_self,
                   HeapTupleHeaderData *htup)
{
  void *addr __attribute__((unused));
  pg_geometry_t KVAR_2;
  pg_geometry_t KVAR_3;
  assert(htup != NULL);
  EXTRACT_HEAP_TUPLE_BEGIN(addr, kds, htup);
  EXTRACT_HEAP_TUPLE_NEXT(addr);
  pg_datum_ref(kcxt,KVAR_2,addr); // pg_geometry_t
  EXTRACT_HEAP_TUPLE_NEXT(addr);
  pg_datum_ref(kcxt,KVAR_3,addr); // pg_geometry_t
  EXTRACT_HEAP_TUPLE_END();

  return EVAL(pgfn_st_contains(kcxt, KVAR_2, KVAR_3));
}
        :
```

SQLのWHERE句を
評価するために
自動生成された関数

列A、列Bから
ジオメトリ型の
データをロード

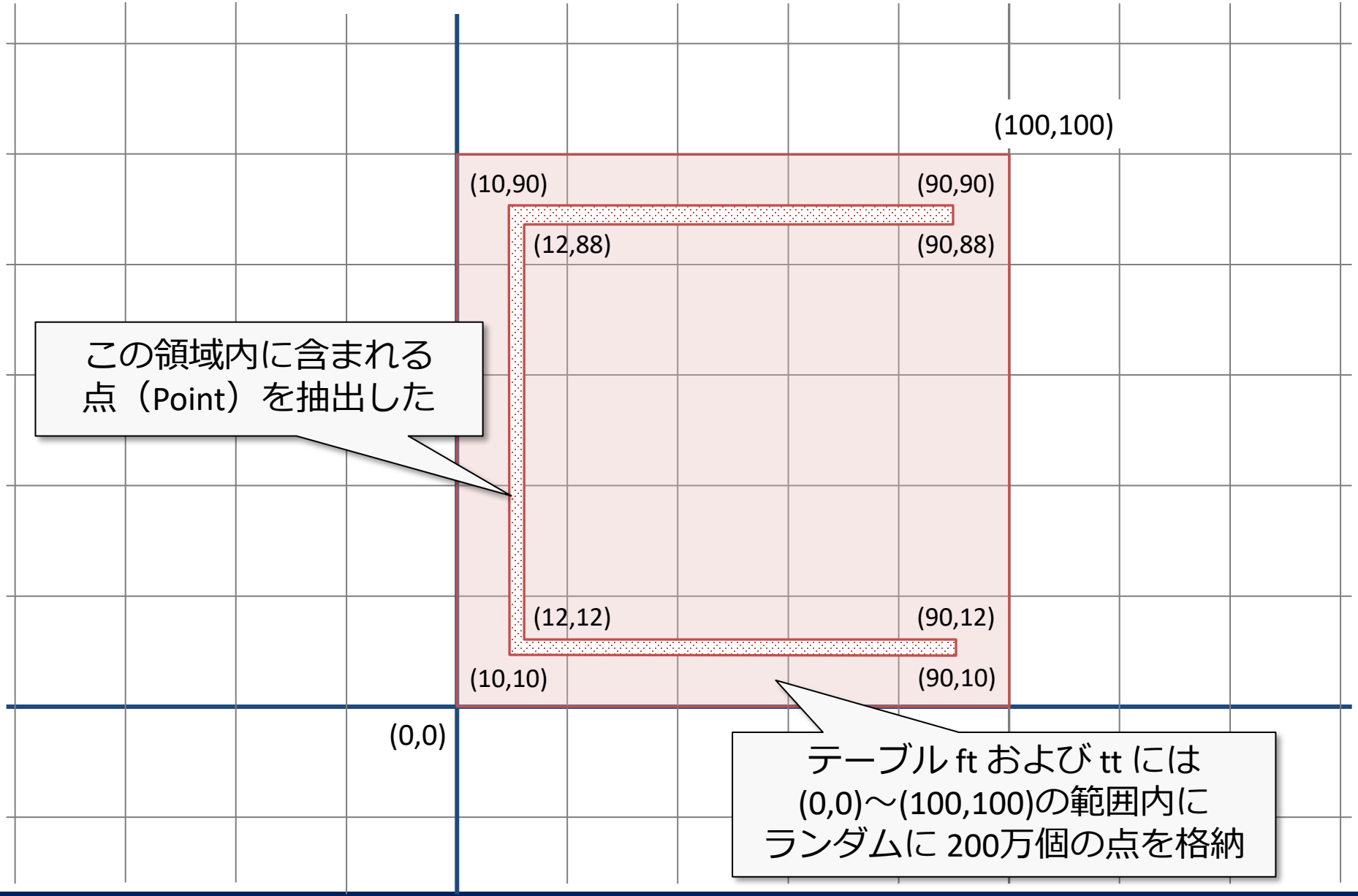
ロードしたデータで
GPU版 st_contains() 関数を
呼出し

検索処理のパフォーマンス（1/2）

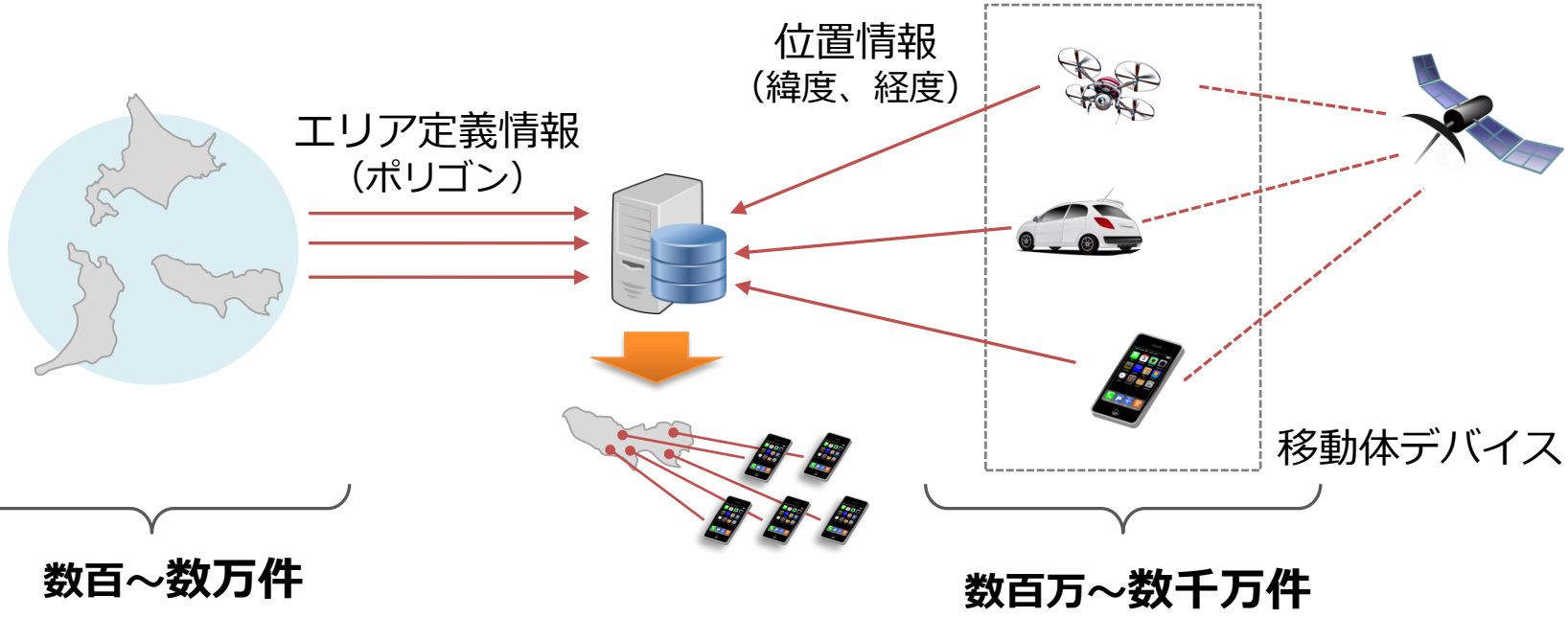
```
--- GpuScan (GPU版PostGIS) + Gstore_Fdw
=# SELECT count(*) FROM ft
    WHERE st_contains('polygon ((10 10,90 10,90 12,12 12,12 88,90 88,90 90,¥
                                 10 90,10 10))', st_makepoint(x,y));
 count
-------
 94440
(1 row)

Time: 75.466 ms

--- 通常版PostGIS + PostgreSQLテーブル
=# SET pg_strom.enabled = off;
SET
=# SELECT count(*) FROM tt
    WHERE st_contains('polygon ((10 10,90 10,90 12,12 12,12 88,90 88,90 90,¥
                                 10 90,10 10))', st_makepoint(x,y));
 count
-------
 94440
(1 row)

