http://www.slideshare.net/kaigai/pgconfasia2016-plcuda

PL/CUDA

~Fusion of HPC Grade Power with In-Database Analytics~

The PG-Strom Project / NEC OSS Promotion Center
KaiGai Kohei <kaigai@ak.jp.nec.com>

自己紹介

KaiGai Kohei

- tw: @kkaigai
- https://github.com/kaigai

PostgreSQL

- SELinux, FDW, CustomScan, ...

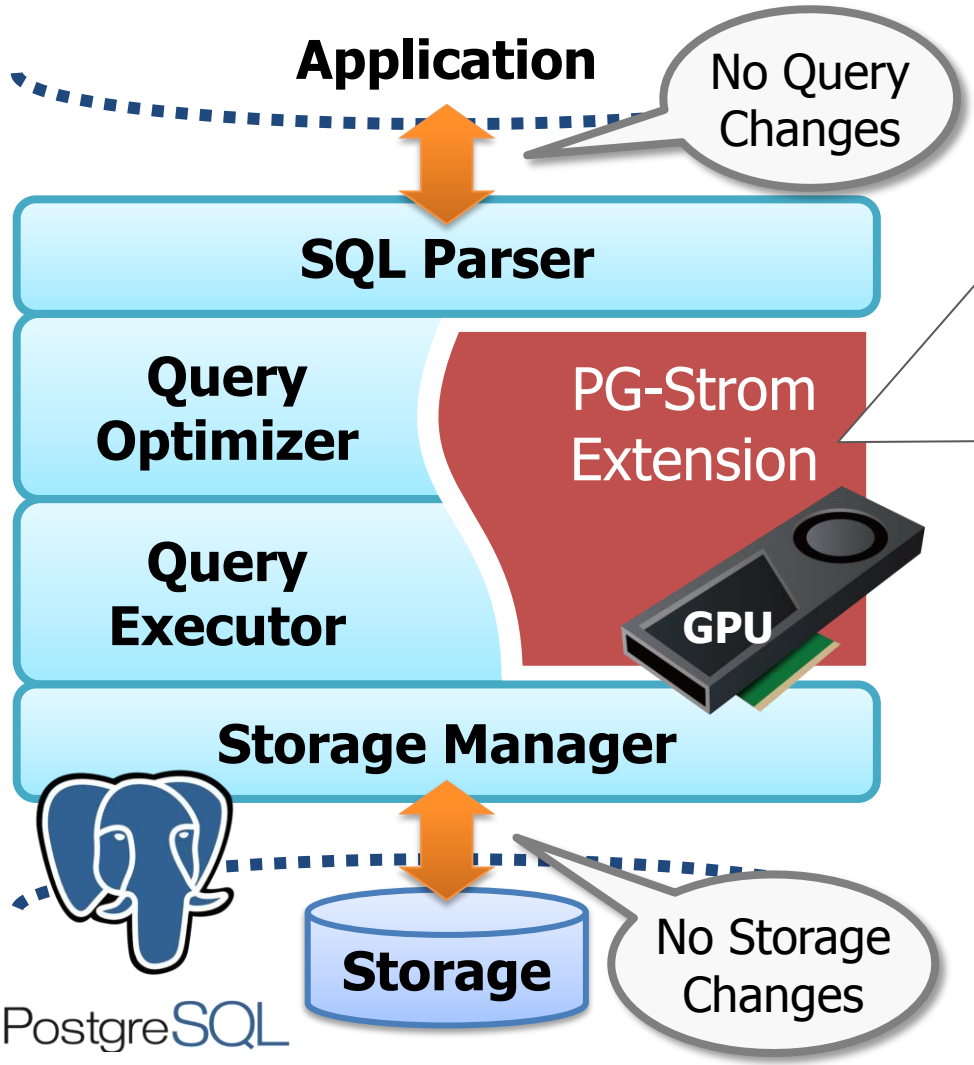
PG-Strom

- GPUを用いたPostgreSQL向け高速化モジュールの作者

お仕事

- NEC OSS推進センタ
- SW開発／ビジネス開拓

PG-Strom概要 (1/2) – アーキテクチャ



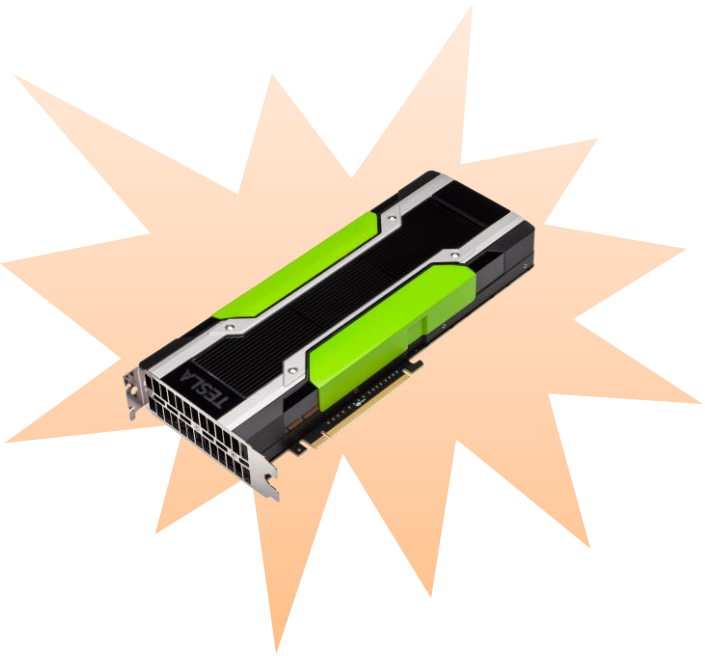
機能

- SQLからGPUコードを自動生成。
- GPUによる非同期/超並列実行
- WHERE句、JOIN、GROUP BY、Projectionに対応

利点

- 数千演算コアを用いた透過的なアクセラレーション
- 解析系ワークロードに対する低コストのソリューション

GPU (Graphic Processor Unit) の特徴



	GPU	CPU
Model	NVIDIA Tesla P100	Intel Xeon E5-2699v4
Architecture	Pascal	Broadwell
Launch	Q2-2016	Q1-2016
# of transistors	15billion	7.2billion
# of cores	**3584 (simple)**	22 (functional)
core clock	1.126GHz ~1.303GHz	2.20GHz ~3.60GHz
Perk FFLOPS (FP32)	**9.3 TFLOPS**	1.2 TFLOPS (with AVX2)
DRAM Size	16GB (HBM2)	max 1.5TB (DDR4)
Memory Band	**732GB/s**	76.8GB/s
Power Consumption	250W	145W

PG-Strom概要 (2/2) – GPUバイナリの自動生成

```
QUERY:  SELECT cat, count(*), avg(x) FROM t0
        WHERE x between y and y + 20.0 GROUP BY cat;
```

例）WHERE句での計算式を
CUDAプログラムに変換。

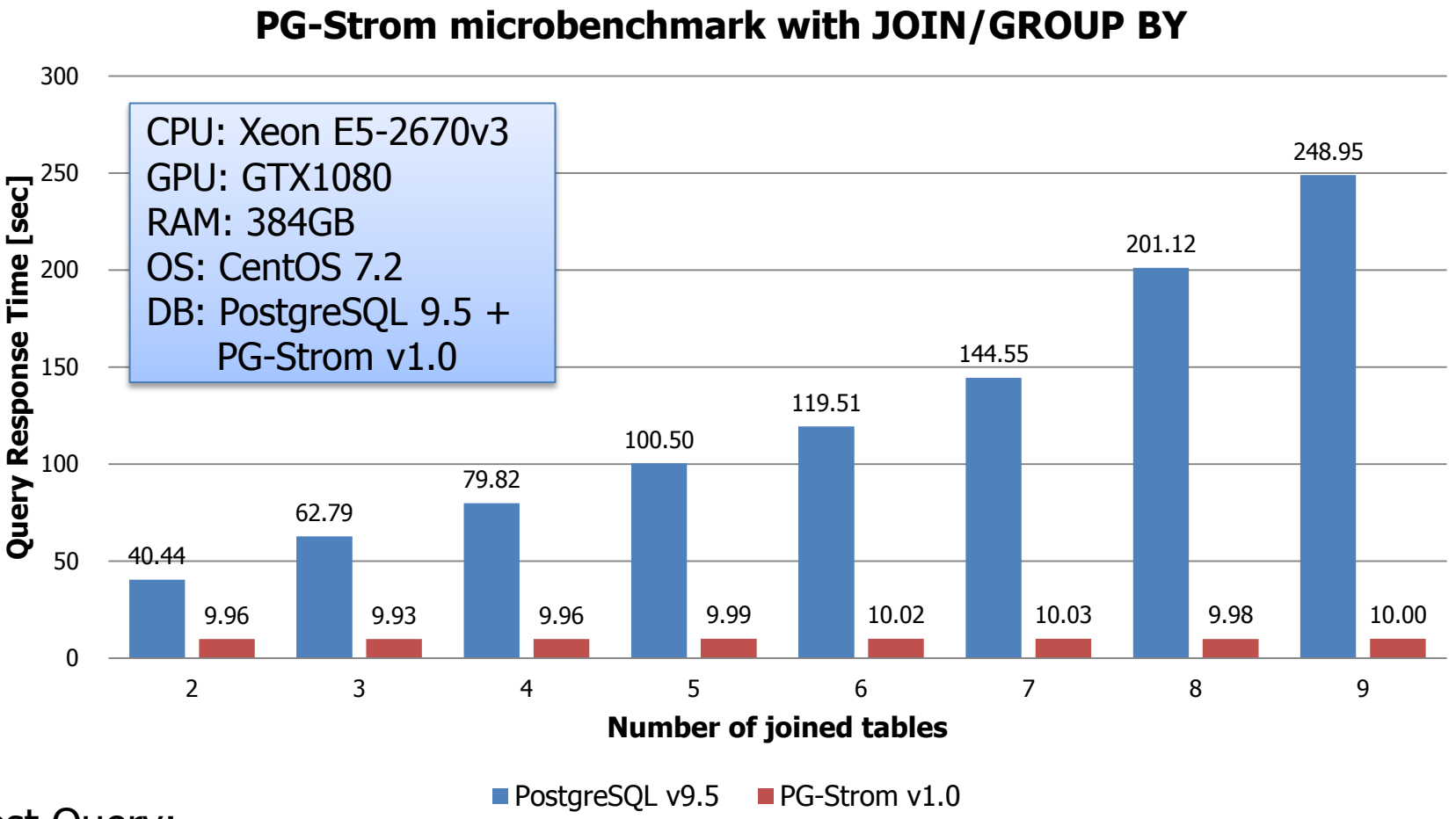
```
        :
STATIC_FUNCTION(bool)
gpupreagg_qual_eval(kern_context *kcxt,
                    kern_data_store *kds,
                    size_t kds_index)
{                                   Reference to input data
  pg_float8_t KPARAM_1 = pg_float8_param(kcxt,1);
  pg_float8_t KVAR_3 = pg_float8_vref(kds,kcxt,2,kds_index);
  pg_float8_t KVAR_4 = pg_float8_vref(kds,kcxt,3,kds_index);

  return EVAL((pgfn_float8ge(kcxt, KVAR_3, KVAR_4) &&
               pgfn_float8le(kcxt, KVAR_3,
                 pgfn_float8pl(kcxt, KVAR_4, KPARAM_1))));
}       :                   SQL expression in CUDA source code
```

Just-in-time Compile

Run-time Compiler (nvrtc)

Parallel Execution

GPUによるSQL実行高速化の一例

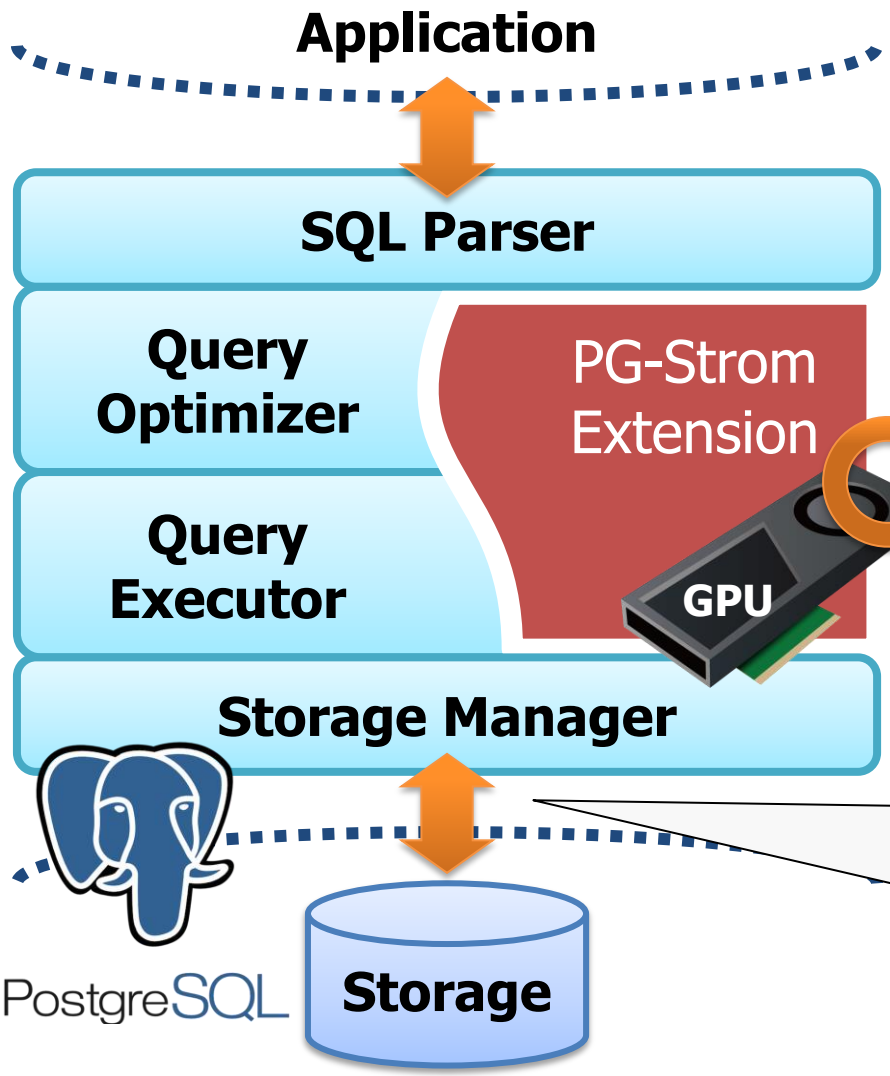


- Test Query:

```
SELECT cat, count(*), avg(x)
  FROM t0 NATURAL JOIN t1 [NATURAL JOIN t2 ...]
 GROUP BY cat;
```

✓ t0 contains 100M rows, t1...t8 contains 100K rows (like a start schema)

v1.0開発過程におけるユーザからのフィードバック



計算集約的ワークロード

- In-database Analytics
- 科学技術計算、マーケティング

➔ by PL/CUDA + Matrix-Array

I/O集約的ワークロード

- 汎用の大規模OLAP
- ETL、レポーティング

➔ by SSD-to-GPU P2P DMA

Introduction of PL/CUDA

失敗から学ぶ (1/3) – アルゴリズムをSQLで記述する

Reference) Query for MAX-MIN method

Apr-2016

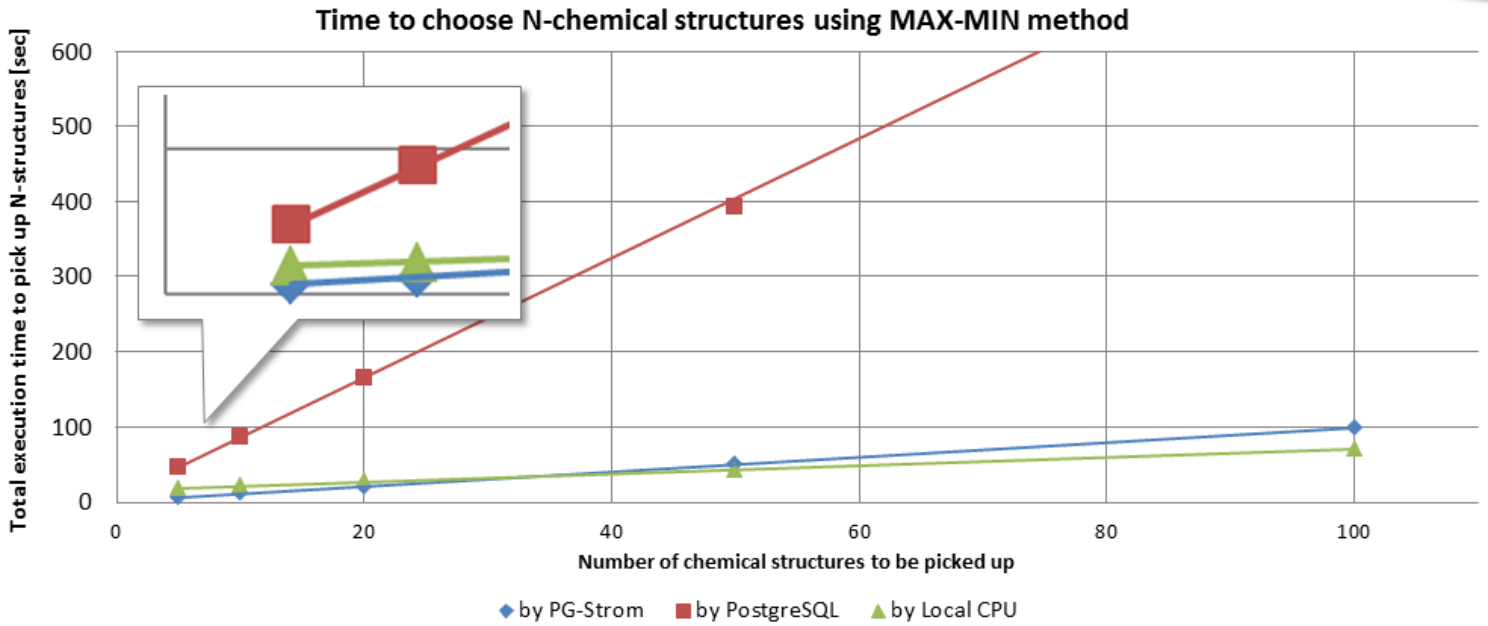
```
WITH next_item AS (
  INSERT INTO pg_temp.subset_table (
    SELECT r.*
      FROM mqn r,
           (SELECT id FROM pg_temp.dist_table
                      ORDER BY dist DESC LIMIT 1) d
     WHERE r.id = d.id)
 RETURNING *
)
SELECT r.id, LEAST(d.dist, sqrt((r.c1 - n.c1)^2 +
                                (r.c2 - n.c2)^2 +
                                      :
                                (r.c41 - n.c41)^2 +
                                (r.c42 - n.c42)^2)) dist
  INTO pg_temp.dist_table_new
  FROM pg_temp.dist_table d,
       next_item n, mqn r
 WHERE r.id = d.id
```

Hot point of the SQL query

失敗から学ぶ (2/3) – 性能上のアドバンテージ (?)

Benchmark Results – MAX-MIN Method

Apr-2016



Summary

- 10M rows, 211MB in total
- workload characteristics: $W_{I/O} \ll W_{CPU}$
- If no GPU support, calculation in RDBMS cannot be an option.
- PG-Strom recorded similar performance with download + R-script.

失敗から学ぶ (3/3) – 問題点

問題① – SQLでアルゴリズムを記述する人なんていない。

- そもそも、アルゴリズムの大半は手続き型言語を前提として開発されている。
- ユーザがCUDAでアルゴリズムの核を記述する必要がないとしても、これは権局、SQLのパズルを考えながらアルゴリズムのロジックを記述しているのと同じ事。

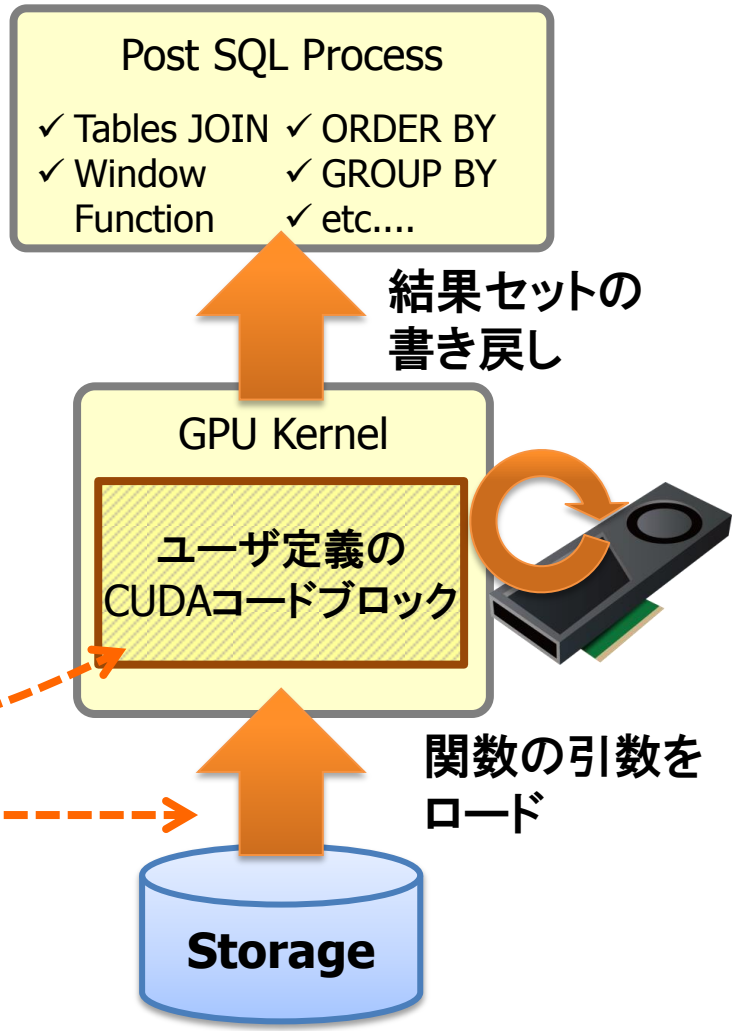
問題② – 性能上のメリットは本当にあった？

- Min-Max法の距離計算において、確かにPG-Stromの実行性能はPostgreSQLを遥かに上回っているが、そもそも、この種の計算をPostgreSQLで行っている人っていないんじゃないの？
- GpuProjectionの性能は、このアルゴリズムを処理するために設計されたCPU版の外部アプリの性能と概ね同等だった。なぜ？
  - ✓ SQL互換性に由来する非効率性
  - ✓ PostgreSQLの行フォーマットに由来する非効率性

解決策 – PL/CUDA + Array-Matrix

PL/CUDA
手動での最適化手段

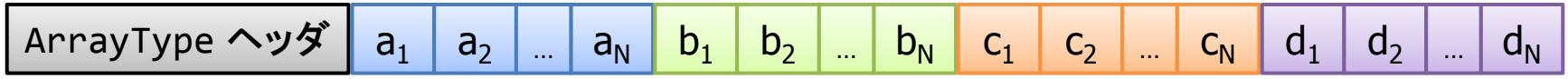
```
CREATE FUNCTION my_logic(matrix, matrix)
RETURNS vector
AS $$
  ユーザ定義のCUDAコードブロック
$$ LANGUAGE 'plcuda';
```



Array-Matrix
非NULLの2次元配列を"行列"と解釈

$$\begin{bmatrix} a_1 & \cdots & d_1 \\ \vdots & \ddots & \vdots \\ a_N & \cdots & d_N \end{bmatrix}$$

4列N行 行列

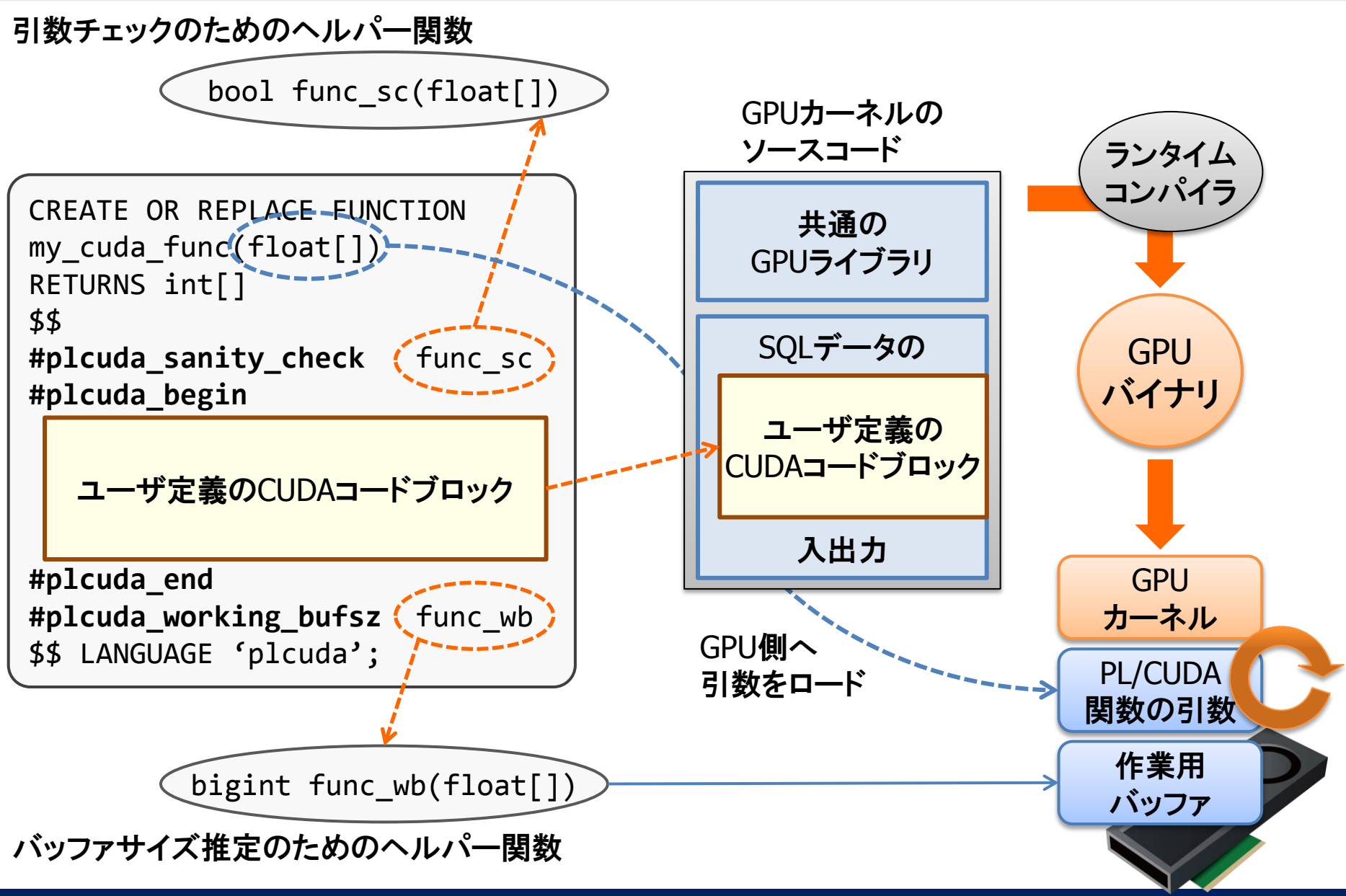


PL/CUDA関数定義の例

```
CREATE OR REPLACE FUNCTION
knn_gpu_similarity(int, int[], int[])
RETURNS float4[]
AS $$
#plcuda_begin
cl_int      k = arg1.value;
MatrixType *Q = (MatrixType *) arg2.value;
MatrixType *D = (MatrixType *) arg3.value;
MatrixType *R = (MatrixType *) results;
     :
nloops = (ARRAY_MATRIX_HEIGHT(Q) + (part_sz - k - 1)) / (part_sz - k);
for (loop=0; loop < nloops; loop++) {
    /* 1. calculation of the similarity */
    for (i = get_local_id(); i < part_sz * part_nums; i += get_local_size()) {
        j = i % part_sz;    /* index within partition */
        /* index of database matrix (D) */
        dindex = part_nums * get_global_index() + (i / part_sz);
        /* index of query matrix (Q) */
        qindex = loop * (part_sz - k) + (j - k);
        values[i] = knn_similarity_compute(D, dindex, Q, qindex);
    }
}
    :
#plcuda_end
$$ LANGUAGE 'plcuda';
```