Time: 332.300 ms
```

200万個のPointから、
指定領域内の数をカウント

検索処理のパフォーマンス（2/2）



ポリゴン × 座標の突合



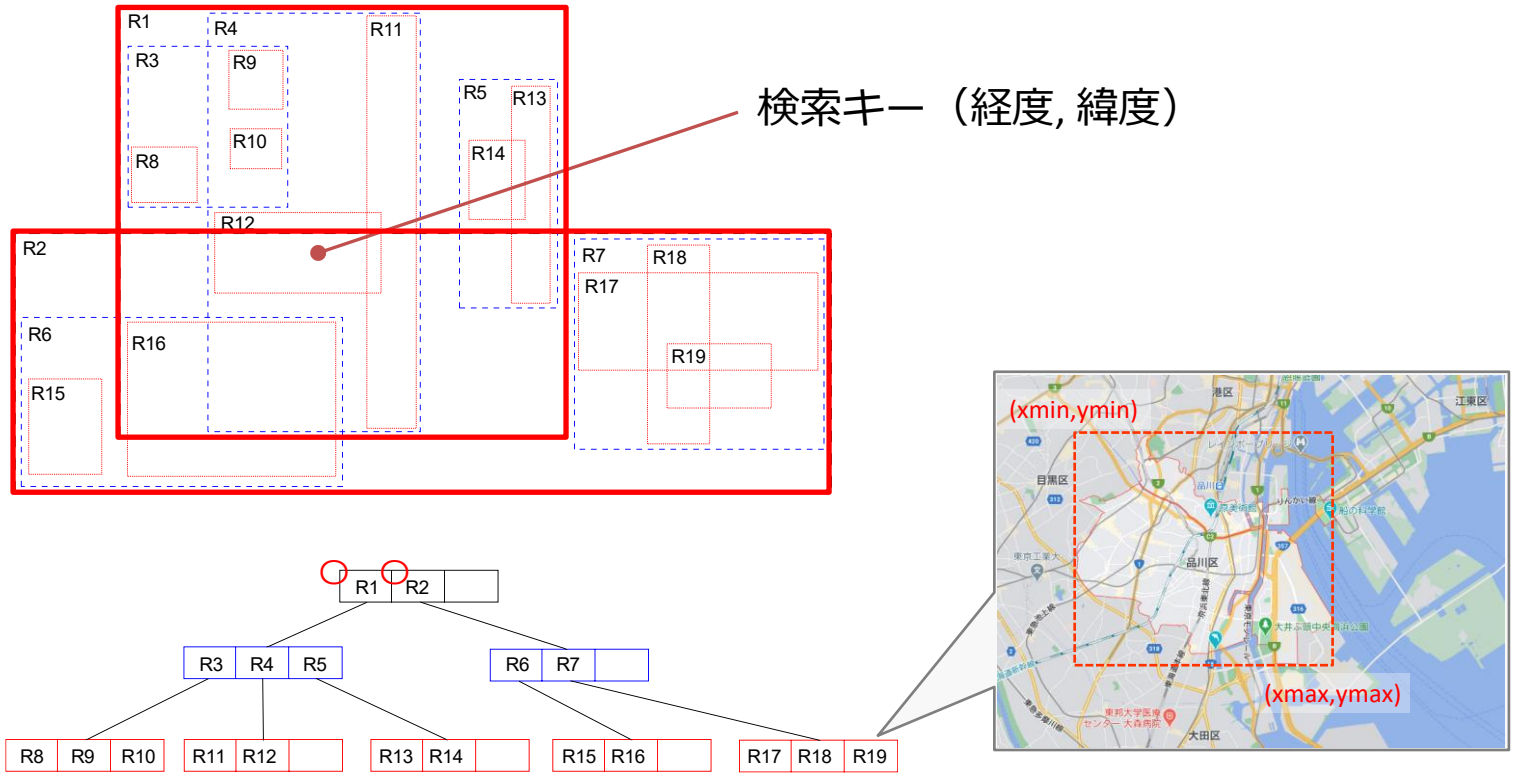
- ナイーブなポリゴン×座標の突合を行おうとすると、
  数万×数千万 = 数兆通りの組合せ、で計算量が爆発。

➔ GPUが数千コアを搭載していても、
  計算量が多すぎて焼け石に水。

焼け石（GPU）に ➔
水冷モジュール

GPUでGiSTインデックス

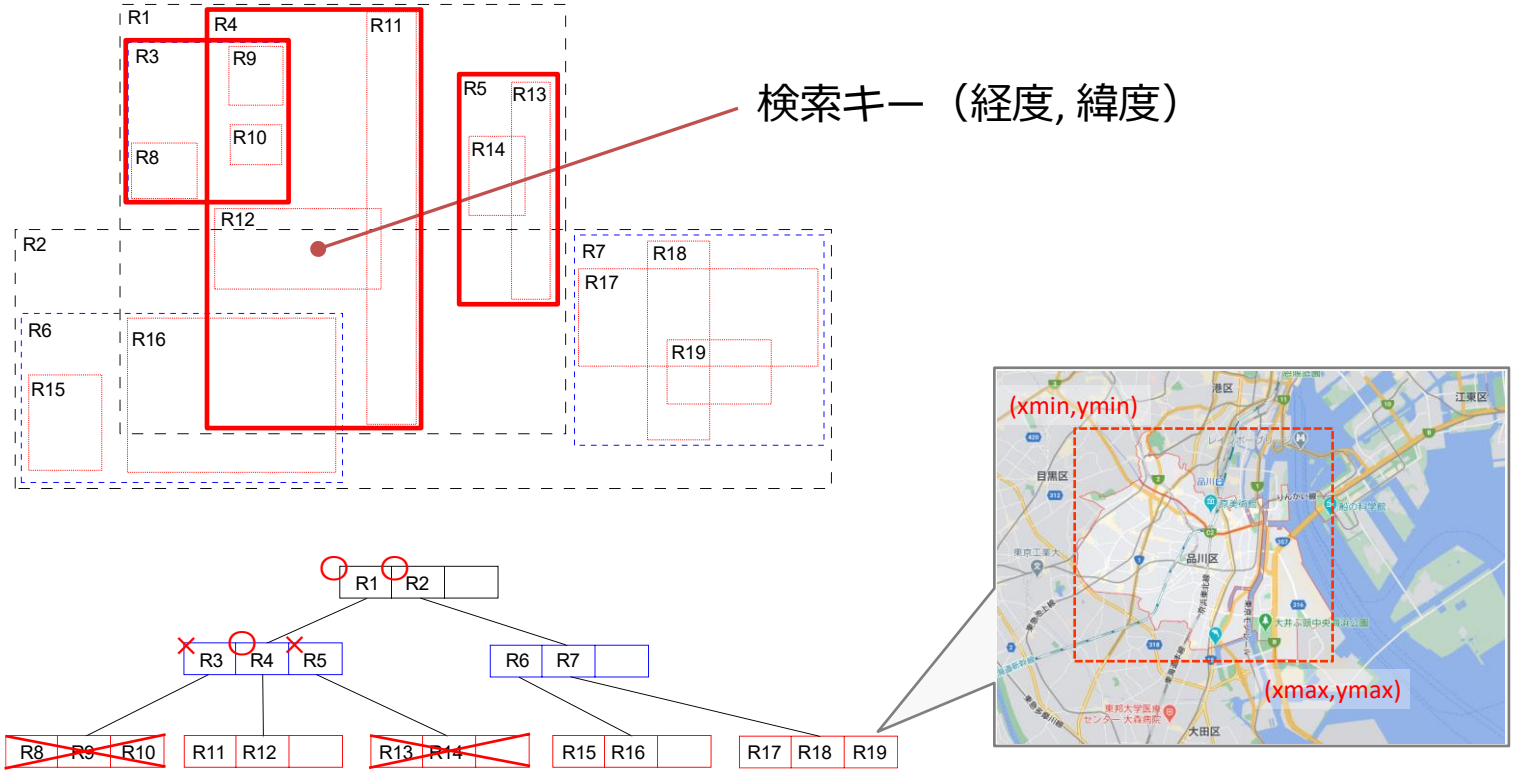
GiSTインデックス（R木）の仕組み



GiSTインデックス（R木）の仕組み

- ✓ R1は(R3,R4,R5)を全て包含する矩形領域の$(X_{min},Y_{min})-(X_{max},Y_{max})$と、下位ノードへのポインタ
- ✓ R4は(R11,R12)を全て包含する矩形領域の$(X_{min},Y_{min})-(X_{max},Y_{max})$と、下位ノードへのポインタ
- ✓ R12は対象ジオメトリを包含する矩形領域の$(X_{min},Y_{min})-(X_{max},Y_{max})$と ItemPointer を含む。
- ✓ 各ノード（BLCKSZ）内のエントリを順に評価。マッチしたものを次の階層へすすめる。
- ✓ 階層ごとに繰り返し評価が発生するので、B-tree並みに速くはできない。

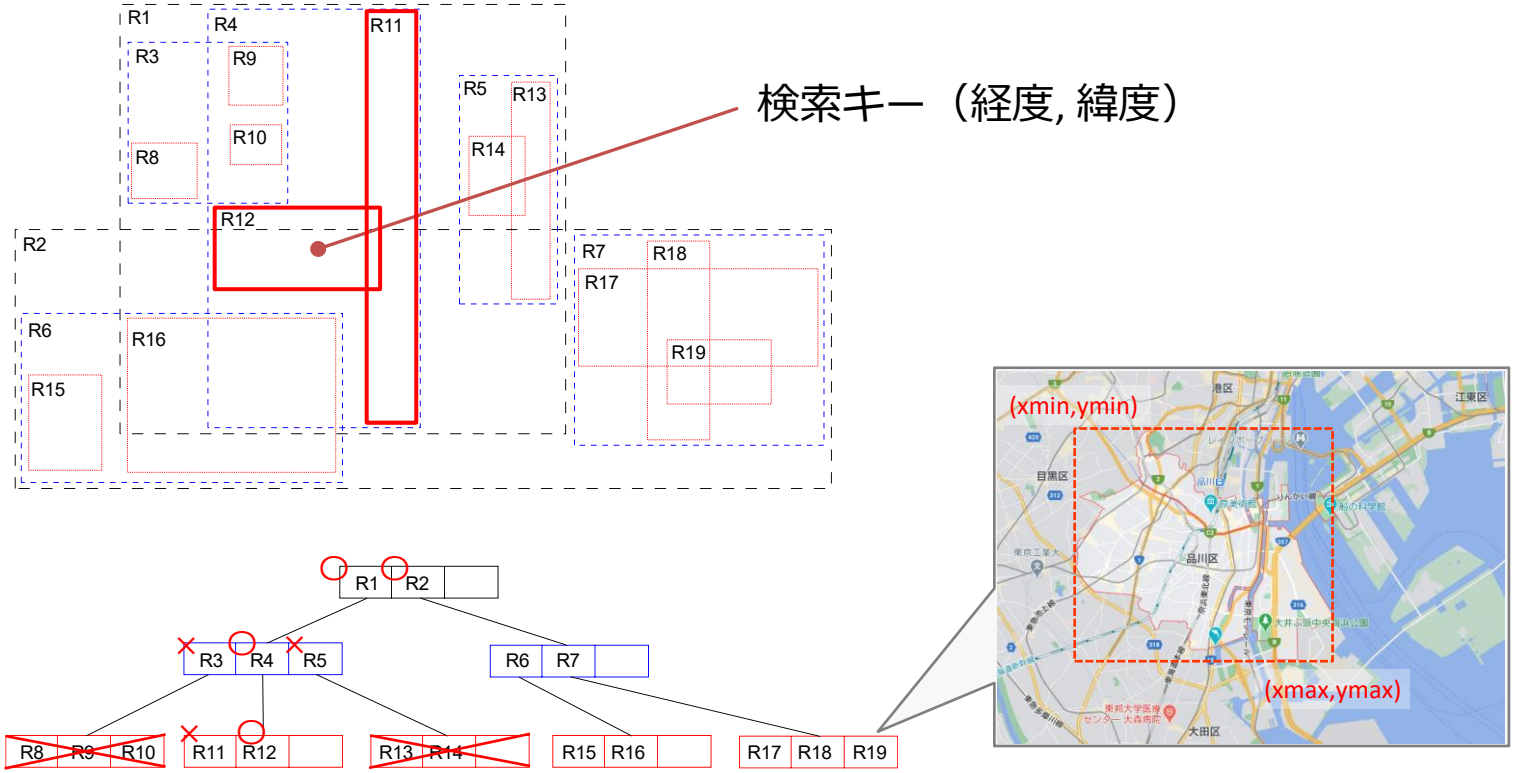
GiSTインデックス（R木）の仕組み



GiSTインデックス（R木）の仕組み

- ✓ R1は(R3,R4,R5)を全て包含する矩形領域の(X_{min},Y_{min}) – (X_{max},Y_{max})と、下位ノードへのポインタ
- ✓ R4は(R11,R12)を全て包含する矩形領域の(X_{min},Y_{min}) – (X_{max},Y_{max})と、下位ノードへのポインタ
- ✓ R12は対象ジオメトリを包含する矩形領域の(X_{min},Y_{min}) – (X_{max},Y_{max})と ItemPointer を含む。
- ✓ 各ノード（BLCKSZ）内のエントリを順に評価。マッチしたものを次の階層へすすめる。
- ✓ 階層ごとに繰り返し評価が発生するので、B-tree並みに速くはできない。

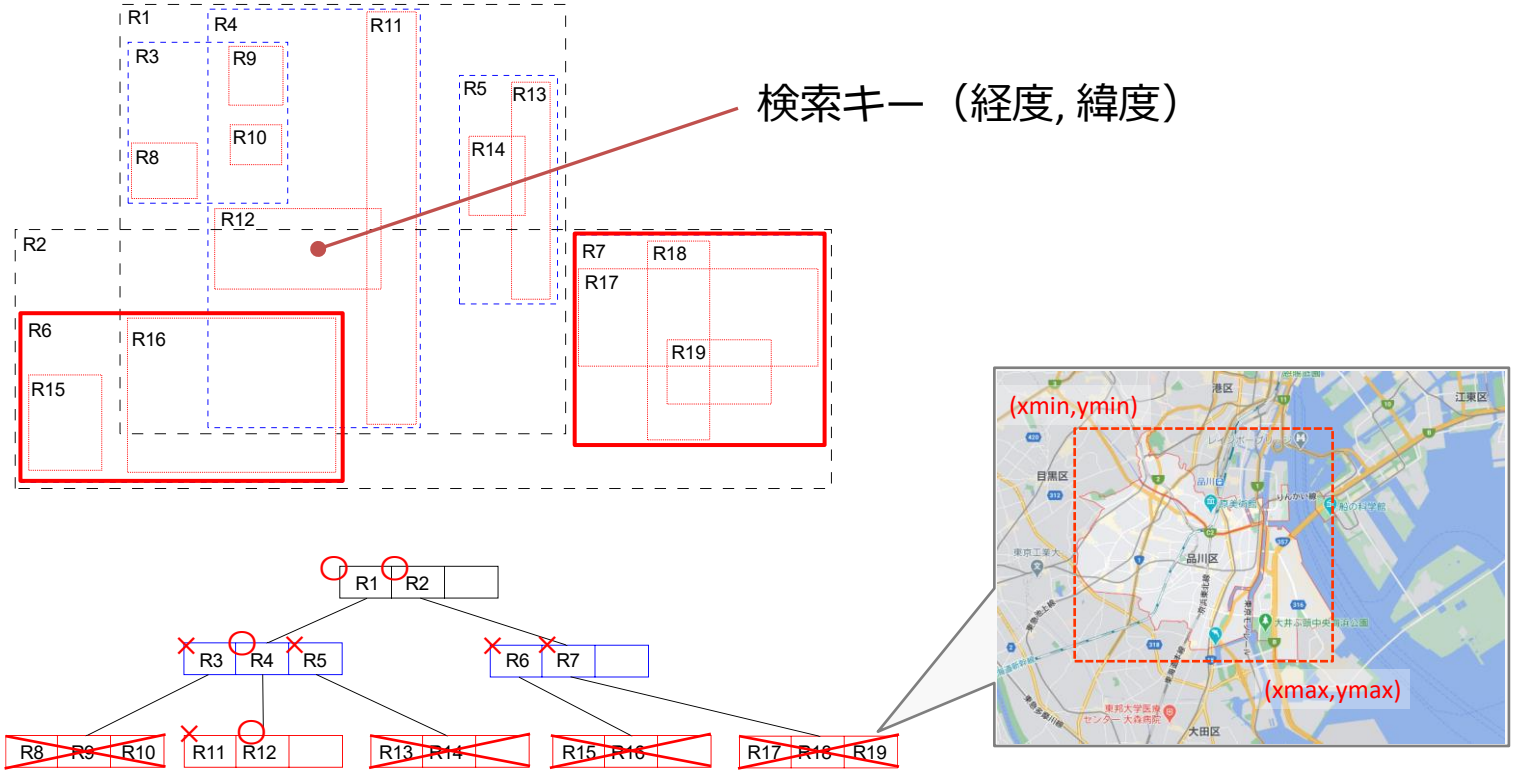
GiSTインデックス（R木）の仕組み



GiSTインデックス（R木）の仕組み

- ✓ R1は(R3,R4,R5)を全て包含する矩形領域の(X_{min},Y_{min}) – (X_{max},Y_{max})と、下位ノードへのポインタ
- ✓ R4は(R11,R12)を全て包含する矩形領域の(X_{min},Y_{min}) – (X_{max},Y_{max})と、下位ノードへのポインタ
- ✓ R12は対象ジオメトリを包含する矩形領域の(X_{min},Y_{min}) – (X_{max},Y_{max})と ItemPointer を含む。
- ✓ 各ノード（BLCKSZ）内のエントリを順に評価。マッチしたものを次の階層へすすめる。
- ✓ 階層ごとに繰り返し評価が発生するので、B-tree並みに速くはできない。

GiSTインデックス（R木）の仕組み

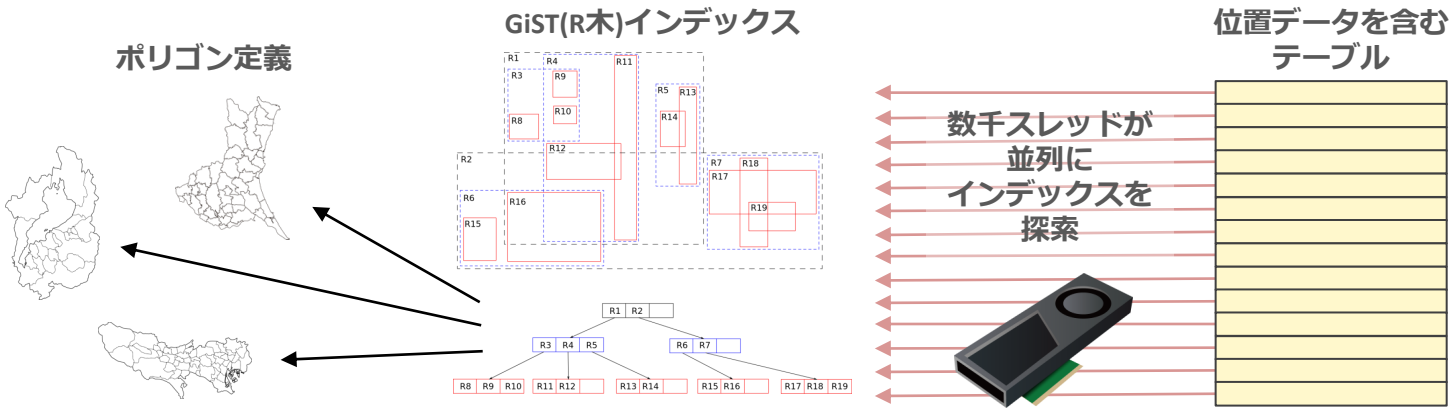


GiSTインデックス（R木）の仕組み

- ✓ R1は(R3,R4,R5)を全て包含する矩形領域の$(X_{min},Y_{min})-(X_{max},Y_{max})$と、下位ノードへのポインタ
- ✓ R4は(R11,R12)を全て包含する矩形領域の$(X_{min},Y_{min})-(X_{max},Y_{max})$と、下位ノードへのポインタ
- ✓ R12は対象ジオメトリを包含する矩形領域の$(X_{min},Y_{min})-(X_{max},Y_{max})$と ItemPointer を含む。
- ✓ 各ノード（BLCKSZ）内のエントリを順に評価。マッチしたものを次の階層へすすめる。
- ✓ 階層ごとに繰り返し評価が発生するので、B-tree並みに速くはできない。

GPU版GiSTインデックス

多角形 × 点の重なり判定などを、GpuJoinの一要素として実装

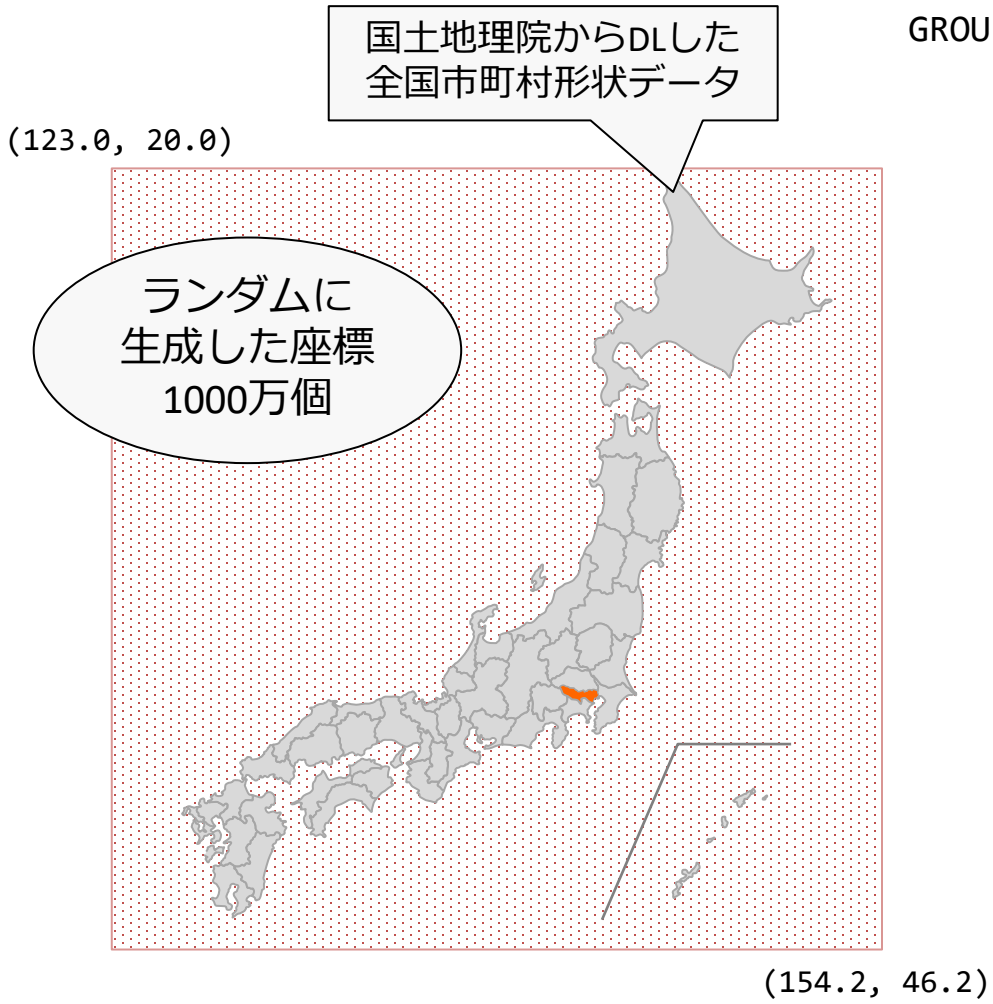


仕組み

- 多角形（エリア定義情報）を保持するテーブルと、位置情報（緯度経度）を保持するテーブルとの JOIN にGiSTインデックスを使用できる。
- JOIN処理時、テーブルとインデックスの双方をGPUにロード。
  先ずGiSTインデックス上のBounding-Box（矩形領域）を使って荒く絞り込み、次にテーブル上の多角形（ポリゴン）と「当たり判定」を行う。
- GPUの数千コアをフル稼働してGiSTインデックスを探索する。
  単純に並列度が高い分、検索速度も速くなるはず。

➔ が、そうは問屋が卸さなかった。。。

簡易テスト：ランダムな点とSt_Contains（9月末ごろ）




```
SELECT n03_001,n03_004,count(*)
  FROM geo_japan j, geopoint p
 WHERE st_contains(j.geom, st_makepoint(x,y))
   AND j.n03_001 like '東京都'
GROUP BY n03_001,n03_004;
```

n03_001	n03_004	count
東京都	あきる野市	105
東京都	三宅村	76
東京都	三鷹市	17
東京都	世田谷区	67
東京都	中央区	12
東京都	中野区	18
東京都	八丈町	105
：	：	：
東京都	豊島区	14
東京都	足立区	55
東京都	青ヶ島村	7
東京都	青梅市	117

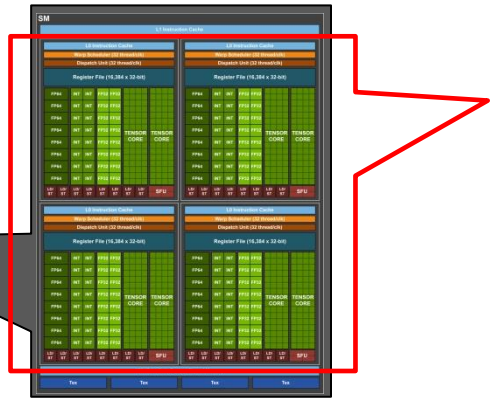
(63 rows)

CPU版：24.892s

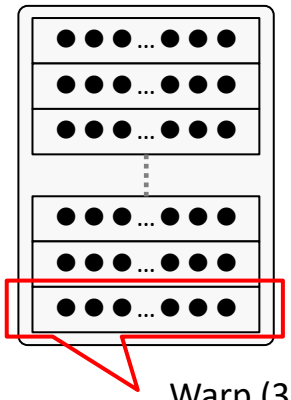
GPU版：33.841s（遅い！）

前提）GPUスレッドのスケジューリングについて

GPU Block Diagram
(Tesla V100; 80SMs)



Streaming Multiprocessors
(64CURE cores/SM)



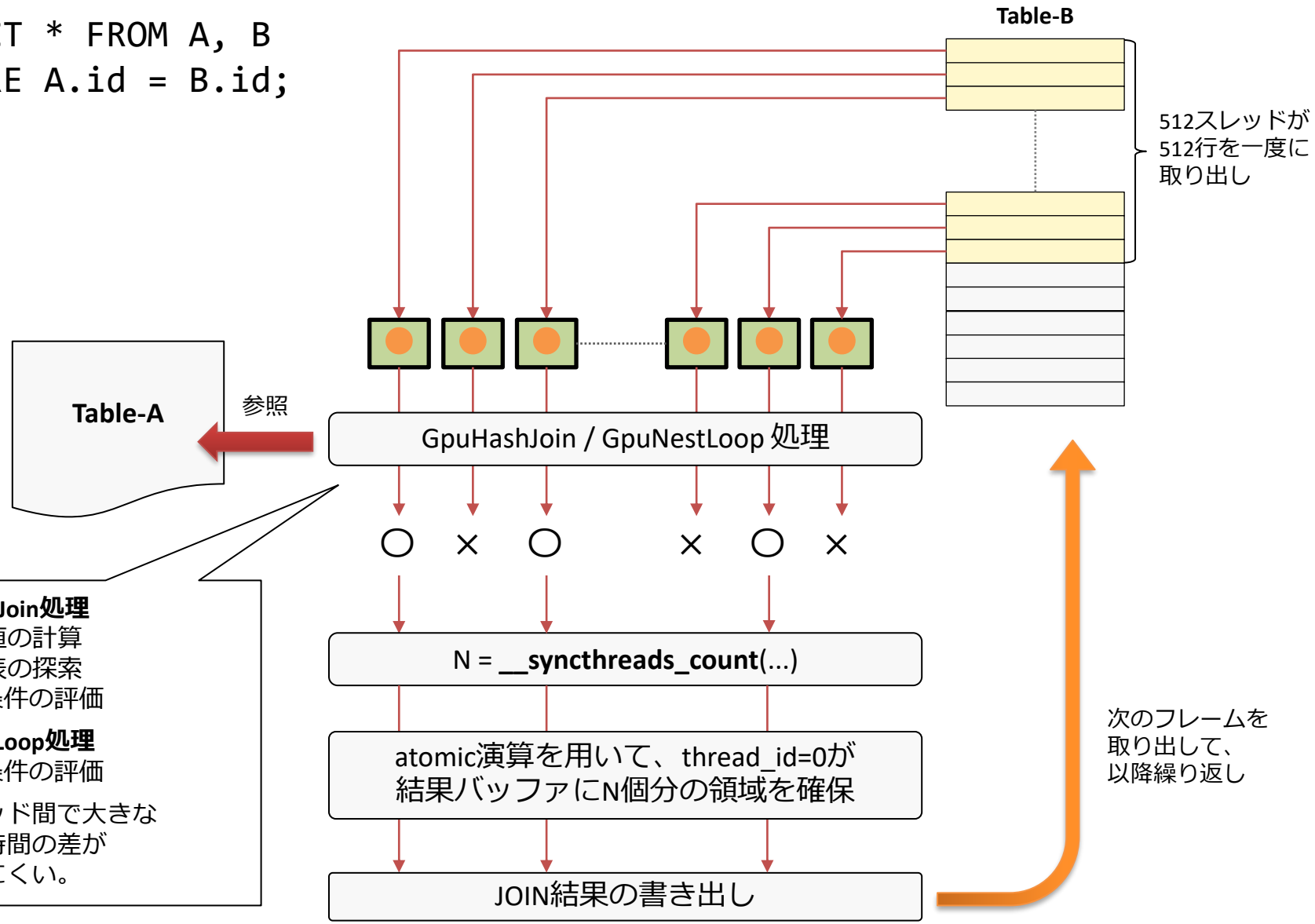
Thread Group
(1～1024 Threads)

Warp (32 threads)

- GPUはStreaming Multiprocessor(SM)と呼ばれる単位でコアがグループ化されている。
  - ✓ レジスタやL1キャッシュ（共有メモリ）はSM単位のリソース
- 同じSM上で実行されるスレッドの組を Thread Group と呼び、リソースが許せば最大で 1024 スレッドまで並行で実行する事ができる。
  - ✓ スレッドの切り替えはH/W的に実現され、例えばDRAMアクセス待ちのタイミングで切り替わったり。
- スレッドはWarp (32threads) 単位でスケジューリングされる。同一Warp内のスレッドは、同時には同じ命令しか実行できない。
  - ✓ 行列計算のように、各コアの負荷が均等になるタイプの処理であればGPUの使用効率は最大化。
  - ✓ スレッド毎に処理時間がバラバラだと、一部のコアが遊んでしまい、処理効率の低下を招く。

前提）GpuJoinの並列処理デザイン