CUDA code block

どのようにGPUカーネルを起動するか



自動生成のGPUコードがベスト性能を出せない理由

```
select x*y from c_test;
```

```
STATIC_FUNCTION(pg_float4_t)
pgfn_float4mul(kern_context *kcxt, pg_float4_t arg1, pg_float4_t arg2)
{
    pg_float4_t result;

    result.isnull = arg1.isnull | arg2.isnull;
    if (!result.isnull)
    {
        result.value = arg1.value * arg2.value;
        CHECKFLOATVAL(&kcxt->e, result,
                      isinf(arg1.value) || isinf(arg2.value),
                      arg1.value == 0.0 || arg2.value == 0.0);
    }
    return result;
}
```

- ✓ 各変数を参照するたびにNULLチェックが必要
- ✓ 個々の四則演算の度にオーバフローチェックが必要
- ✓ プリミティブな演算を関数呼び出しで実現せざるを得ない

データ形式に起因する非効率さ (1/2)

行形式(Row-oriented)データ

× 参照されない値も含む

× データアクセスに複数回のメモリ参照

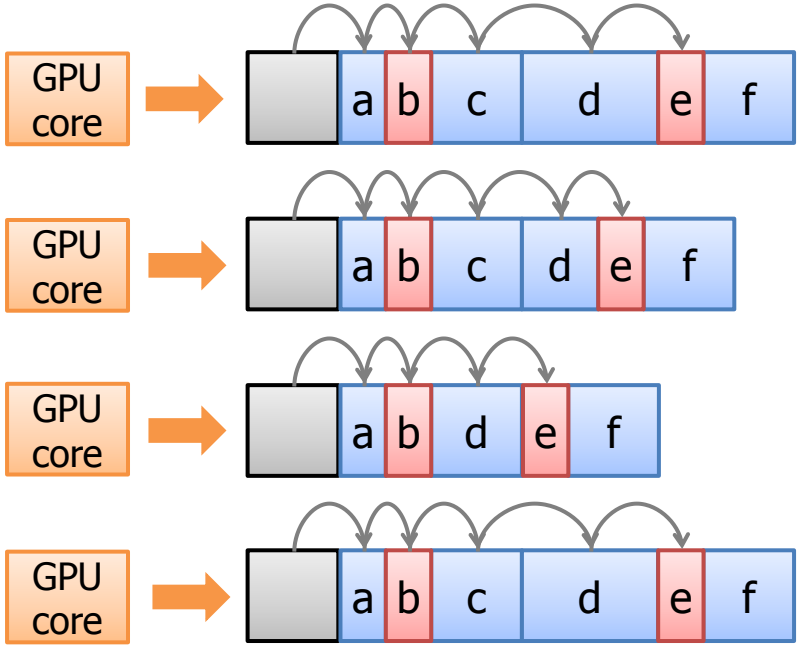
○ PostgreSQLにおける標準のデータ構造

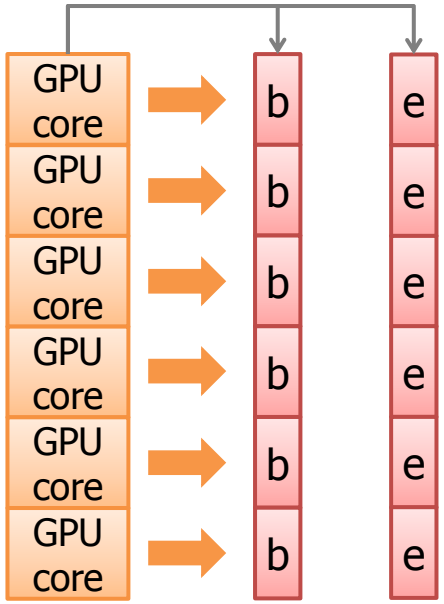
列形式(Column-oriented)データ

○ 参照される列のみをロードできる

○ *O(1)*でデータを参照できる

× データ形式の変換が必要



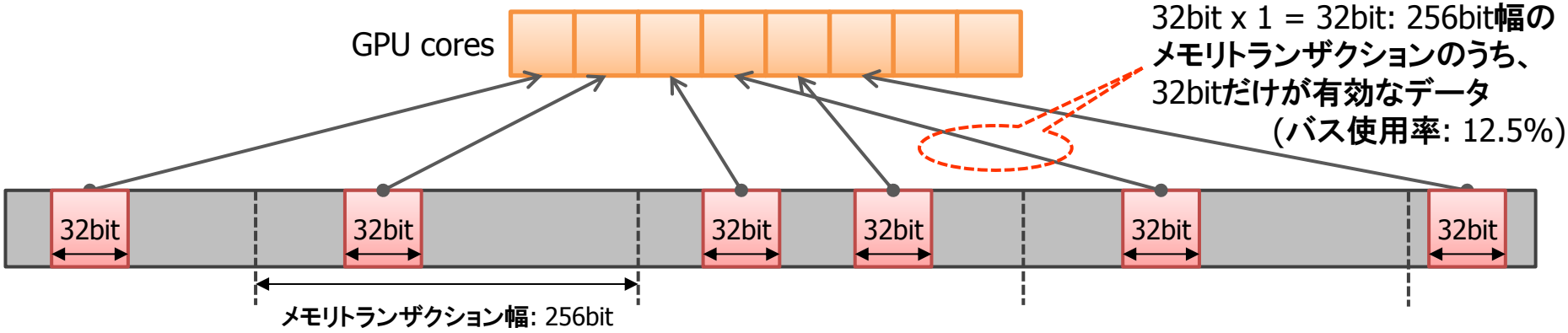


➔ 通常のSQLワークロード程度の負荷では、さすがにデータ形式変換のコストを正当化できないが、高度なアルゴリズム処理となると話は変わってくる。

データ形式に起因する非効率さ (2/2)

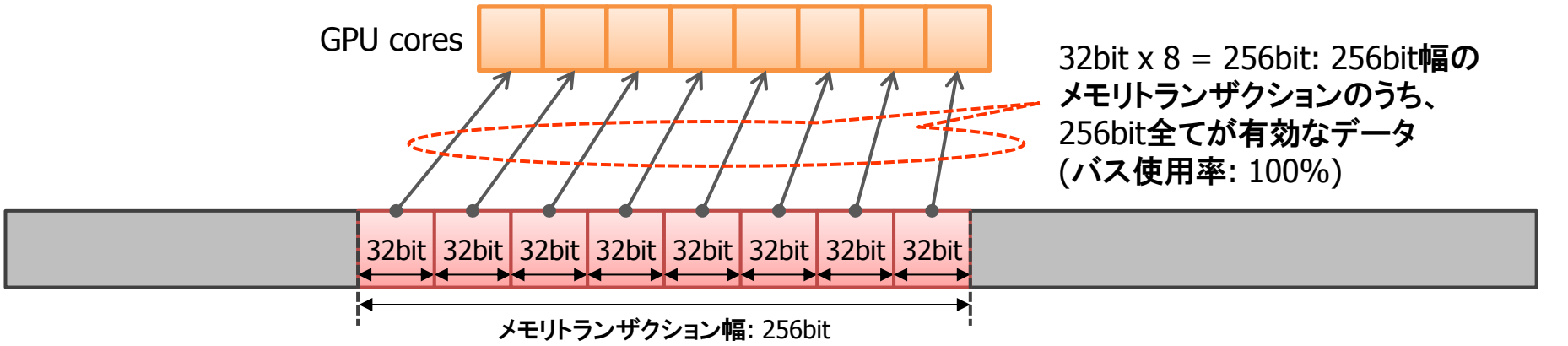
ランダムメモリアクセス (random memory access)

➔ メモリトランザクション回数が増える一方、メモリバスの使用率は決して高くない。



コアレスメモリアクセス (coalesced memory access)

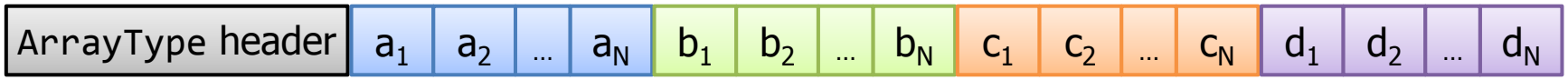
➔ 最小限のメモリトランザクションで、メモリバスの使用率を最大化する事ができる。



行列表現としての二次元配列

$$\begin{bmatrix} a_1 & \cdots & d_1 \\ \vdots & \ddots & \vdots \\ a_N & \cdots & d_N \end{bmatrix}$$ 4列N行行列

Array-Matrix
非NULL固定長の二次元配列を“行列”と見なす



datatype[] **array_matrix**(variadic *datatype*[])

- 入力データを蓄積し二次元配列として出力する集約関数
- *datatype*はint2, int4, int8, float4 および float8 のどれか。
- この二次元配列はNULLを含んでいてはならない。

SETOF record **matrix_unnest**(*datatype*[])

- m列n行の二次元配列を、m列から成るn個のレコードへと展開する関数。

課題

- 可変長データを扱う事ができない。
- PostgreSQLの可変長データの制約により、最大でも1GBになってしまう。

PL/CUDA関数の呼び出し例

```
SELECT row_number() OVER (),
       float4_as_int4(R.key_id) key_id,
       R.score
  FROM matrix_unnest(
         (SELECT my_plcuda_function(A.matrix,
                                    B.matrix)
            FROM (SELECT cbind(array_matrix(id),
                               array_matrix(x, y, z)) matrix
                    FROM normal_table
                   WHERE tag LIKE '%abc%') A,
                 (SELECT matrix
                    FROM matrix_table) B
         )
       ) AS R(key_id real, score real)
 ORDER BY score DESC
 LIMIT 1000;
```

SQLによる後処理
(JOIN, Window関数)

2つのMatrix-like Arrayを
引数に持つ
PL/CUDA関数の呼出し

Matrix-like Arrayを
N個のレコードに再展開

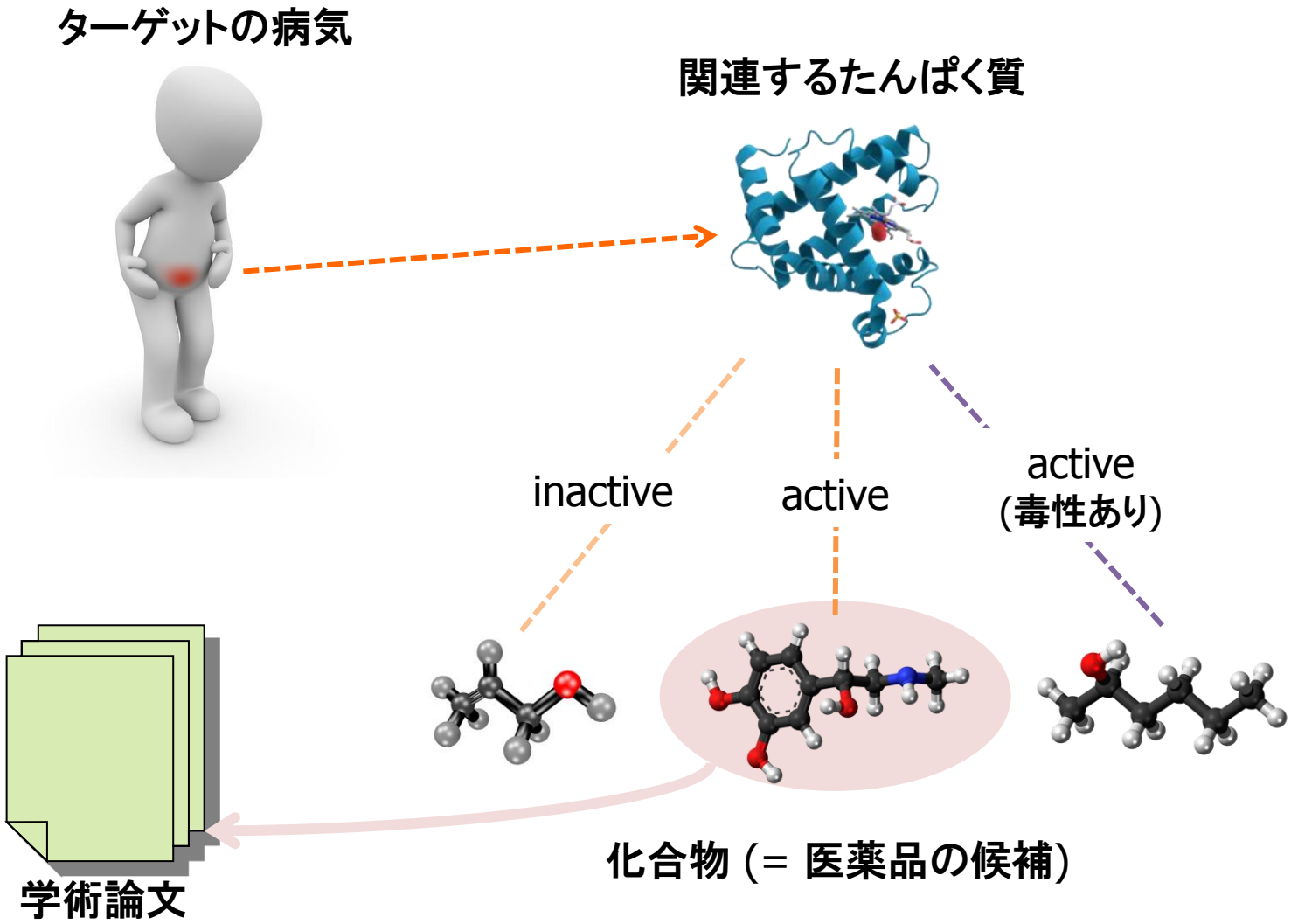
Matrix-like Arrayの生成、
または構築済みのものをロード

Case Study

創薬における類似度サーチ

背景 – 病気と候補化合物の関係

ターゲットたんぱく質に “active” である化合物の探索

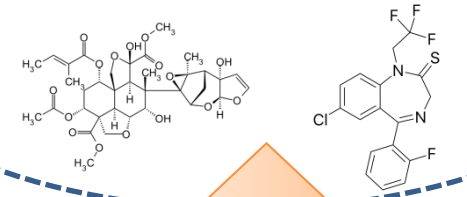


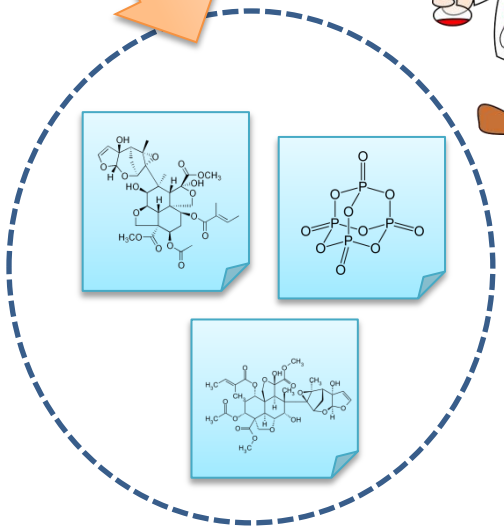
k-NN法による類似化合物サーチ (1/2)

ターゲットたんぱく質

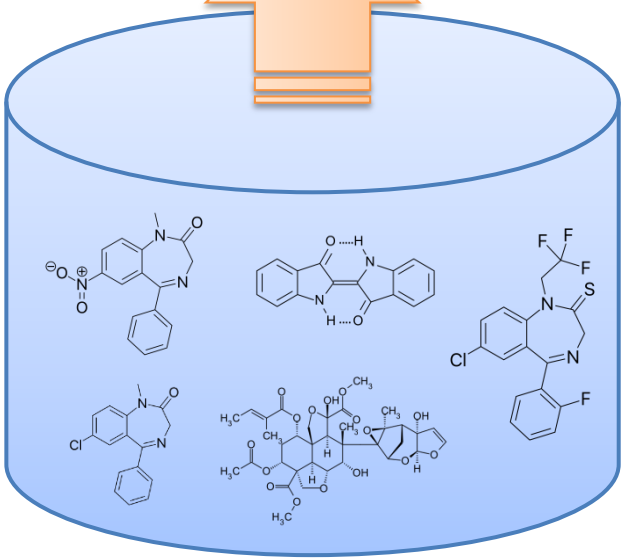
学術論文等から
ターゲットたんぱく質に
"active" である化合物
をピックアップ

"似た特性の化合物"は
"active"である可能性も高いはず。





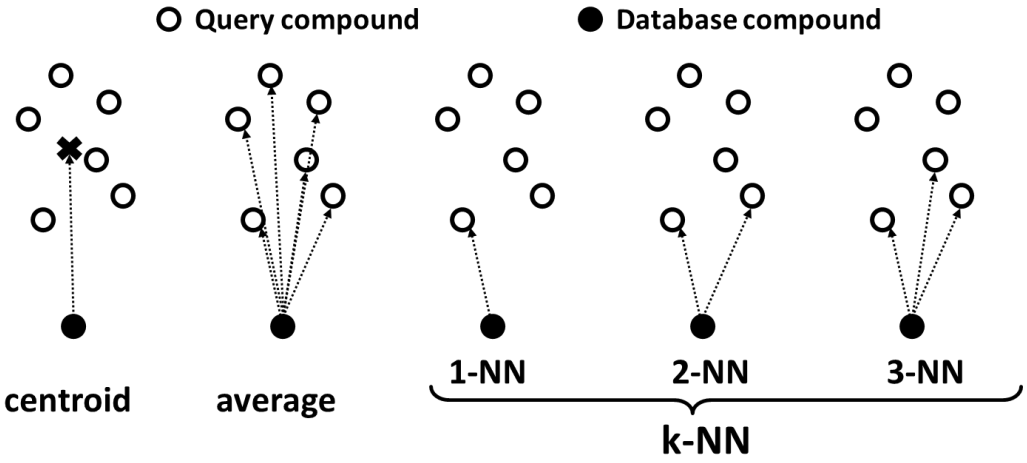
類似度による
サーチ



クエリ化合物群
(**Q**; ~1000件程度)

データベース化合物群
(**D**; ~1000万件程度)

k-NN法による類似化合物サーチ (2/2)



Definition of distance on similarity search using multiple reference molecules

類似度 = 距離の定義

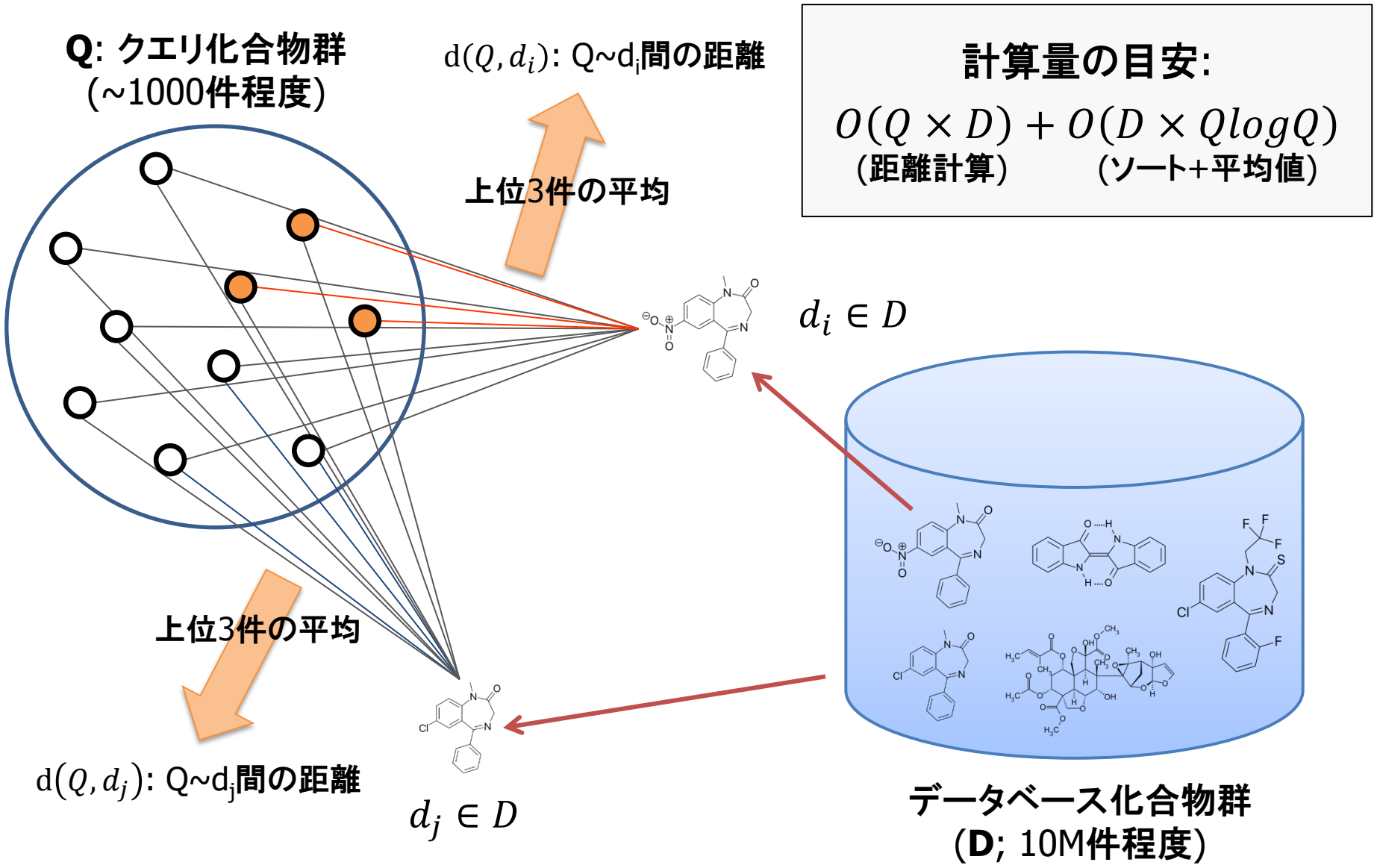
化合物のデータ構造

ID	NAME	Fingerprint (1024bit)
1	CHEMBL153534	0000000000010000001000000000000010000000000001000000...
2	CHEMBL405398	0000000000000001001000000000000000000000000000100000...
3	CHEMBL503634	0000010000000000000000000010000001000000000000000000...
:	:	:

Tanimoto Indexによる類似度の定義:

$$Similarity(A,B) = |A_{FP} \cap B_{FP}| / |A_{FP} \cup B_{FP}|$$

必要な計算量



計算量の目安:

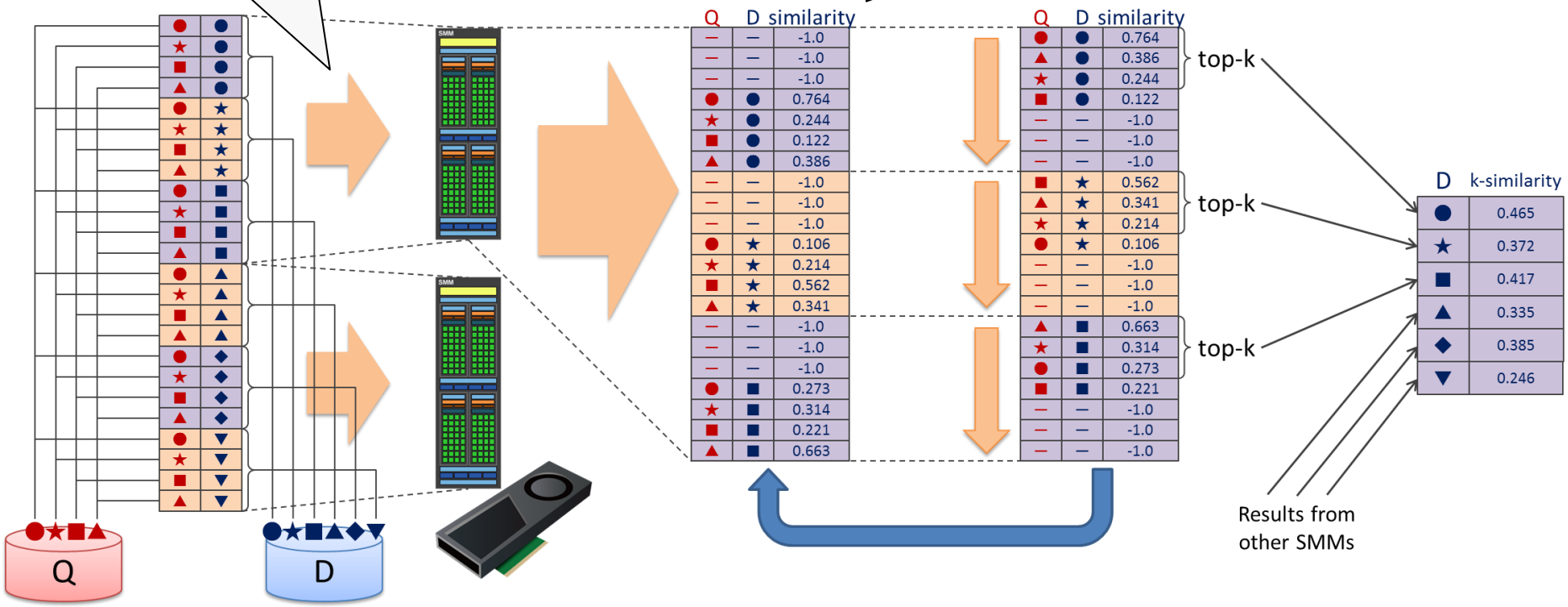
$$O(Q \times D) + O(D \times QlogQ)$$

(距離計算)　(ソート+平均値)