```
SELECT * FROM A, B
 WHERE A.id = B.id;
```



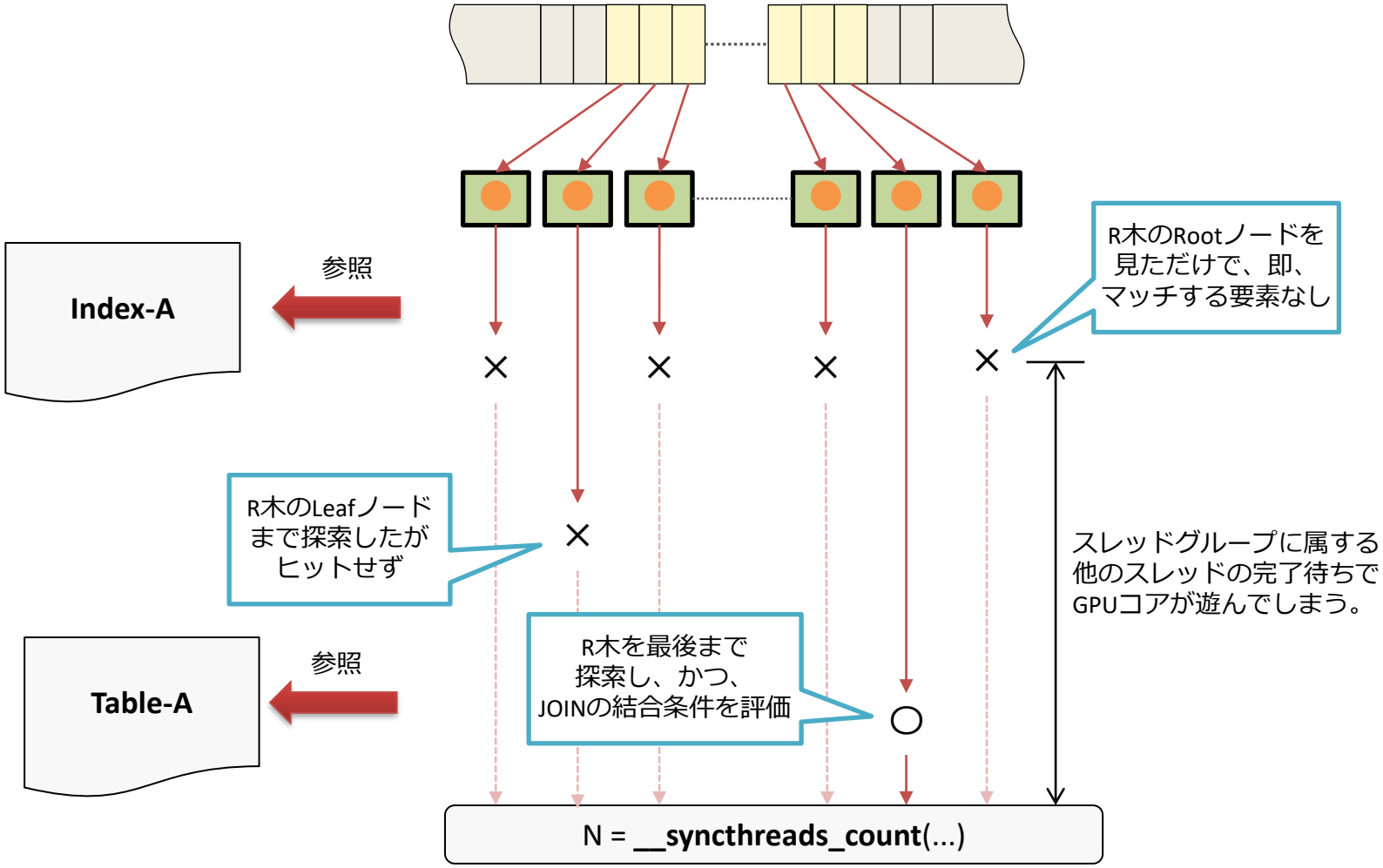
GpuHashJoin処理
- Hash値の計算
- Hash表の探索
- JOIN条件の評価

GpuNestLoop処理
- JOIN条件の評価

➔スレッド間で大きな処理時間の差がつきにくい。

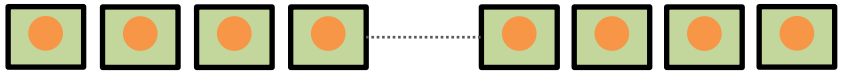
GPUでのインデックス探索の課題

スレッド間の処理時間の差が大きく、同期待ちを招く



ナイーブなGPU版GiSTインデックス実装（９月末）

GPUの利用効率が上がらず、しばしばCPUより低速だった



繰り返し

各スレッドが１行分のデータをロード
GiST-Index探索用のキー値を抽出する。

キー値を用いてGiST-Indexを探索する

GiST-Indexにヒットした？ — No

JOIN結合条件を評価する

JOIN結合条件は真？ — No

found=true

nitems = __syncthreads_count(found);

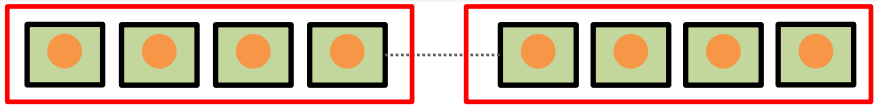
found=false

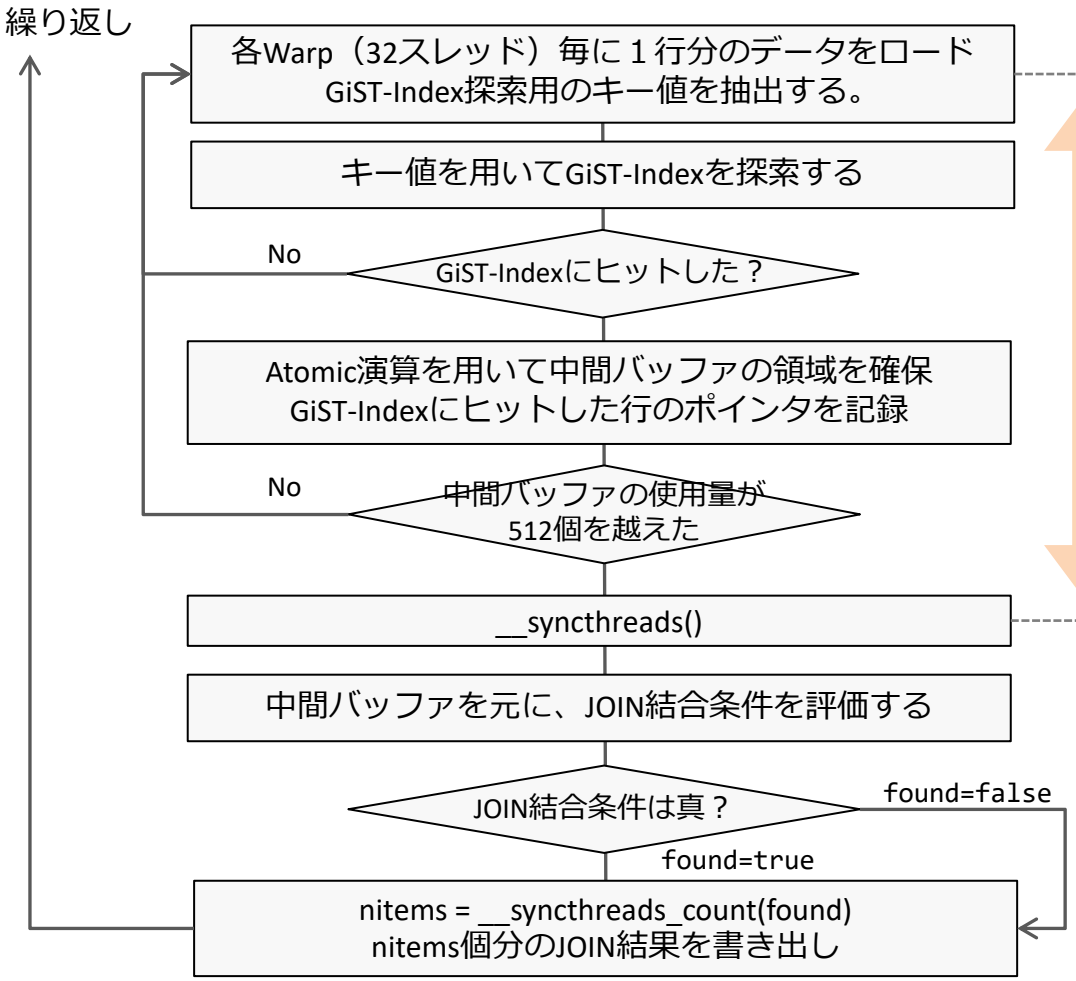
nitems個分のJOIN結果を書き出し

スレッドグループ（512スレッド）内に、
１個でもGiST-IndexのLeafまで降下して
探索したり、JOIN結合条件の評価まで
行うスレッドがいると、他の511スレッ
ドはその完了を待たされてしまう。
結果として、GPUコアの使用率が全く
向上しないという事になる。

GPUのスレッドスケジューリングを意識した実装（11月頭）

できる限りコアが遊ばないよう、同期ポイントを最小化



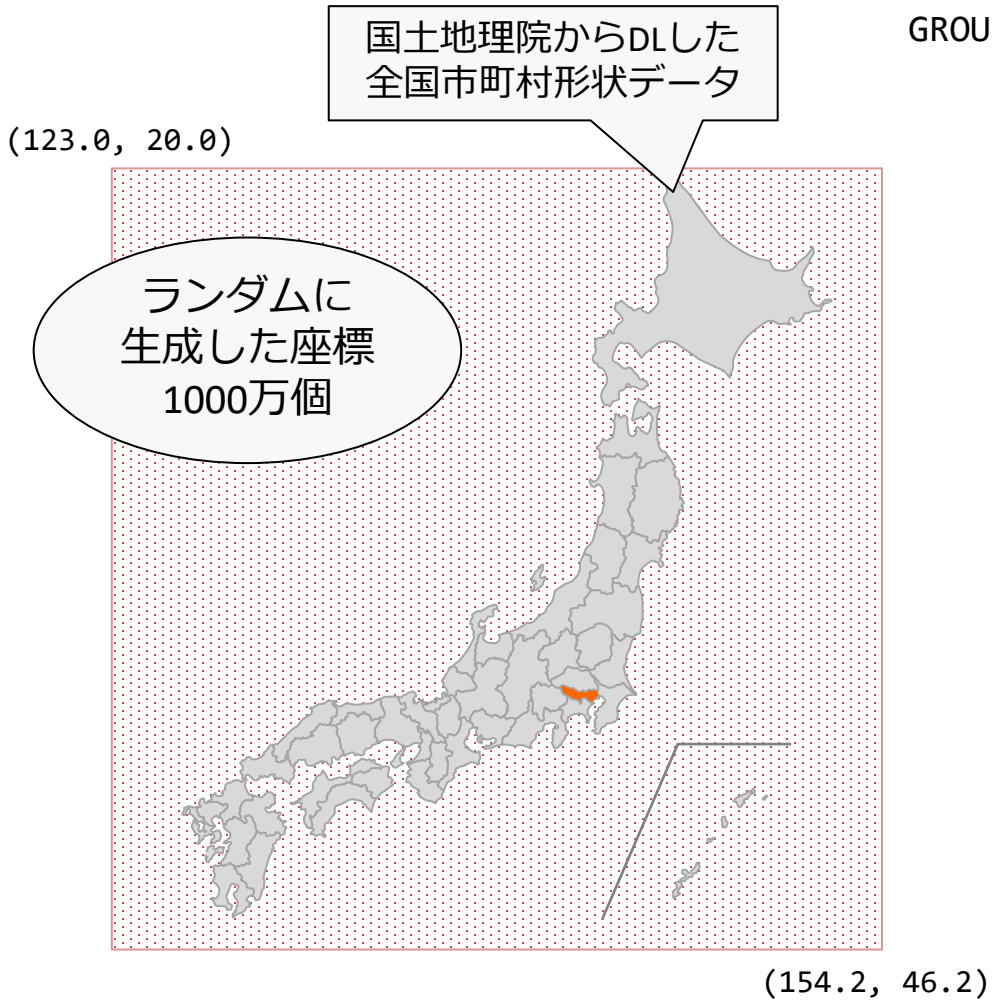


中間バッファの使用量に余裕がある限り、同期ポイントに達する前にどんどんReadポインタを進めて GiST-Index の探索を進める。

➔ 一部のWarp（スレッド）で探索に時間を要しても、他のWarpは先に次の行を処理できるため、GPUコアの使用率を高くできる。

簡易テスト：ランダムな点とSt_Contains（最新版）