PL/CUDA関数の実装 (1/3)

Step-1
全てのQ×Dの論理的な組み合わせを、
複数のパーティションに分割。これらを
GPUの実行ユニットであるSMMに割り当て。

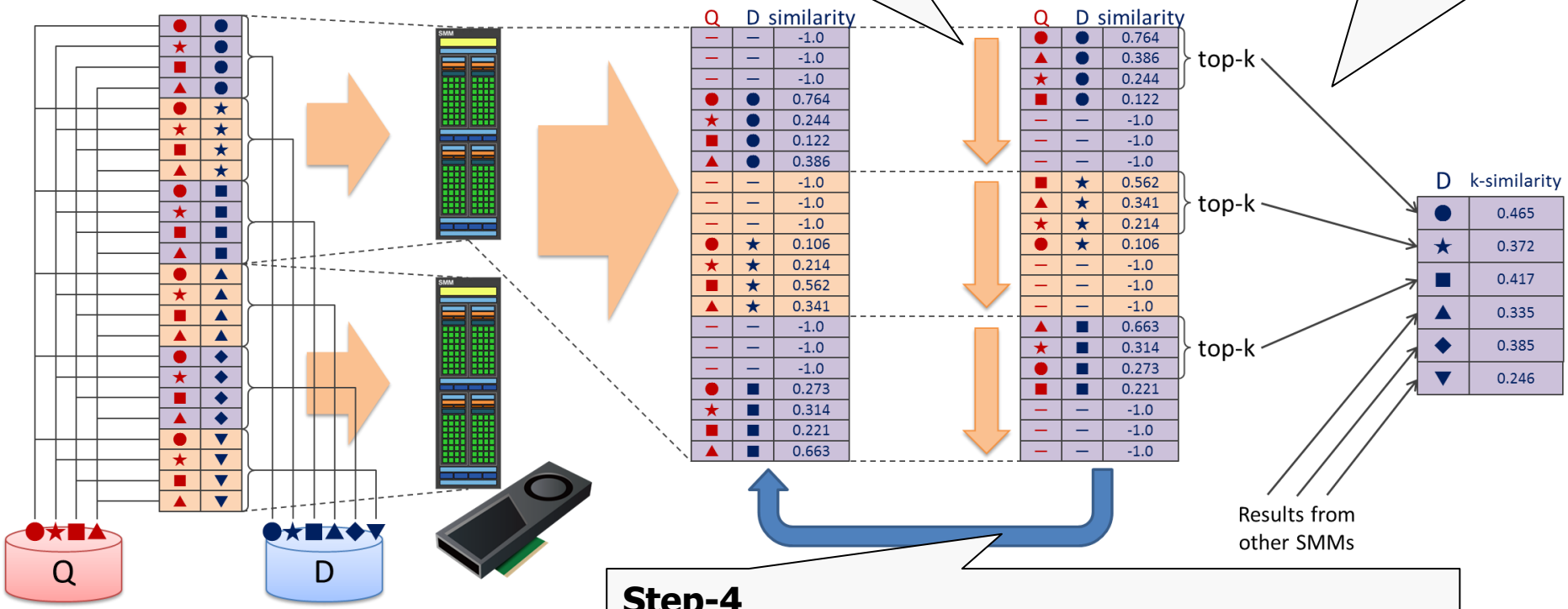
Step-2
各GPUコアがQ化合物群とd化合物間の
類似度スコアを算出。
L1キャッシュと同等のレイテンシでアクセス
可能な〝共有メモリ〟にこれを格納



PL/CUDA関数の実装 (2/3)

Step-3
類似度スコアによる Bitonic-Sorting を実行。
ある d 化合物に対する q 化合物が、類似度の順に並ぶことになる。

Step-5
類似度上位k件による平均値を算出し、結果バッファに格納。



Step-4
Q化合物群のサイズが共有メモリよりも巨大な場合、
Step-2以降を繰り返し。

PL/CUDA関数の実装 (3/3)

```
real[]                              -- ID+Similarity of D化合物 (2xN)
knn_gpu_similarity(int   k,         -- k-value
                   int[] Q,         -- ID+Fingerprint of Q化合物 (33xM)
                   int[] D);        -- ID+Fingerprint of D化合物 (33xN)
```

```
CREATE OR REPLACE FUNCTION
knn_gpu_similarity(int,    -- k-value
                   int[],  -- ID+bitmap of Q
                   int[])  -- ID+bitmap of D
RETURNS float4[]           -- result: ID+similarity
AS $$
#plcuda_decl
      :
#plcuda_begin
#plcuda_kernel_blocksz   ¥
    knn_gpu_similarity_main_block_size
#plcuda_num_threads      ¥
    knn_gpu_similarity_main_num_threads
#plcuda_shmem_blocksz    8192
cl_int      k = arg1.value;
MatrixType *Q = (MatrixType *) arg2.value;
MatrixType *D = (MatrixType *) arg3.value;
MatrixType *R = (MatrixType *) results;
      :
for (loop=0; loop < nloops; loop++)
{
  /* 1. calculation of the similarity */
  for (i = get_local_id();
       i < part_sz * part_nums;
       i += get_local_size()) {
    j = i % part_sz;    /* index within partition */
    dindex = part_nums * get_global_index()
           + (i / part_sz);
    qindex = loop * (part_sz - k) + (j - k);
    if (dindex < ARRAY_MATRIX_HEIGHT(D) &&
        qindex < ARRAY_MATRIX_HEIGHT(Q)) {
      values[i] = knn_similarity_compute(D, dindex,
                                         Q, qindex);
    }
  }
  __syncthreads();

  /* 2. sorting by the similarity for each partition */
  knn_similarity_sorting(values, part_sz, part_nums);
  __syncthreads();
      :
}
#plcuda_end
#plcuda_sanity_check    knn_gpu_similarity_sanity_check
#plcuda_working_bufsz   0
#plcuda_results_bufsz   knn_gpu_similarity_results_bufsz
$$ LANGUAGE 'plcuda';
```

PL/CUDA関数の呼出し

```
PREPARE knn_sim_rand_10m_gpu_v2(int)     -- arg1:@k-value
AS
SELECT  row_number() OVER (),
        fp.name,
        similarity
  FROM (SELECT float4_as_int4(key_id) key_id, similarity
          FROM matrix_unnest(
                 (SELECT rbind( knn_gpu_similarity($1,Q.matrix,
                                                   D.matrix))
                    FROM (SELECT cbind(array_matrix(id),
                                       array_matrix(bitmap)) matrix
                            FROM finger_print_query) Q,
                         (SELECT matrix
                            FROM finger_print_10m_matrix) D
                 )
          ) AS sim(key_id real, similarity real)
        ORDER BY similarity DESC) sim,
       finger_print_10m fp
 WHERE fp.id = sim.key_id
 LIMIT 1000;
```

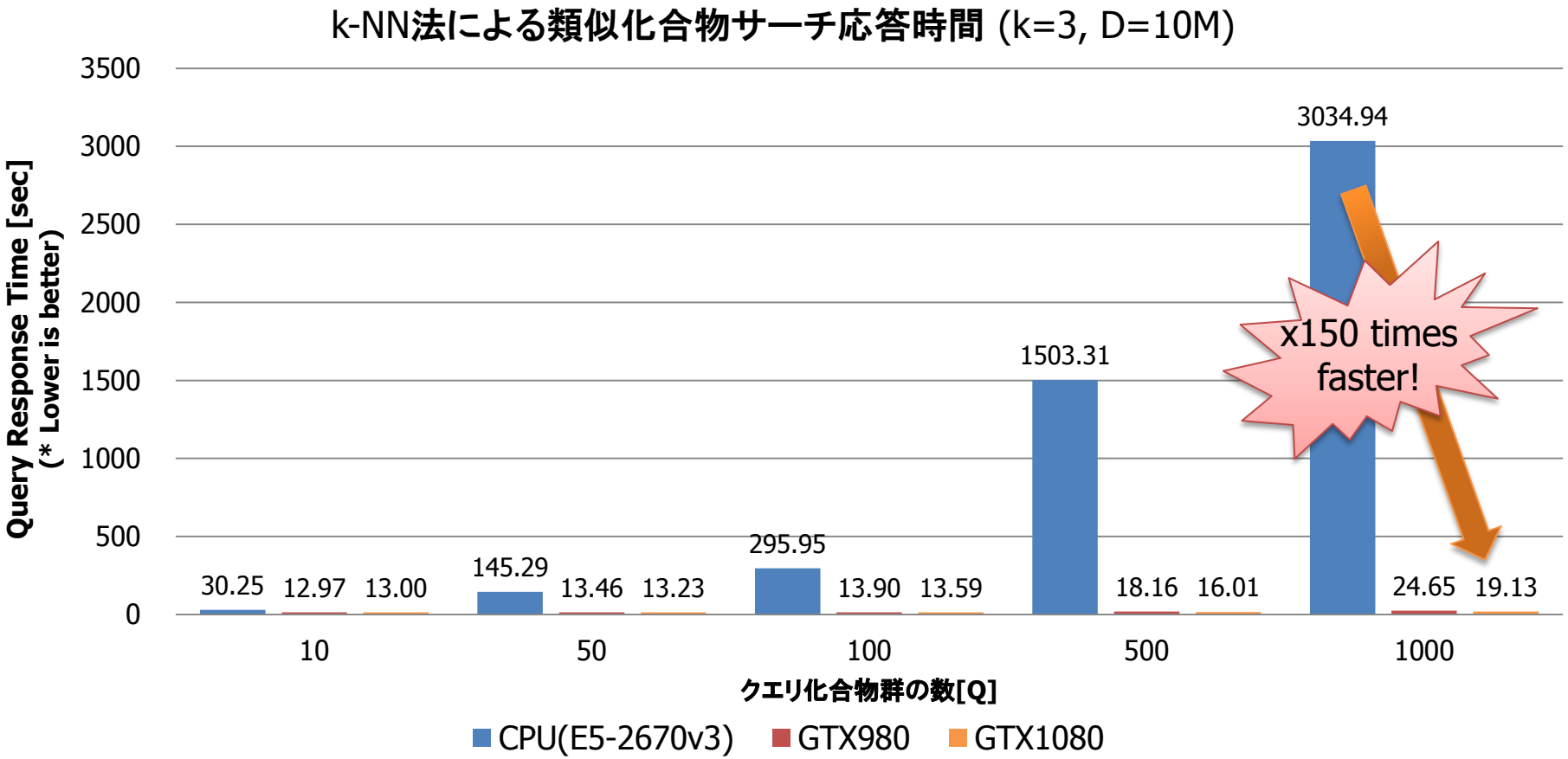
SQLによる後処理
- 他のテーブルとJOINして化合物ID➔化合物名を変換
- window関数による類似度順位の計算

Q行列/D行列を引数にとるPL/CUDA関数の呼出し。

PL/CUDA関数の返り値である3xNのArray-Matrixを展開し、通常の3列N行のレコードへ変換

テーブルから読み出したレコードをArray-Matrixに変換（または、事前にビルド）

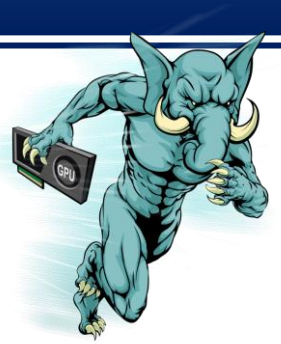
パフォーマンス



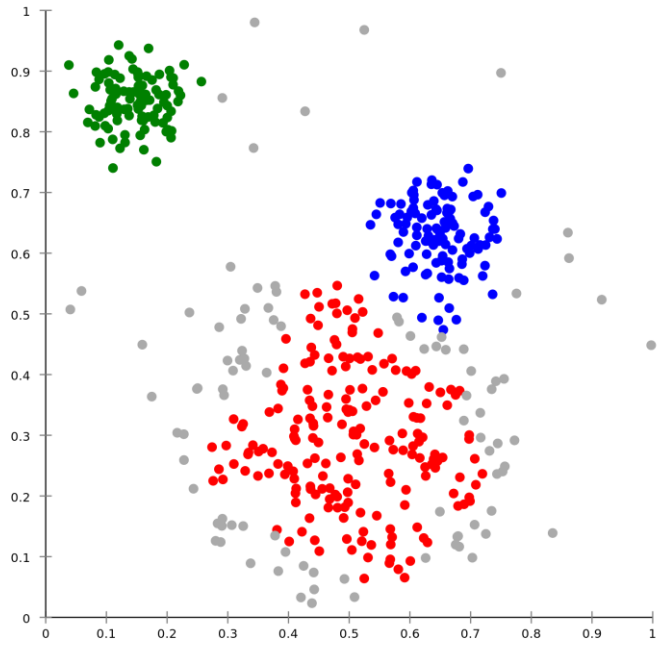
- CPU版は、同等のロジックをC言語によるバイナリ版で実装して比較計測
- D化合物群の数は1000万レコード、Q化合物群の数は10,50,100,500,1000個の5通り
  ➔ 最大で100億通りの組合せを計算。これは実際の創薬ワークロードの規模と同等。
- HW) CPU: Xeon E5-2670v3, GPU: GTX980 / GTX1080, RAM:384GB
- SW) CentOS7, CUDA8.0, PostgreSQL v9.5 + PG-Strom v1.0