```
SELECT n03_001,n03_004,count(*)
  FROM geo_japan j, geopoint p
 WHERE st_contains(j.geom, st_makepoint(x,y))
   AND j.n03_001 like '東京都'
GROUP BY n03_001,n03_004;
```



n03_001	n03_004	count
東京都	あきる野市	105
東京都	三宅村	76
東京都	三鷹市	17
東京都	世田谷区	67
東京都	中央区	12
東京都	中野区	18
東京都	八丈町	105
：	：	：
東京都	豊島区	14
東京都	足立区	55
東京都	青ヶ島村	7
東京都	青梅市	117

(63 rows)

CPU版：24.892s

GPU版： 3.733s

６倍強の
高速化

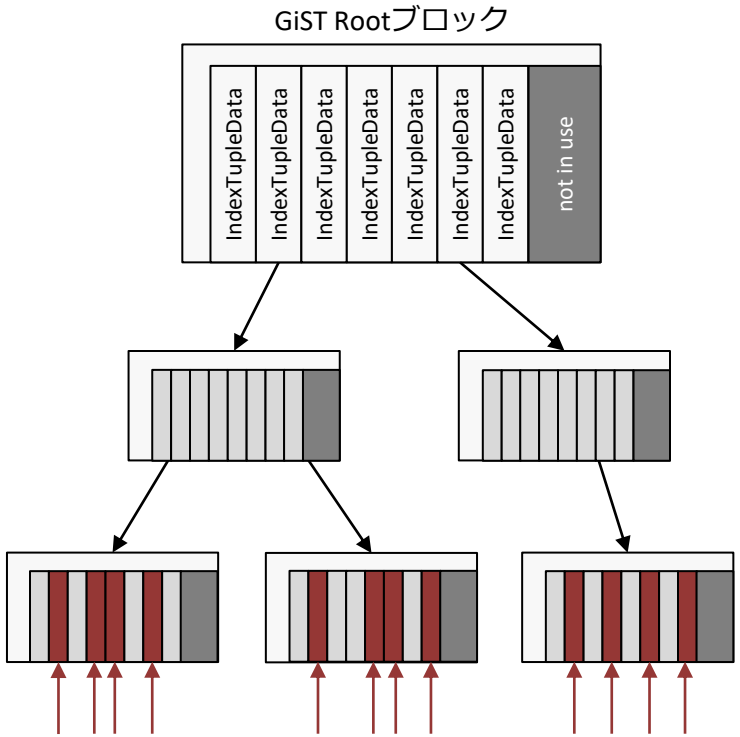
簡易テストの実行計画（CPU版）

```
postgres=# EXPLAIN (analyze, costs off)
            SELECT n03_001,n03_004,count(*)
              FROM geo_japan j, geopoint p
             WHERE st_contains(j.geom, st_makepoint(x,y))
               AND j.n03_001 like '東京都’
             GROUP BY n03_001,n03_004;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Finalize GroupAggregate (actual time=24880.443..24880.682 rows=63 loops=1)
   Group Key: j.n03_001, j.n03_004
   ->  Gather Merge (actual time=24880.430..24892.342 rows=299 loops=1)
         Workers Planned: 4
         Workers Launched: 4
         ->  Partial GroupAggregate (actual time=24855.860..24855.949 rows=60 loops=5)
               Group Key: j.n03_001, j.n03_004
               ->  Sort (actual time=24855.846..24855.865 rows=529 loops=5)
                     Sort Key: j.n03_001, j.n03_004
                     Sort Method: quicksort  Memory: 64kB
                     ->  Nested Loop (actual time=50.124..24854.788 rows=529 loops=5)
                           ->  Parallel Seq Scan on geopoint p (actual time=0.120..258.900
                                                                rows=2000000 loops=5)
                           ->  Index Scan using geo_japan_geom_idx on geo_japan j
                                 (actual time=0.012..0.012 rows=0 loops=10000000)
                               Index Cond: (geom ~ st_makepoint(p.x, p.y))
                               Filter: (((n03_001)::text ~~ '東京都'::text) AND
                                       st_contains(geom, st_makepoint(p.x, p.y)))
                               Rows Removed by Filter: 0
 Planning Time: 0.160 ms
 Execution Time: 24892.586 ms
(22 rows)
```

簡易テストの実行計画（GPU版）