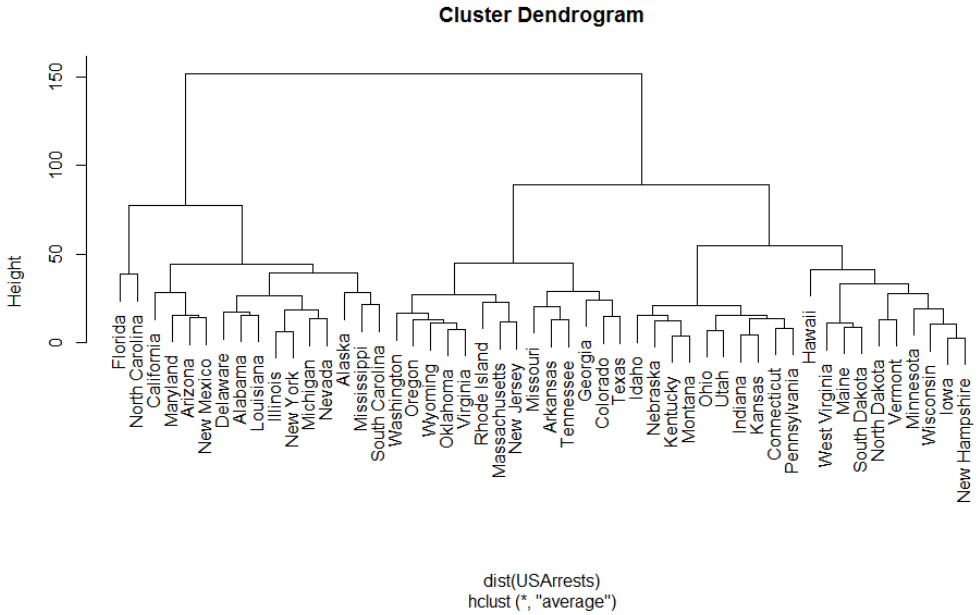
Another Usage

In-databaseでk-meansクラスタリング



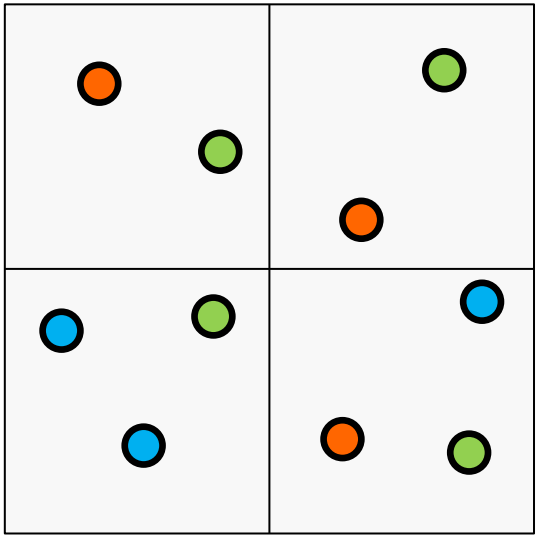
クラスター分析


0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

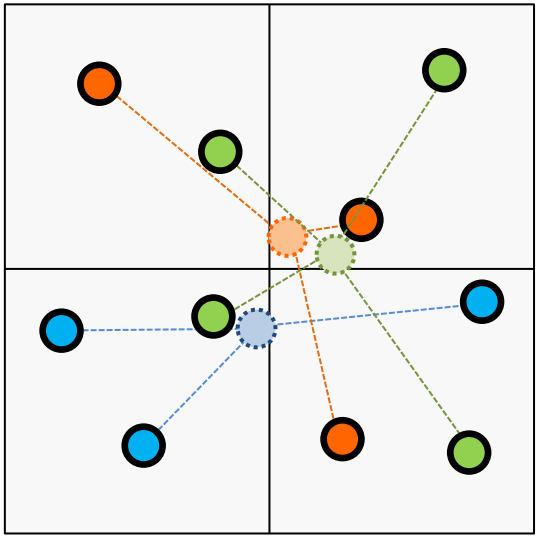

Cluster Dendrogram
Height
0
50
100
150
Florida
North Carolina
California
Maryland
Arizona
New Mexico
Delaware
Alabama
Louisiana
Illinois
New York
Michigan
Nevada
Alaska
Mississippi
South Carolina
Washington
Oregon
Wyoming
Oklahoma
Virginia
Rhode Island
Massachusetts
New Jersey
Missouri
Arkansas
Tennessee
Georgia
Colorado
Texas
Idaho
Nebraska
Kentucky
Montana
Ohio
Utah
Indiana
Kansas
Connecticut
Pennsylvania
Hawaii
West Virginia
Maine
South Dakota
North Dakota
Vermont
Minnesota
Wisconsin
Iowa
New Hampshire
dist(USArrests)
hclust (*, "average")

k-meansクラスタリング アルゴリズム

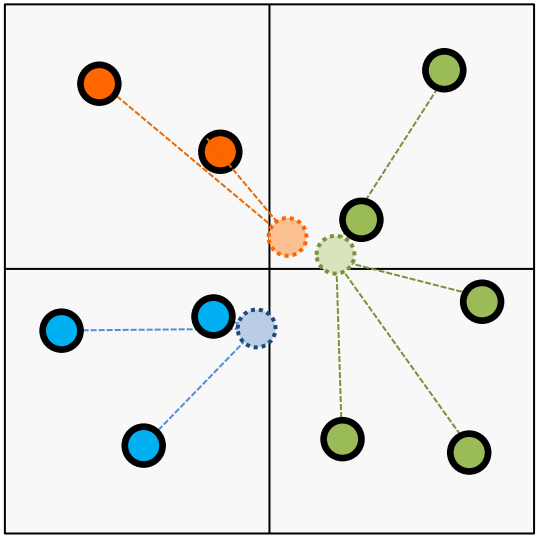
1. ランダムで初期クラスタを割り当て。



2. 各クラスタ毎にクラスタ中心点を計算

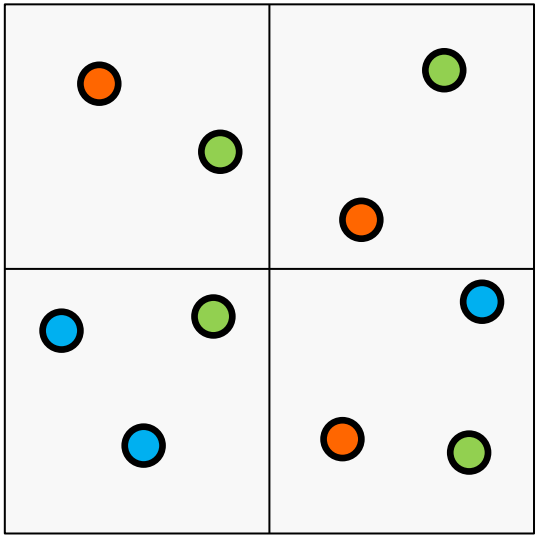


3. 各要素から最も近傍のクラスタ中心点を選択。クラスタ割当てを更新。

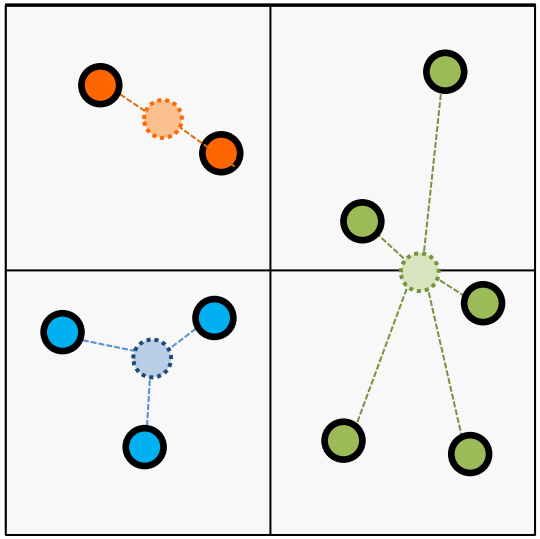


k-meansクラスタリング アルゴリズム

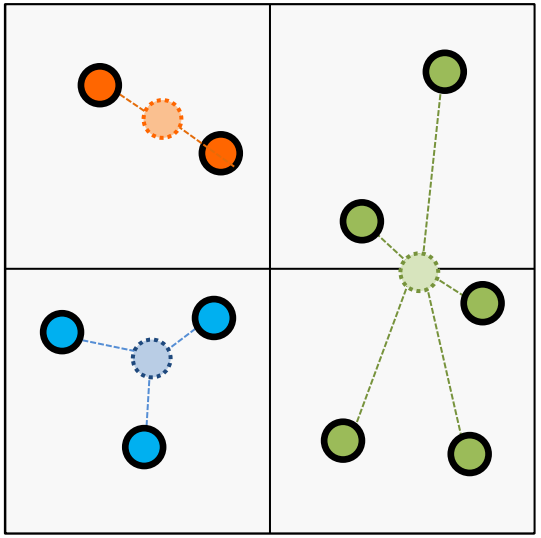
1. ランダムで初期クラスタを割り当て。



5. 新しいクラスタ割当てに基づいて、クラスタ中心点を再計算



6. クラスタ中心点が変動しなくなったので計算終了



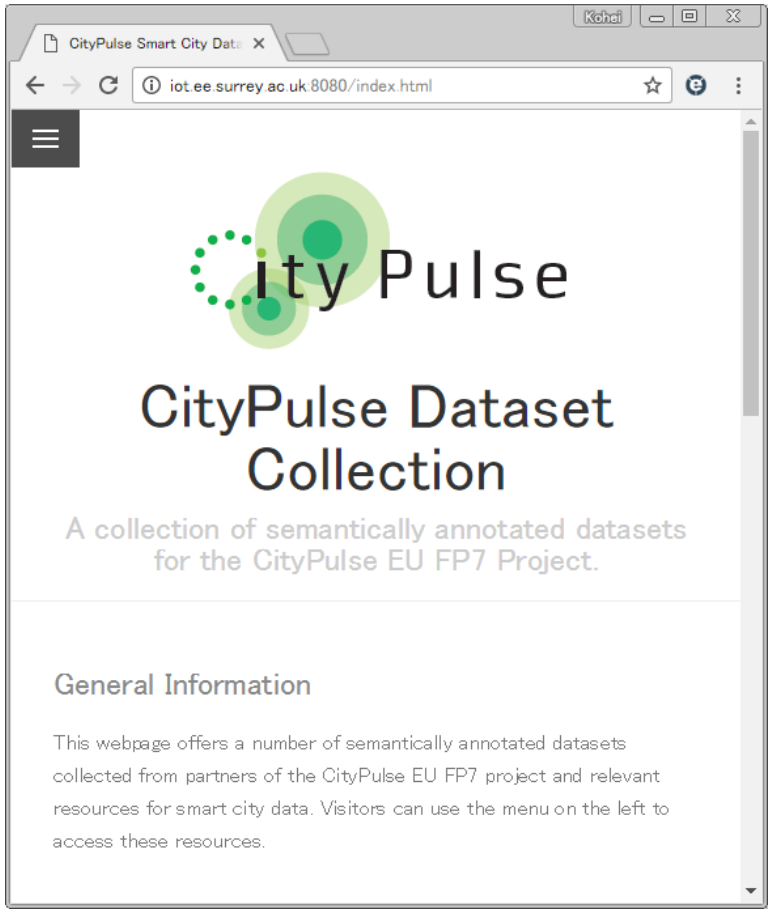
4. 収束するか、一定回数に達するまで繰り返し

PL/CUDAによるk-meansクラスタリングの実装

```
CREATE OR REPLACE FUNCTION
gpu_kmeans(real[],     -- ID + Data Matrix
           int,        -- k-value (number of clusters)
           int = 10,   -- max number of iteration
           int = 1)    -- seed of initial randomness
RETURNS int[]
AS $$
#plcuda_decl
        :
KERNEL_FUNCTION_MAXTHREADS(void)
update_centroid(MatrixType *D,
                MatrixType *R,
                MatrixType *C)
{
        :
    /* accumulate the local centroid */
    for (did = get_global_id();
         did < nitems;
         did += get_global_size())
    {
        /* pick up the target cluster */
        cid = r_values[nitems + did];
        atomicAdd(&l_cent[cid], 1.0);
        for (index=1; index < width; index++)
            atomicAdd(&l_cent[index * k_value + cid],
                      d_values[index * nitems + did]);
    }
    __syncthreads();
    /* write back to the global C-matrix */
    for (index = get_local_id();
         index < width * k_value;
         index += get_local_size())
        atomicAdd(&c_values[index], l_cent[index]);
}
```

```
        :
#plcuda_begin
        :
status = pgstromLaunchDynamicKernel4((void *)
                                setup_initial_cluster,
                                (kern_arg_t)(D),
                                (kern_arg_t)(R),
                                (kern_arg_t)(C),
                                (kern_arg_t)(r_seed),
                                nitems, 0, 0);
if (status != cudaSuccess)
    PLCUDA_RUNTIME_ERROR_RETURN(status);

for (loop=0; loop < nloops; loop++)
{
         :
    status = pgstromLaunchDynamicKernelMaxThreads3(
                    (void *)kmeans_update_cluster,
                    (kern_arg_t)(D),
                    (kern_arg_t)(R),
                    (kern_arg_t)(C),
                    (kern_arg_t)k_value,
                    nitems, 0,
                    sizeof(cl_int) + sizeof(cl_float));
    if (status != cudaSuccess)
        PLCUDA_RUNTIME_ERROR_RETURN(status);
          :
}
#plcuda_sanity_check    gpu_kmeans_sanity_check
#plcuda_working_bufsz   gpu_kmeans_working_bufsz
#plcuda_results_bufsz   gpu_kmeans_results_bufsz
#plcuda_end
$$ LANGUAGE 'plcuda';
```

k-meansクラスタリングの検証に使用したデータ



データセットの概要

- 測定区間毎に収集された自動車通行量のパブリックデータ
- デンマーク・オーフス市(Arhus, Denmark)における449観測点ののデータ。
- データ件数：1350万件（2014年2月～6月）