```
postgres=# EXPLAIN (analyze, costs off)
            SELECT n03_001,n03_004,count(*)
              FROM geo_japan j, fgeopoint p
             WHERE st_contains(j.geom, st_makepoint(x,y))
               AND j.n03_001 like '東京都’
             GROUP BY n03_001,n03_004;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 HashAggregate (actual time=3725.900..3725.938 rows=63 loops=1)
   Group Key: j.n03_001, j.n03_004
   ->  Custom Scan (GpuJoin) on fgeopoint p (actual time=3725.169..3725.415 rows=2646 loops=1)
         Outer Scan: fgeopoint p (never executed)
         Depth 1: GpuGiSTJoin
                  HeapSize: 7841.91KB (estimated: 3113.70KB), IndexSize: 13.28MB
                  IndexFilter: (j.geom ~ st_makepoint(p.x, p.y)) on geo_japan_geom_idx
                  Rows Fetched by Index: 5065
                  JoinQuals: st_contains(j.geom, st_makepoint(p.x, p.y))
         GPU Preference: GPU0 (Tesla V100-PCIE-16GB)
         ->  Seq Scan on geo_japan j (actual time=0.154..17.959 rows=6173 loops=1)
               Filter: ((n03_001)::text ~~ '東京都'::text)
               Rows Removed by Filter: 112726
 Planning Time: 0.297 ms
 Execution Time: 3733.403 ms
(15 rows)
```

GPU版に固有の最適化



あらかじめ、明らかに条件に該当しないエントリ
（この場合は ‘東京都’ 以外）を無効化

```
EXPLAIN
SELECT n03_001,n03_004,count(*)
  FROM geo_japan j, geopoint p
 WHERE st_contains(j.geom, st_makepoint(x,y))
   AND j.n03_001 like '東京都’
 GROUP BY n03_001,n03_004;

                    QUERY PLAN
-----------------------------------------------------
 GroupAggregate
  Group Key: j.n03_001, j.n03_004
   -> Sort
      Sort Key: j.n03_001, j.n03_004
     -> Nested Loop
       -> Seq Scan on geopoint p
       -> Index Scan using geo_japan_geom_idx on geo_japan j
            Index Cond: (geom ~ st_makepoint(p.x, p.y))
            Filter: (((n03_001)::text ~~ '東京都') AND
                  st_contains(geom, st_makepoint(p.x, p.y)))
 (9 rows)
```

- PostgreSQLの場合、Indexに付随する条件句の評価は、Indexを用いて候補となる行を取り出した後にしか行えない。つまり、結果的に無駄となる行フェッチが発生する。
- GPU版GiSTインデックスでは、予め「明らかに条件に該当しないエントリ」のLeaf要素を無効化するため、何度も何度も自明な条件句の評価を行う必要はない。

➔ 『１週間分のデータから直近30分のイベントだけを取り出す』といった特性がある場合に、通常のGiSTインデックス探索よりも有利に働く。

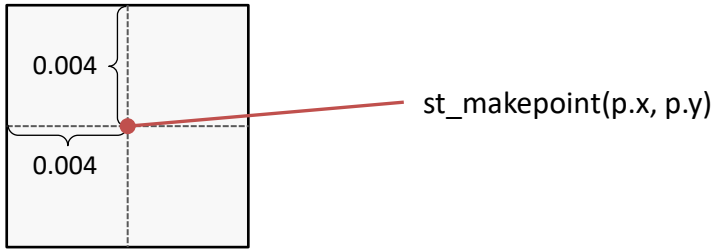
小噺）ちょっと感動した話

```
postgres=# EXPLAIN  SELECT n03_001,n03_004,count(*)
                      FROM geo_japan j, geopoint p
                     WHERE st_dwithin(j.geom, st_makepoint(x,y), 0.004)
                       AND j.n03_001 like '東京都’
                     GROUP BY n03_001,n03_004;
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------
 HashAggregate  (cost=1936845.15..1936893.73 rows=4858 width=29)
   Group Key: j.n03_001, j.n03_004
   ->  Custom Scan (GpuJoin) on geopoint p  (cost=159401.97..1480539.90 rows=60840700 width=21)
         Outer Scan: geopoint p  (cost=0.00..163696.15 rows=10000115 width=16)
         Depth 1: GpuGiSTJoin(nrows 10000115...60840700)
                  HeapSize: 3113.70KB
                  IndexFilter: (j.geom && st_expand(st_makepoint(p.x, p.y),
                                                    '0.004'::double precision)) on geo_japan_geom_idx
                  JoinQuals: st_dwithin(j.geom, st_makepoint(p.x, p.y), '0.004'::double precision)
         GPU Preference: GPU0 (Tesla V100-PCIE-16GB)
         ->  Seq Scan on geo_japan j  (cost=0.00..8928.24 rows=6084 width=1868)
               Filter: ((n03_001)::text ~~ '東京都'::text)
(11 rows)
```

st_expand()

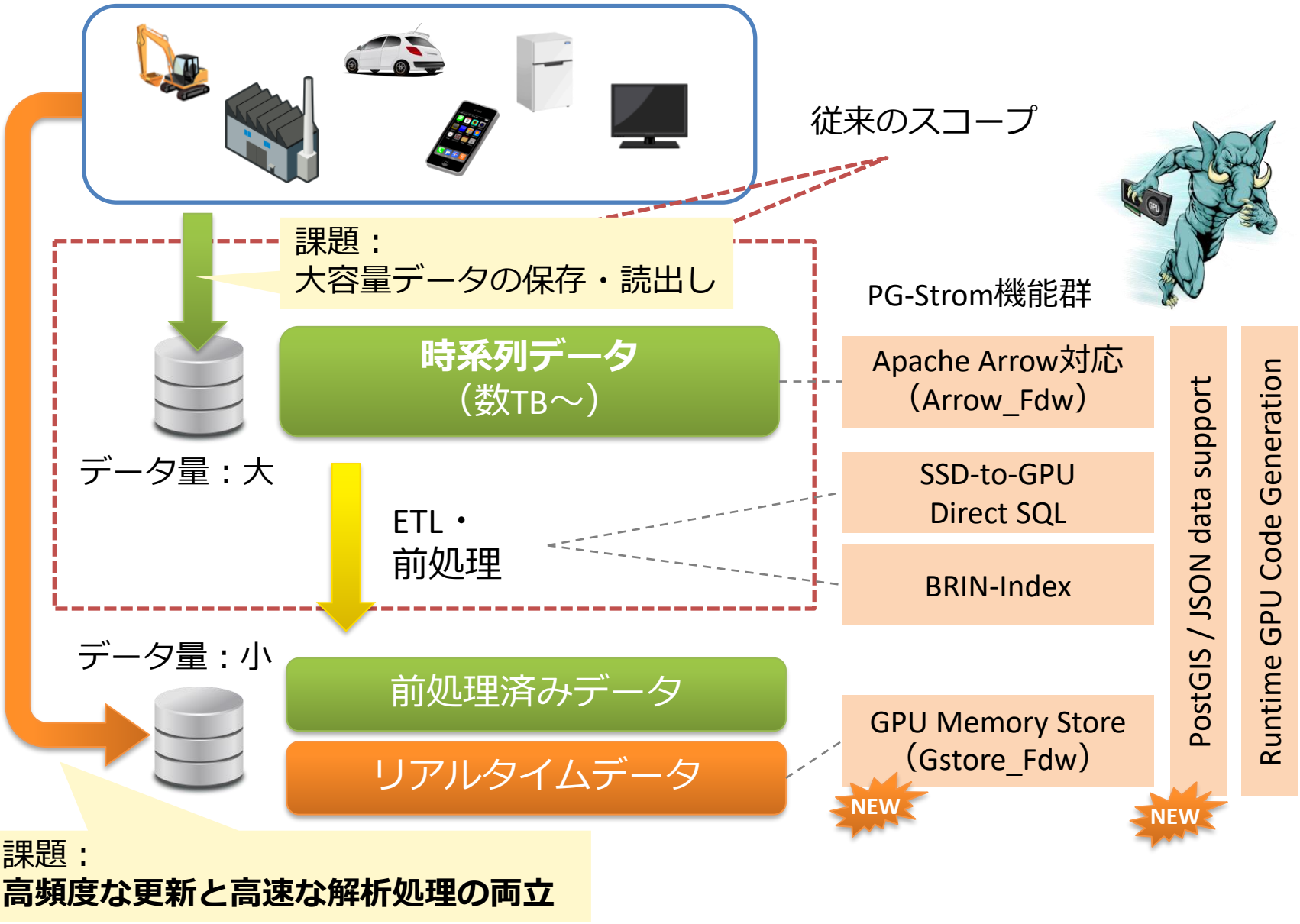
✓ 引数1(点)を上下左右に引数2(実数)だけ拡張した矩形領域を返す。

➔ これとGiSTインデックスのBounding-Boxが共通領域を持てば、st_dwithin()が真となる可能性がある。



リアルタイムデータ分析のための GPUメモリストア（Gstore_Fdw）

IoT/M2Mデータのライフサイクルと PG-Stromの機能



背景と課題

GPUとホストメモリは“遠い”

- 通常、GPUはPCI-Eバスを介してホストシステムと接続する。
- PCI-E Gen 3.0 x16レーンだと、片方向 16GB/s 「しかない」
  行って帰ってのレイテンシは数十マイクロ秒単位。
- DRAMの転送速度は140GB/sでレイテンシは100ns程度。
  もしL1/L2に載っていれば数ns単位でアクセス可能

  出展：https://gist.github.com/eshelman/343a1c46cb3fba142c1afdcdeec17646

高い更新頻度

- もし100万デバイスが10秒に一度、
  現在位置を更新するなら？

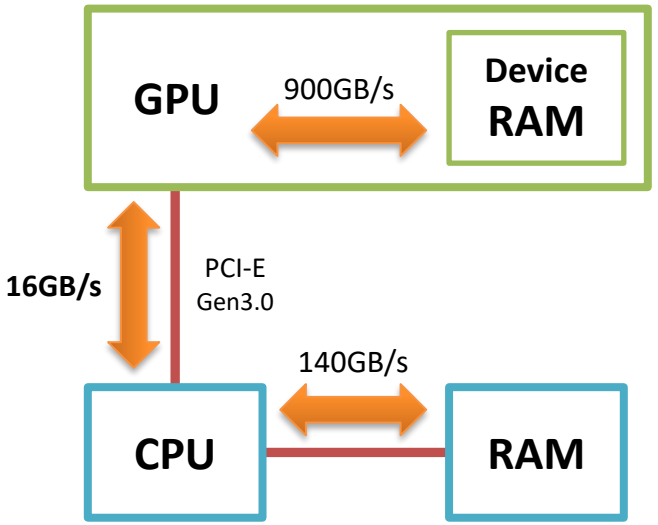
➔ 単純 UPDATE を毎秒10万回実行。

➔ 1.0sec / 10万 = 10us

リアルタイム性に対する要請

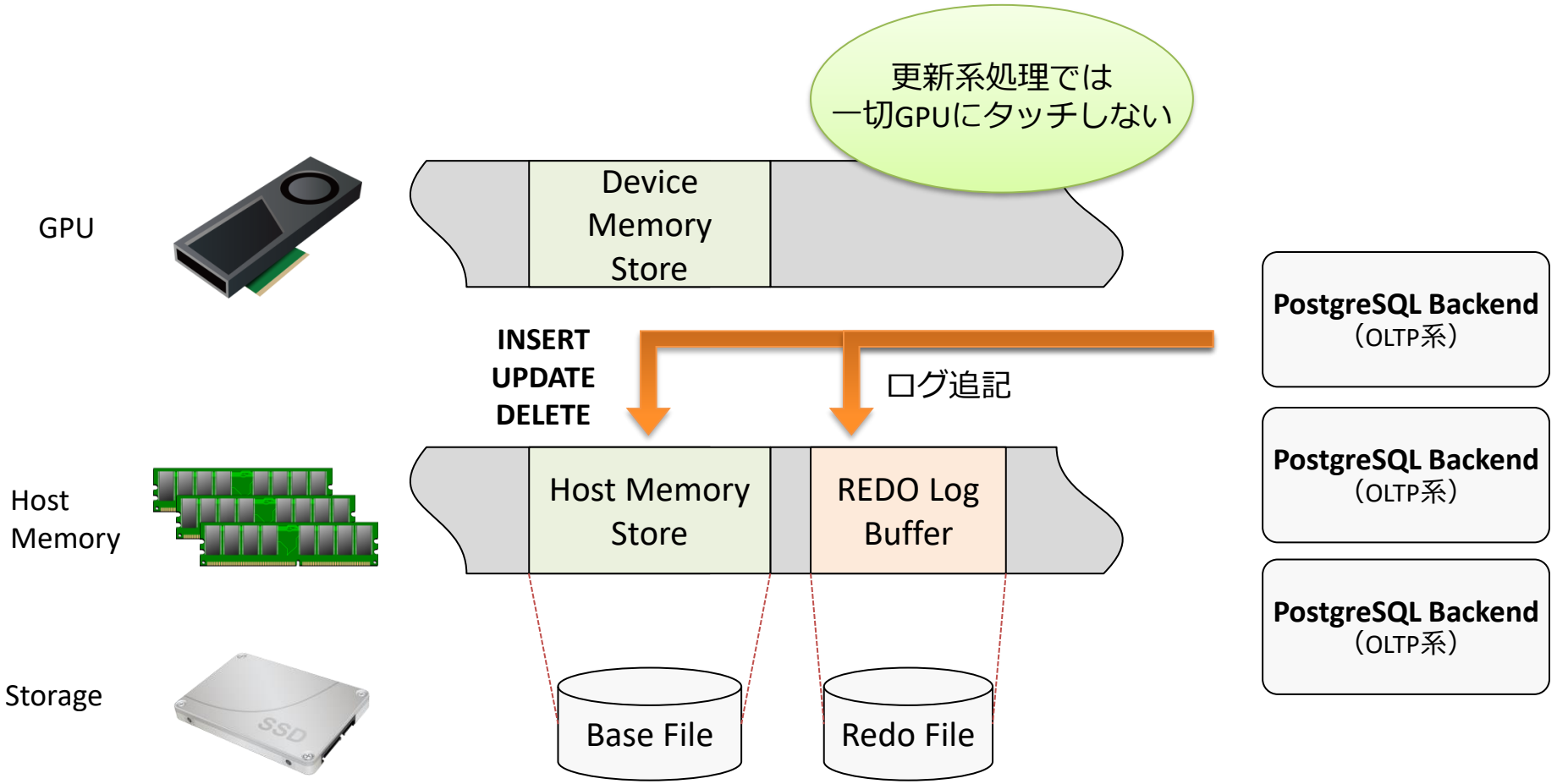
- 利用者が検索、解析したいのは、
  「その時点での最新の状態」

➔ 後でまとめてバッチ、では通用しない。



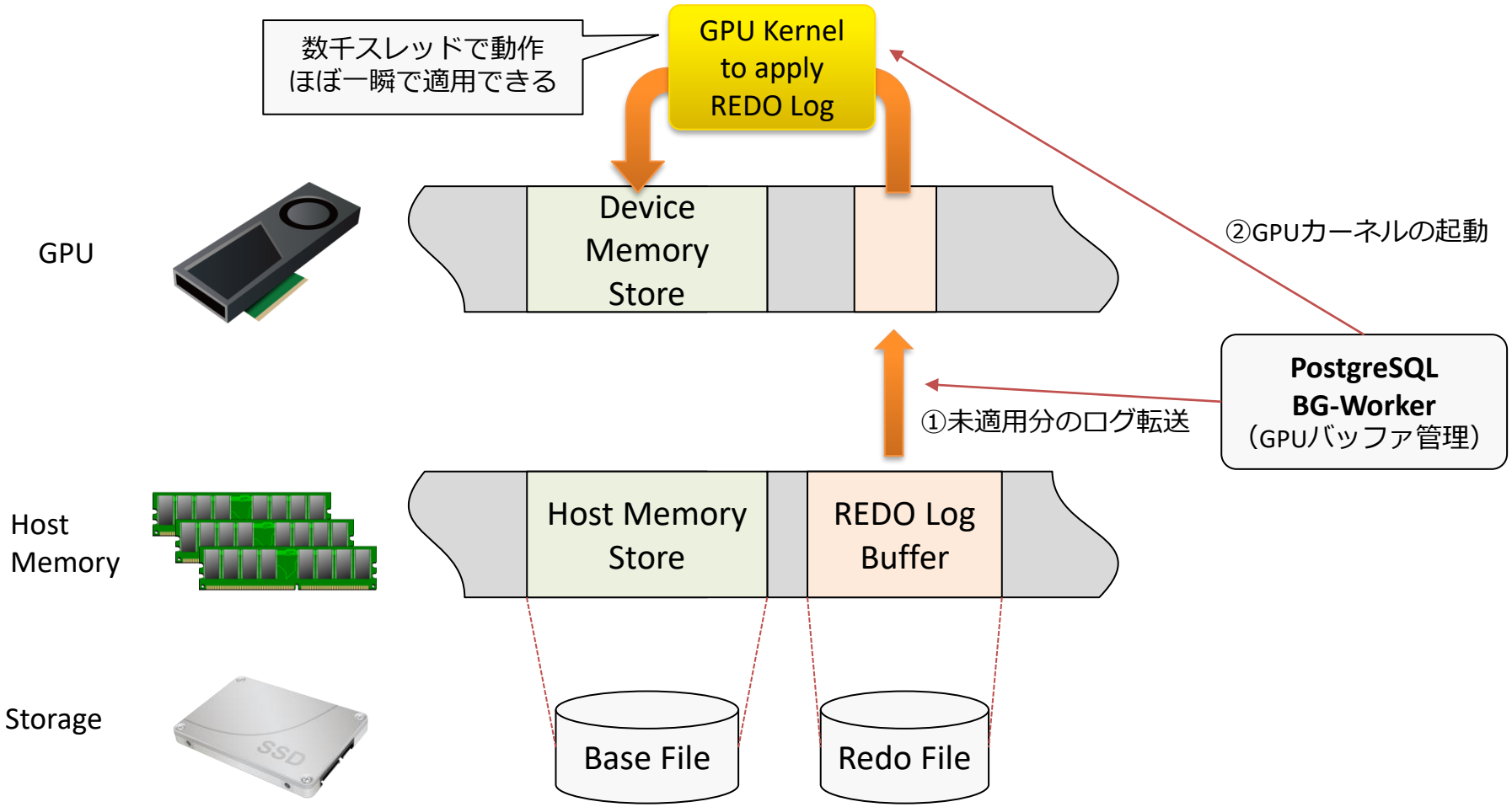
Gstore_Fdwアーキテクチャ

REDOログを利用してGPUメモリ側を非同期に更新する。
検索クエリの実行前に更新を適用すれば辻褄は合う！



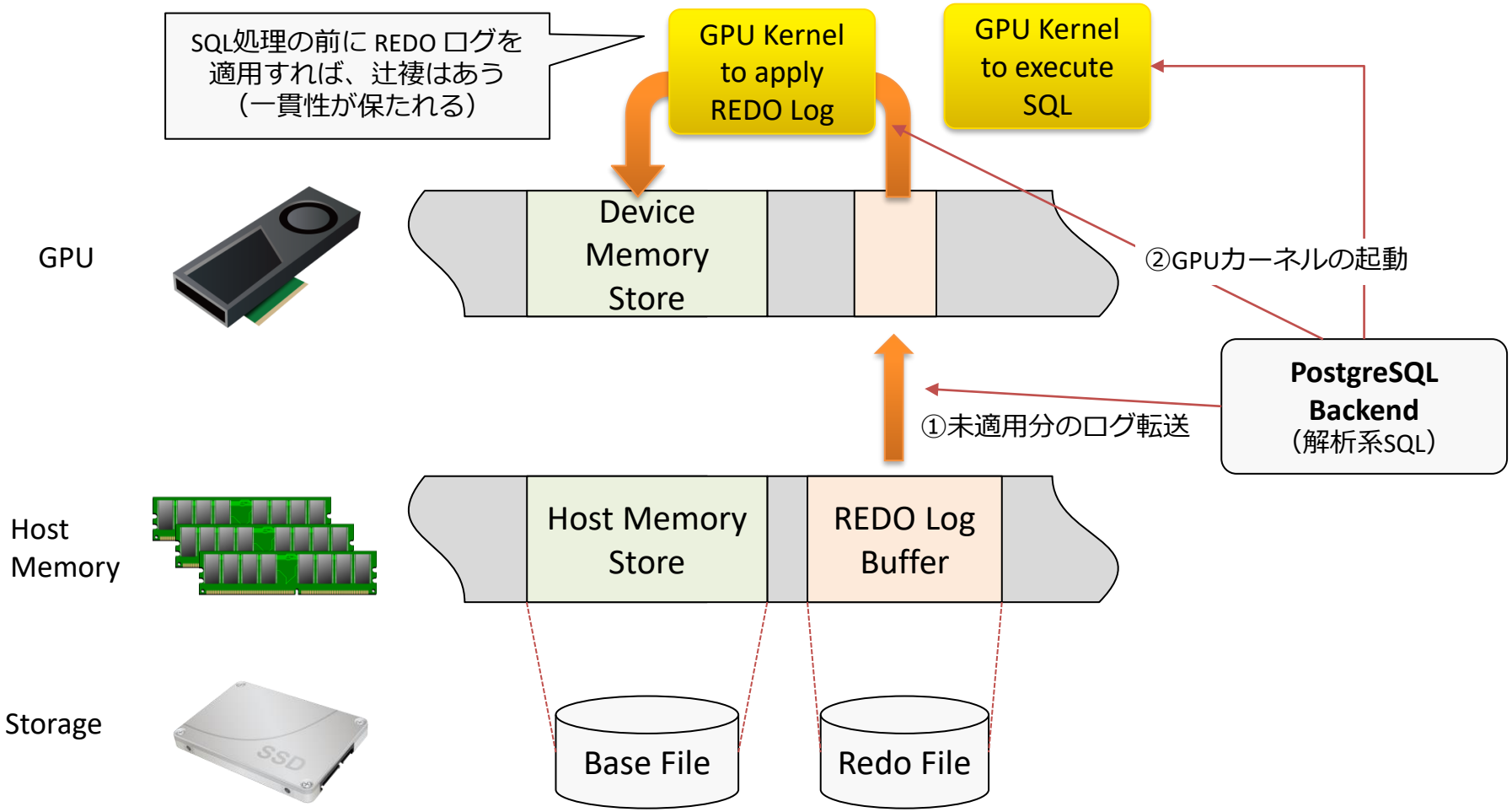
Gstore_Fdwアーキテクチャ

REDOログを利用してGPUメモリ側を非同期に更新する。
検索クエリの実行前に更新を適用すれば辻褄は合う！



Gstore_Fdwアーキテクチャ

REDOログを利用してGPUメモリ側を非同期に更新する。
検索クエリの実行前に更新を適用すれば辻褄は合う！



Gstore_Fdw外部テーブルの定義