データに含まれるもの

- 平均速度
- 平均観測時間
- 自動車台数
- 測定区間（始点、終点）の緯度・経
- など…

やった事

- 平均速度や自動車台数によって測定区間を5つのカテゴリに分割する。

GPU版k-means関数の呼出し

```
SELECT report_id, k, c
  FROM (SELECT report_id, k, c,
               row_number() OVER (PARTITION BY report_id
                                  ORDER BY c DESC) rank
          FROM (SELECT report_id, k, count(*) c
                  FROM matrix_unnest(
                         (SELECT gpu_kmeans(
                                   array_matrix(
                                     int4_as_float4(report_id),
                                     avg_measured_time,
                                     avg_speed,
                                     vehicle_count),
                                   5)
                            FROM tr_rawdata)
                         ) R(report_id int, k int)
                GROUP BY report_id, k
               ) __summary_1
       ) __summary_2
   WHERE rank = 1;
```

Pick-up most frequent cluster

Run k-means clustering logic

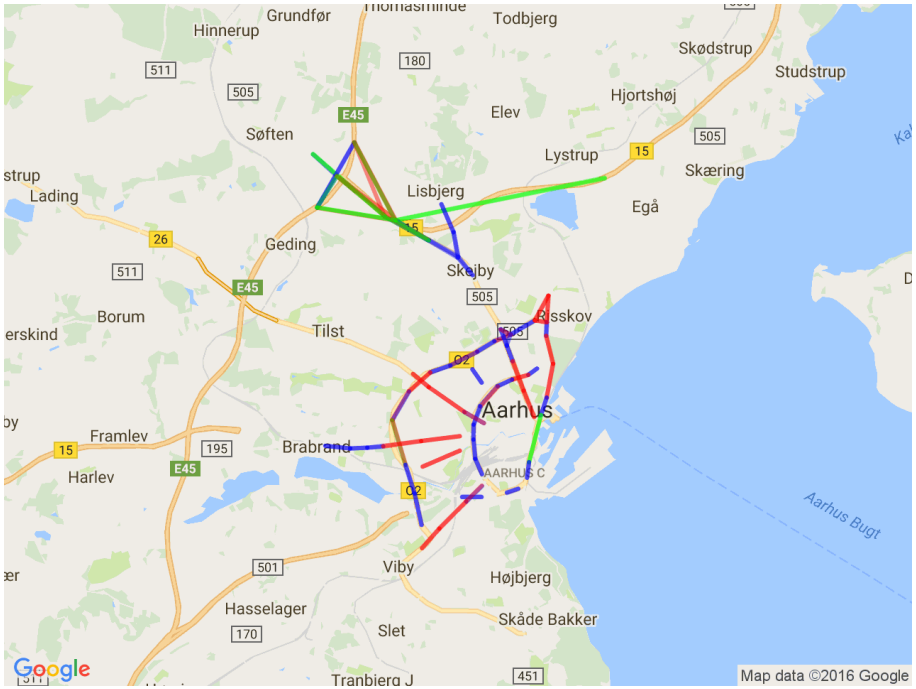
Make a matrix from the raw-data

GPU版k-means (1/3) – 全データによるクラスタリング

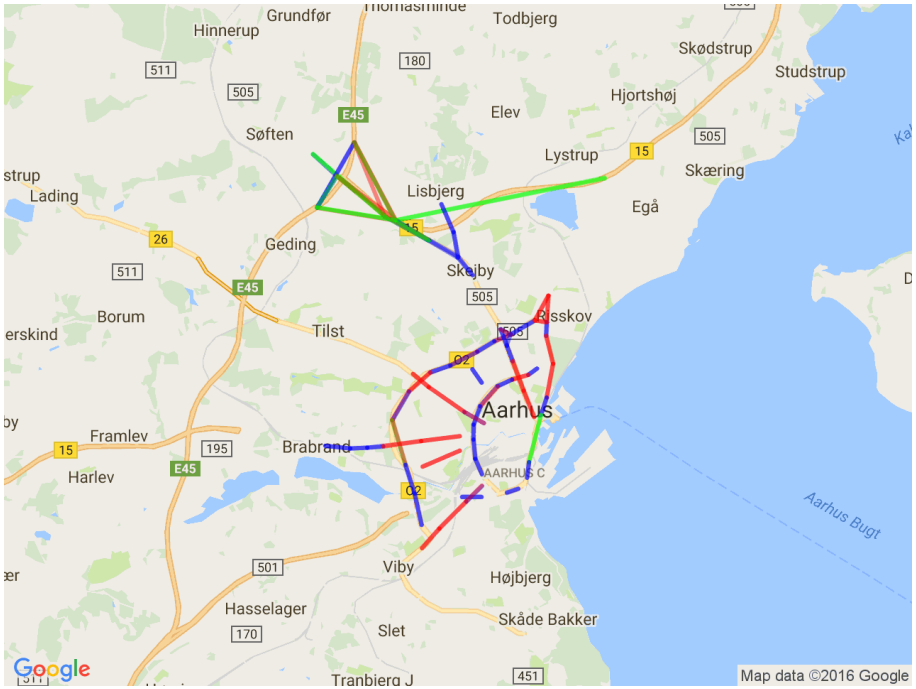
```
$ wget -O map.png "`psql traffic -At -f ~/traffic.sql`"
```

GPU版k-means (2/3) – 日中と夜間

日中 (8-17)

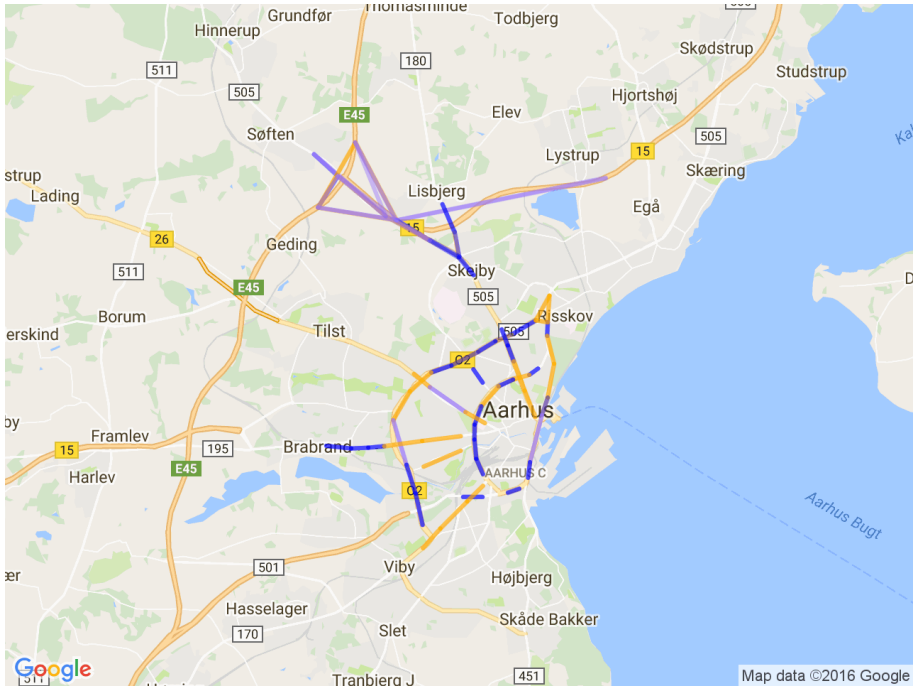


夜間 (18-7)

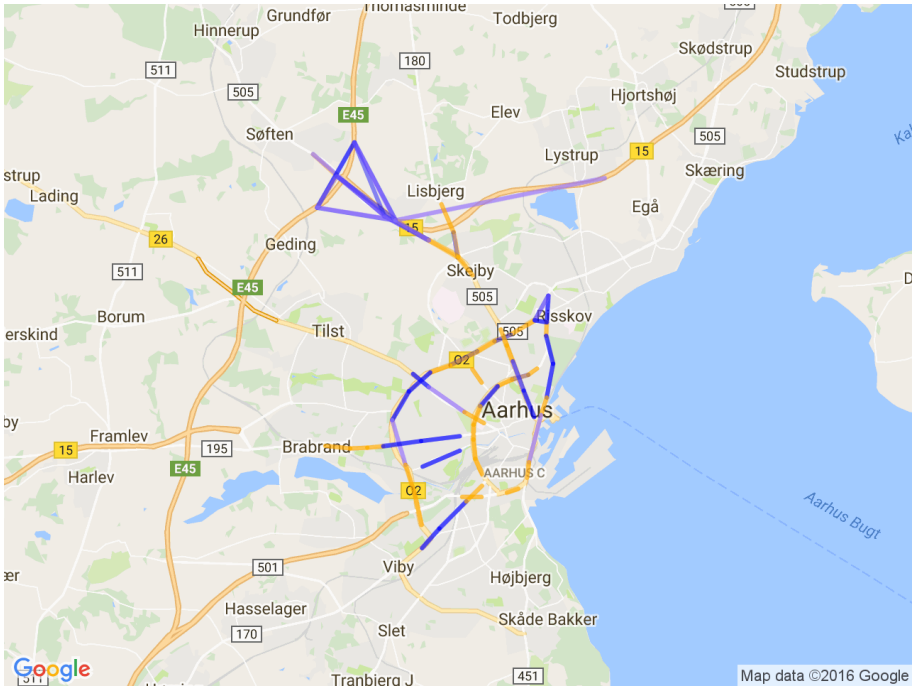


GPU版k-means (3/3) – 平日と週末

平日



週末

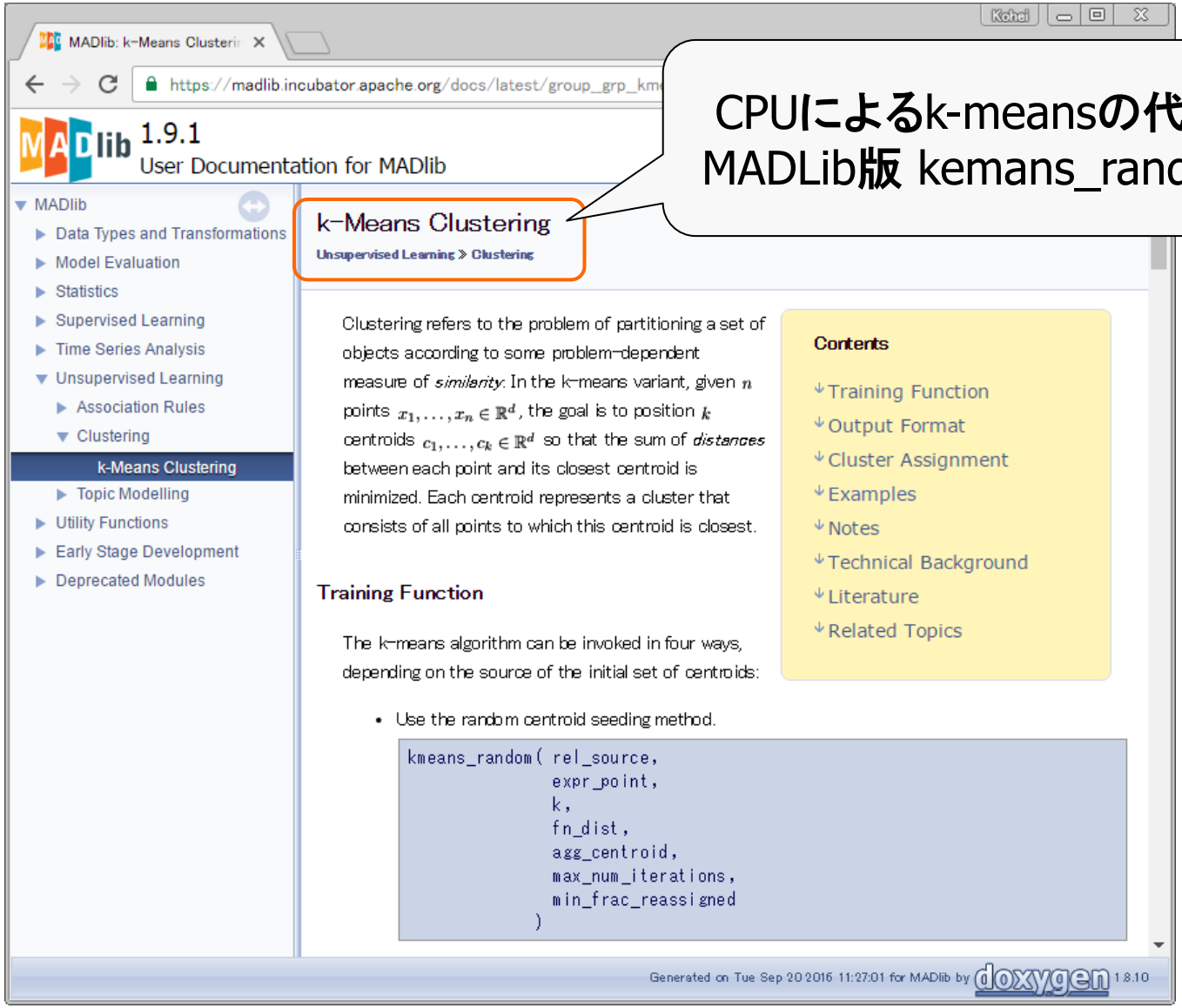


GPU版k-means関数の呼出し

```
SELECT report_id, k, c
  FROM (SELECT report_id, k, c,
               row_number() OVER (PARTITION BY report_id
                                  ORDER BY c DESC) rank
          FROM (SELECT report_id, k, count(*) c
                  FROM matrix_unnest(
                         (SELECT gpu_kmeans ( array_matrix(
                                                int4_as_float4(report_id),
                                                avg_measured_time,
                                                avg_speed,
                                                vehicle_count),
                                              5)
                            FROM tr_rawdata
                           WHERE extract('hour' from timestamp)
                                 between 7 and 17
                         )
                        ) R(report_id int, k int)
                GROUP BY report_id, k
               ) __summary_1
       ) __summary_2
   WHERE rank = 1;
```