```
CREATE FOREIGN TABLE ft (
  id int,
  x  float,
  y  float,
  z  text
) server gstore_fdw
  options (gpu_device_id '1',
           base_file     '/opt/pgdata-dev/ft.base',
           redo_log_file '/opt/pmem-dev/ft.redo’,
           max_num_rows  '4000000',
           primary_key   'id');
```

（補足）

- base_fileはNVMEストレージ上に配置するのが望ましい。
  - ✓ mmap(2)してバイト単位のランダムアクセスが多発するため、PMEMの微妙なアクセスの遅さが性能差として見えてしまう。
- redo_log_fileはPMEM上に配置するのが望ましい。
  - ✓ トランザクションのコミット時にREDOログを永続化する必要があるため。追記書き出しのみだが、細かい単位でCPUキャッシュをフラッシュする。（逆にブロックデバイスのようなfsyncは必要ないが）

更新処理のパフォーマンス（1/2）

```
--- 200万行を投入
INSERT INTO ft (SELECT x, 100*random(),
                          100*random(),
                          md5(x::text)
                  FROM generate_series(1,2000000) x);

--- 座標に見立てたx,yをランダムな値で更新
--- (pgbenchでクライアント数：20を想定)
UPDATE ft SET x = 100*random(),
              y = 100*random()
        WHERE id = (SELECT 20*(100000.0*random())::int + :client_id);

--- 実行計画
EXPLAIN ....;
                              QUERY PLAN
-----------------------------------------------------------------------
 Update on ft  (cost=0.02..100.03 rows=10000 width=58)
   InitPlan 1 (returns $0)
     ->  Result  (cost=0.00..0.02 rows=1 width=4)
   ->  Foreign Scan on ft  (cost=0.00..100.01 rows=10000 width=58)
         Filter: (id = $0)
         Index Cond: id = $0
(6 rows)
```

更新処理のパフォーマンス（2/2）

```
$ pgbench -p 6543 postgres -n -f test.sql -c 20 -j 10 -T 10
transaction type: test.sql
scaling factor: 1
query mode: simple
number of clients: 20
number of threads: 10
duration: 10 s
number of transactions actually processed: 1236401
latency average = 0.162 ms
tps = 123631.393482 (including connections establishing)
tps = 123674.819689 (excluding connections establishing)
```

（補足）

- まだ排他ロックで実装している箇所があり、クライアント数が20を越えた辺りでサチり始める。
  - ➔ Atomic演算による置き換えや、マルチGPU化による排他ロック自身の分散。
- PostgreSQLの通常テーブル（on NVME-SSD）：
  20クライアント 65k TPS、80クライアント160k TPS程度
- 通常テーブルと外部テーブルの内部ロジックの違いに起因する差異は未検証

検索処理のパフォーマンス（再掲）

```
--- GpuScan (GPU版PostGIS) + Gstore_Fdw
=# SELECT count(*) FROM ft
    WHERE st_contains('polygon ((10 10,90 10,90 12,12 12,12 88,90 88,90 90,¥
                                 10 90,10 10))', st_makepoint(x,y));
 count
-------
 94440
(1 row)

Time: 75.466 ms

--- 通常版PostGIS + PostgreSQLテーブル
=# SET pg_strom.enabled = off;
SET
=# SELECT count(*) FROM tt
    WHERE st_contains('polygon ((10 10,90 10,90 12,12 12,12 88,90 88,90 90,¥
                                 10 90,10 10))', st_makepoint(x,y));
 count
-------
 94440
(1 row)

Time: 332.300 ms
```

200万個のPointから、
指定領域内の数をカウント

Gstore_Fdw機能の今後

WIP

機能強化

- マルチGPU（マルチバッファ）への対応
  - ✓GPUバッファ容量の拡大と、分析時に複数台のGPUへ負荷を割り振る効果。
  - ✓更新時の排他ロックを分散させる効果。
- 継続的なソフトウェア品質改善

クラウド提供

- 従来のNVME-GPU連携フォーカスと異なり、
  GPUのみでシステムを構成できるためハードウェア要件が緩い。

モデル名	p3.2xlarge	p3.8xlarge	p3.16xlarge	NC6s v	NC12s v3	NC24s v3
vCPU	8 core	32 core	64 core	6 core	12 core	24 core
RAM	61GB	244GB	488GB	112 GB	224 GB	448 GB
GPU	1 x Tesla V100 (16GB; 5120C)	4 x Tesla V100 (16GB; 5120C)	8 x Tesla V100 (16GB; 5120C)	1 x Tesla V100 (16GB; 5120C)	2 x Tesla V100 (16GB; 5120C)	4 x Tesla V100 (16GB; 5120C)
オンデマンド料金	4.194USD/h	16.776USD/h	32.552USD/h	¥469.73/h	¥939.46/h	¥2066.85/h

※ オンデマンド料金は、各社東京リージョンの本体のみ単価（2020/11/11現在）で記載。

amazon web services

Microsoft Azure

GPU版PostGIS（GiSTインデックス）を用いた実ワークロードの性能評価

位置データアプリケーションを用いた評価

ユーザ毎・都道府県ごとに、訪問した（近傍のチェックイン履歴のある）駅の割合を導出し、都道府県ごとの駅制覇マップを作成する。

検索における工夫

近傍に位置する駅は両方訪問した事にする。

集計クエリの構造

（ユーザ, 都道府県）ごとに、訪問した駅の数を集計

```
TRUNCATE train_visit_summary;
INSERT INTO train_visit_summary (
       -- 集計結果を集計テーブルに保持しておき、Web画面の表示時に
       -- 未訪問の都道府県も含めて出力する。
 WITH cte_station_visit AS (
  SELECT DISTINCT s.pref_cd, s.station_cd, u.uid
         -- “訪問”した駅コード、その都道府県コード、ユーザIDを
         -- 重複なしで取り出す
    FROM train_station_list s, train_user_station u
         -- 駅の位置と、ユーザのチェックイン情報を突合する
   WHERE st_dwithin(st_makepoint(s.lon, s.lat),
                    st_makepoint(u.lon, u.lat), 0.004)
         -- 駅の位置とユーザのチェックイン情報の位置が一定範囲内に存在すれば、
         -- その駅を訪問したものとみなす。
     AND s.station_cd = s.station_g_cd
 )
  SELECT v.pref_cd, v.uid, count(*)
    INTO train_visit_summary
    FROM cte_station_visit v
   GROUP BY 1, 2
);
```

実行結果（CPU）

```
                                                                                 QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Insert on train_visit_summary (actual time=564337.937..564337.937 rows=0 loops=1)
  ->  Subquery Scan on "*SELECT*" (actual time=561347.255..564263.657 rows=222985 loops=1)
     ->  GroupAggregate (actual time=561347.253..564245.853 rows=222985 loops=1)
         Group Key: v.pref_cd, v.uid
        ->  Sort (actual time=561347.227..562318.377 rows=18041063 loops=1)
            Sort Key: v.pref_cd, v.uid
            Sort Method: quicksort  Memory: 1632107kB
           ->  Subquery Scan on v (actual time=534684.752..543837.130 rows=18041063 loops=1)
              ->  Group (actual time=534684.748..542789.954 rows=18041063 loops=1)
                  Group Key: s.pref_cd, s.station_cd, u.uid
                 ->  Sort (actual time=534684.737..538315.419 rows=31188357 loops=1)
                     Sort Key: s.pref_cd, s.station_cd, u.uid
                     Sort Method: quicksort  Memory: 2248387kB
                    ->  Nested Loop (actual time=0.072..485222.366 rows=31188357 loops=1)
                       ->  Seq Scan on train_user_station u
                                   (actual time=0.010..2436.186 rows=23969240 loops=1)
                       ->  Index Scan using train_station_list_st_makepoint_idx on train_station_list s
                                   (actual time=0.018..0.019 rows=1 loops=23969240)
                           Index Cond: (st_makepoint(lon, lat) &&
                                        st_expand(st_makepoint(u.lon, u.lat), 0.004))
                           Filter: ((station_cd = station_g_cd) AND
                                    st_dwithin(st_makepoint(lon, lat),
                                               st_makepoint(u.lon, u.lat), 0.004))
                           Rows Removed by Filter: 1
 Planning Time: 0.241 ms
 Execution Time: 564369.525 ms
(21 rows)
```

実行結果（GPU）

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 Insert on train_visit_summary (actual time=29994.448..29994.448 rows=0 loops=1)
   ->  Subquery Scan on "*SELECT*" (actual time=29873.410..29925.354 rows=222985 loops=1)
         ->  HashAggregate (actual time=29873.407..29911.557 rows=222985 loops=1)
               Group Key: s.pref_cd, u.uid
               ->  HashAggregate (actual time=22546.625..26156.456 rows=18041063 loops=1)
                     Group Key: s.pref_cd, s.station_cd, u.uid
                     ->  Custom Scan (GpuJoin) on train_user_station u
                                 (actual time=614.179..14651.453 rows=31188357 loops=1)
                           Outer Scan: train_user_station u
                                 (actual time=49.108..2245.314 rows=23969240 loops=1)
                           Depth 1: GpuGiSTJoin
                                    HeapSize: 992.73KB (estimated: 13.12KB), IndexSize: 928.62KB
                                    IndexFilter: (st_makepoint(s.lon, s.lat) &&
                                                  st_expand(st_makepoint(u.lon, u.lat), 0.004))
                                                         on train_station_list_st_makepoint_idx
                                    Rows Fetched by Index: 33137684
                                    JoinQuals: st_dwithin(st_makepoint(s.lon, s.lat),
                                                          st_makepoint(u.lon, u.lat), 0.004)
                           GPU Preference: GPU0 (Tesla V100-PCIE-16GB)
                           ->  Seq Scan on train_station_list s
                                       (actual time=0.020..2.591 rows=12542 loops=1)
                                 Filter: (station_cd = station_g_cd)
                                 Rows Removed by Filter: 2534
 Planning Time: 0.513 ms
 Execution Time: 30087.473 ms
(19 rows)
```

まとめ（1/2）

GPU版PostGIS

- PostGIS関数をGPU上で並列実行するための機能拡張。
- GiSTインデックスにも対応し、H/Wの特性を活かしたインデックス探索を実装。

GPUメモリストア

- リアルタイムのデータ更新と分析ワークロードを両立するための機能。
- GPUにデータを常駐させ、PMEMを利用してトランザクションを永続化。

想定用途

- 携帯電話や自動車の“位置情報“と、エリア定義情報との突合。
- リアルタイムの広告配信やプッシュ型イベント通知など。
- これら位置情報分析を含む “計算ヘビー“ なワークロードを
  手元のワークステーションやクラウドで手軽に実行できるように。

リソース

- GitHub: https://github.com/heterodb/pg-strom
- Document: http://heterodb.github.io/pg-strom/ja/
- Contact: Tw: @kkaigai / ✉ kaigai@heterodb.com

まとめ（2/2）

PG-Stromプロジェクトは皆さんのご参加を歓迎します

リソース

- GitHub: https://github.com/heterodb/pg-strom
- Document: http://heterodb.github.io/pg-strom/ja/
- Contact: Tw: @kkaigai / ✉ kaigai@heterodb.com


HeteroDB