実は条件句を追加しただけ。
（これがSQLの柔軟性！）

パフォーマンス (1/3)



CPUによるk-meansの代表的実装として、
MADLib版 kemans_random() 関数を使用

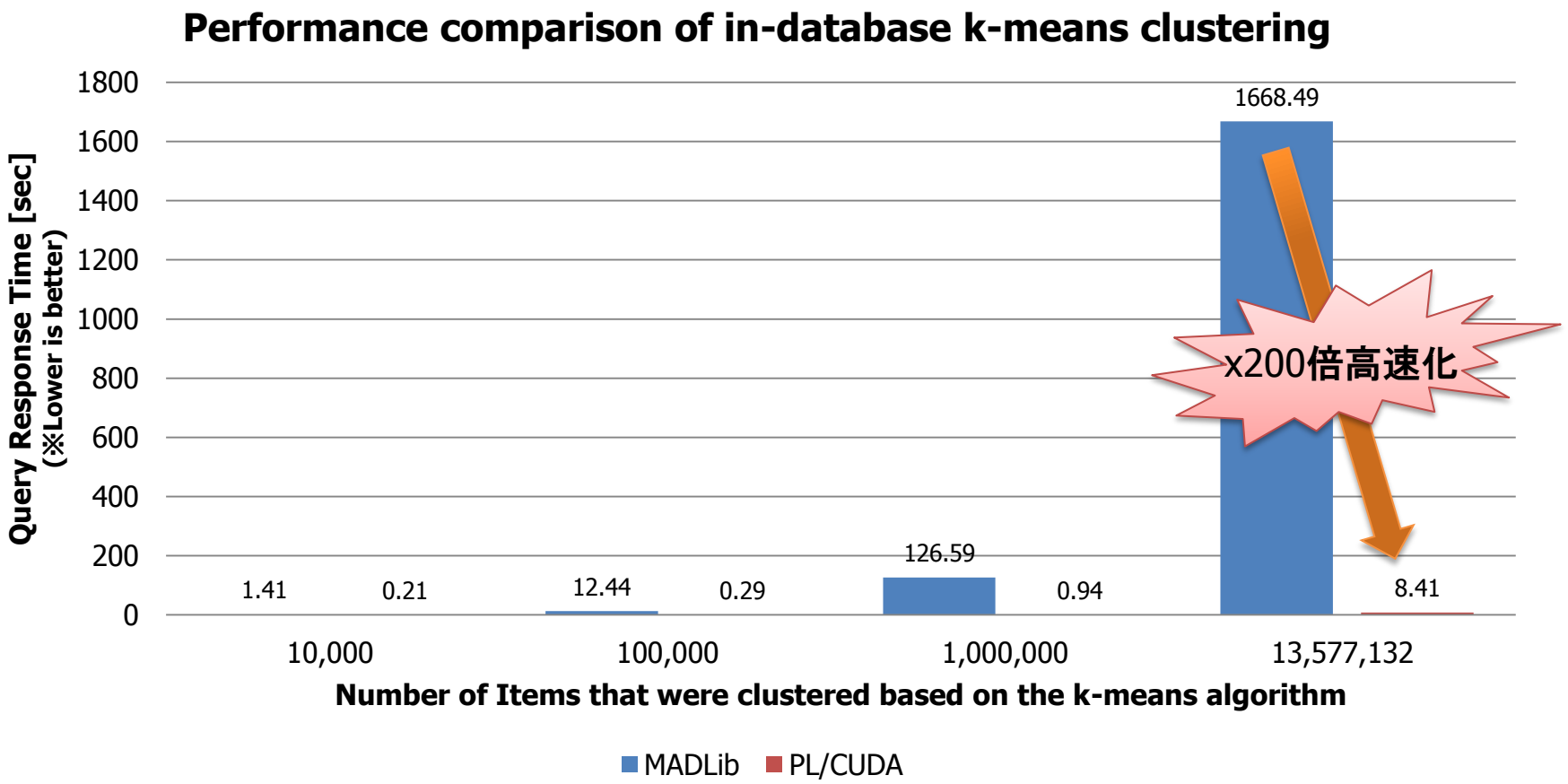
パフォーマンス (2/3) – MADLib版k-meansクラスタリングの呼出し

```
SELECT report_id, k, c
  FROM (SELECT report_id, k, c,
               row_number() OVER (PARTITION BY report_id
                                  ORDER BY c DESC) rank
          FROM (SELECT t.report_id,
                       (madlib.closest_column(centroids,
                                              t.attrs)).column_id as k,
                       count(*) c
                  FROM tr_rawdata_madlib_s t,
                       (SELECT centroids
                          FROM madlib.kmeans_random('tr_rawdata_madlib',
                                                    'attrs',
                                                    5)
                       ) km;
                 GROUP BY t.report_id, k
               ) __summary_1
       ) __summary_2
 WHERE rank = 1;
```

最近傍クラスタの選択

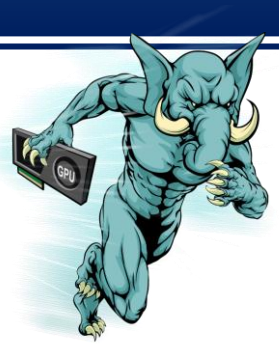
クラスタ中心点の導出

パフォーマンス (3/3) – GPU版 vs CPU版実装



- 測定環境
  - HW) CPU: Xeon E5-2670v3, GPU: GTX1080, RAM: 384GB
  - SW) CentOS7, CUDA8.0, PostgreSQL v9.5 + PG-Strom v1.0, MADLib 1.9
- CPU版は、同等のロジックをC言語によるバイナリ版で実装して比較計測

まとめ



まとめ (1/3) – おさらい

PL/CUDAとは？

- PG-Stromのオリジナルのコンセプトは自動最適化／自動コード生成
- 手動最適化と引換えに、PL/CUDAはGPU性能を最大限引き出すための手段。

➔ たぶん、高度なアルゴリズムをSQLで書いている人はいないので正しい選択

利点

- TFLOPS級の計算エンジンをIn-Database Analyticsで使用できる。
- 外部アプリケーションを使用する場合と異なり、データセット全体をDBからエクスポートする必要がなくなる。
  - ➔ 取り出す必要があるのは“処理結果”だけ
- 解析アルゴリズムの前処理/後処理で、SQLによる柔軟なデータ操作が可能
  - ✓ JOIN、GROUP BY、Window関数、etc...

まとめ (2/3) – 適用領域

実証実験① – 創薬領域における類似化合物検索

- 化合物の特徴を fingerprint (= 特徴ベクトル)として表現。
  GPUで類似度スコアを計算しスコア上位の組合せを抽出。

実証実験② – センサデータを用いた教師なし学習

- センサが生成した情報を用いてGPUで要素間距離を計算。
  道路の特徴を抽出して数個のカテゴリへと自動分類。

考えられる適用領域

- ✓ 化合物探索 … 医薬、化学品、素材系
- ✓ レコメンデーションエンジン … e-コマース領域
- ✓ アノマリー検知 … セキュリティ分野
- ✓ データマイニング … マーケティング
- ✓ …など…

まとめ (3/3) – 課題と展望

技術面

- 1GBを越えるサイズのArray-Matrixの取り扱い
  - ✓ PostgreSQL可変長データ型の制約により、SQL関数の個々の引数が1GBより大きなサイズを持つ事ができない。
  - ✓ (↑12/1のdeveloper meetingでも議論する)
- 繰り返し何度も同じArray-Matrixを使用する場合の処理コスト
  - ✓ GPU側に静的なArray-Matrixを置いたままにできれば嬉しいかも。

運用面

- SQLとCUDAを両方書ける人はなかなか多くない。
  - ✓ "定番の" アルゴリズムをパッケージ化
  - ✓ 専門エンジニアによるコンサルティング、構築サービス

実証実験のお誘い、お待ちしております。

リソース

リポジトリ

https://github.com/pg-strom/devel

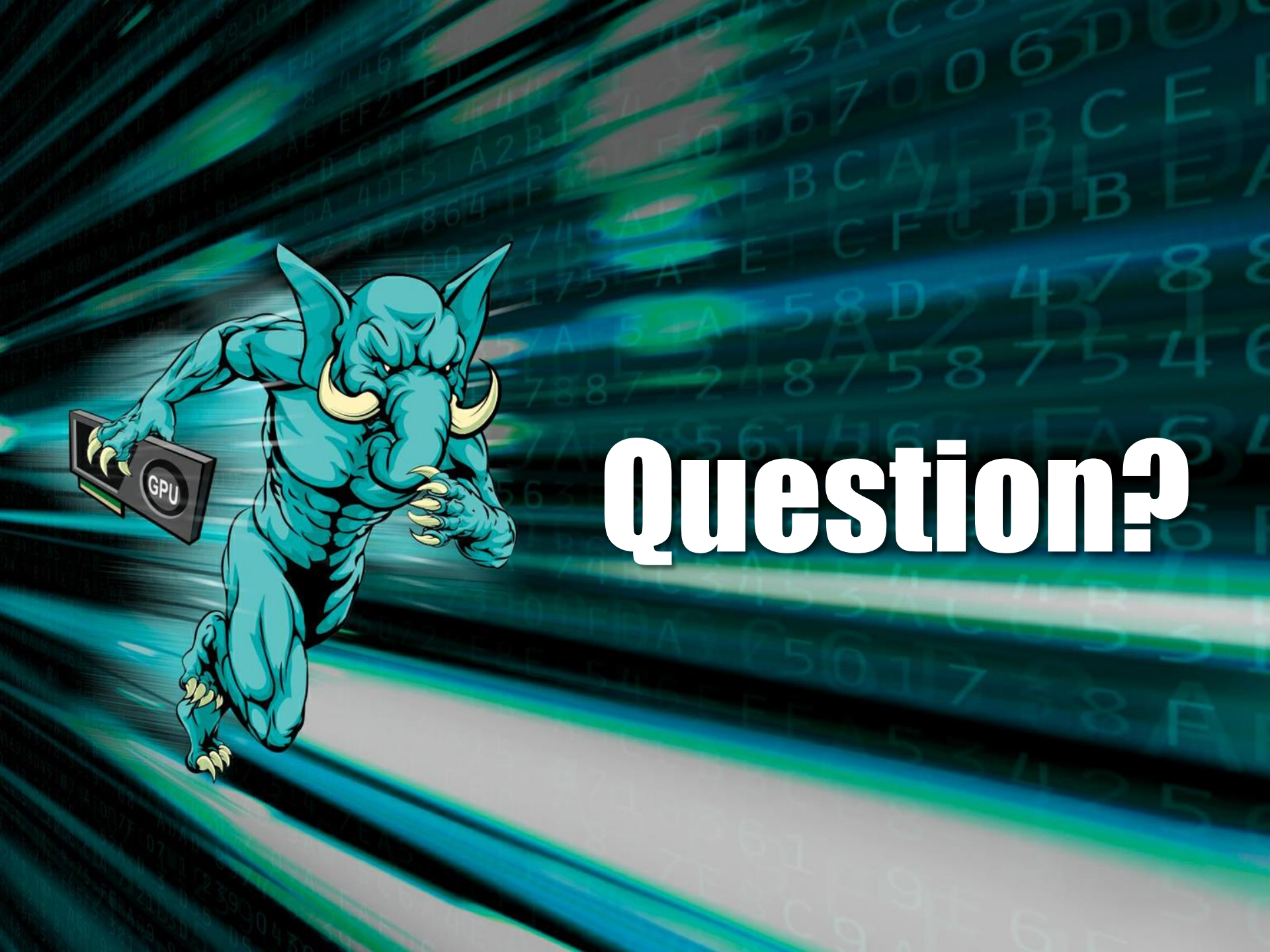
本日のスライド

http://www.slideshare.net/kaigai/pgconfasia2016-plcuda

コンタクト

- e-mail: kaigai@ak.jp.nec.com
- Tw: @kkaigai

一緒に開発しようぜ！って人もお待ちしております。


GPU
Question?