Empowered by Innovation NEC

GPGPUがPostgreSQLを加速する

～PG-Stromご紹介～

NEC OSS推進センター
The PG-Strom Project
海外 浩平 <kaigai@ak.jp.nec.com>
(Tw: @kkaigai)

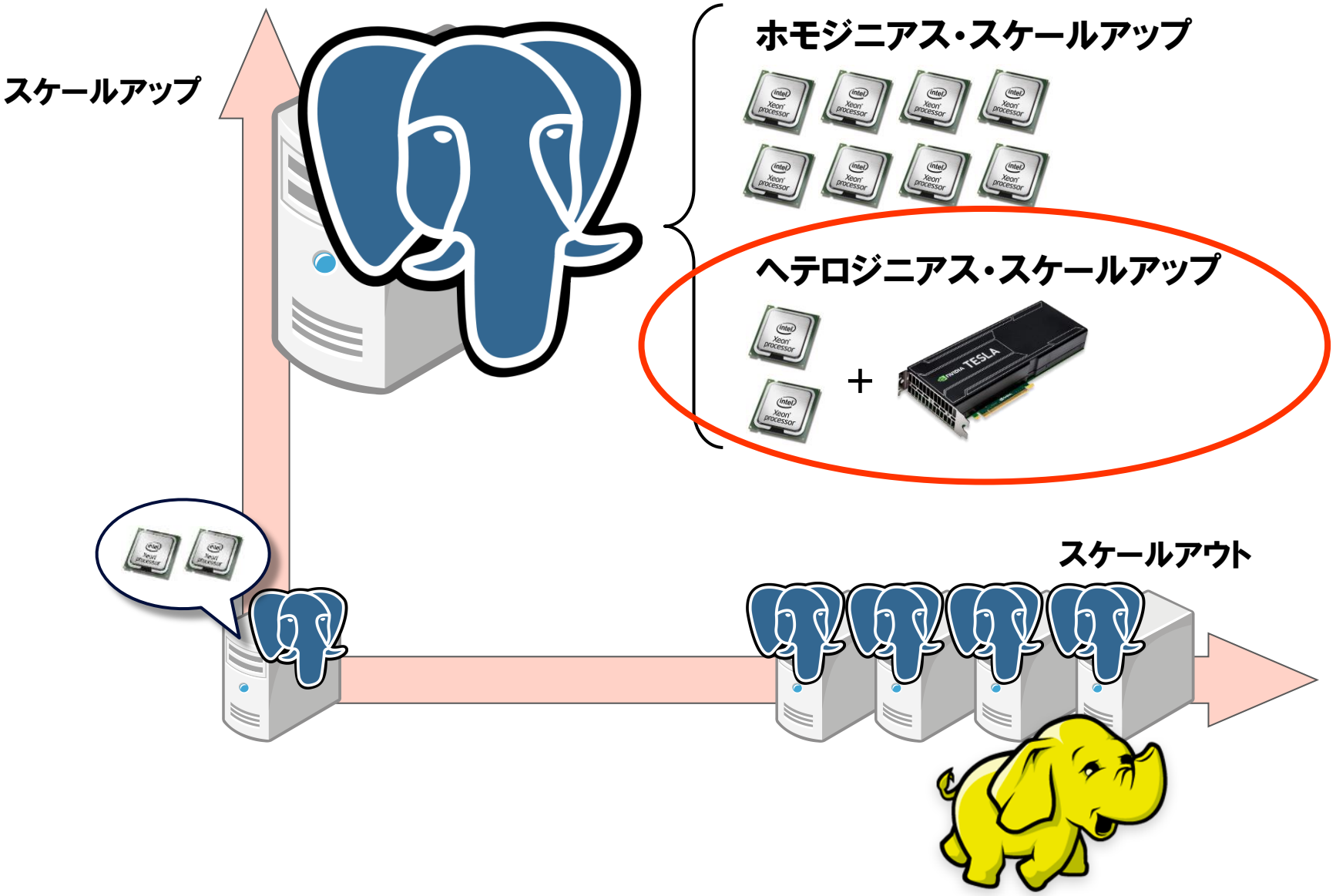
自己紹介

- 名前: 海外 浩平
- 所属: NEC OSS推進センター
- 好きなもの: コアの多いプロセッサ
- 嫌いなもの: コアの少ないプロセッサ
- 経歴:
  - HPC ➔ OSS/Linux ➔ SAP ➔ GPU/PostgreSQL
- Tw: @kkaigai

主な仕事

- SELinux周り諸々 (2004~)
  - Lockless AVC、JFFS2 XATTRなど
- PostgreSQL周り諸々 (2006~)
  - SE-PostgreSQL、Security Barrier View、Writable FDWなど
- PG-Strom (2012~)

処理性能向上のアプローチ



GPU (Graphic Processor Unit) の特徴

SOURCE: CUDA C Programming Guide (v6.5)

	GPU	CPU
Model	Nvidia Tesla K20X	Intel Xeon E5-2690 v3
Architecture	Kepler	Haswell
Launch	Nov-2012	Sep-2014
# of transistors	7.1billion	3.84billion
# of cores	2688 (simple)	12 (functional)
Core clock	732MHz	2.6GHz, up to 3.5GHz
Peak Flops (single precision)	3.95TFLOPS	998.4GFLOPS (AVX2使用)
DRAM size	6GB, GDDR5	768GB/socket, DDR4
Memory band	250GB/s	68GB/s
Power consumption	235W	135W
Price	$3,000	$2,094

GPUの特徴

- チップに占める演算ユニットの比率が高い
- キャッシュ・制御ロジックの比率が小さい

➔ 単純な演算の並列処理に向く。
複雑なロジックの処理は苦手。

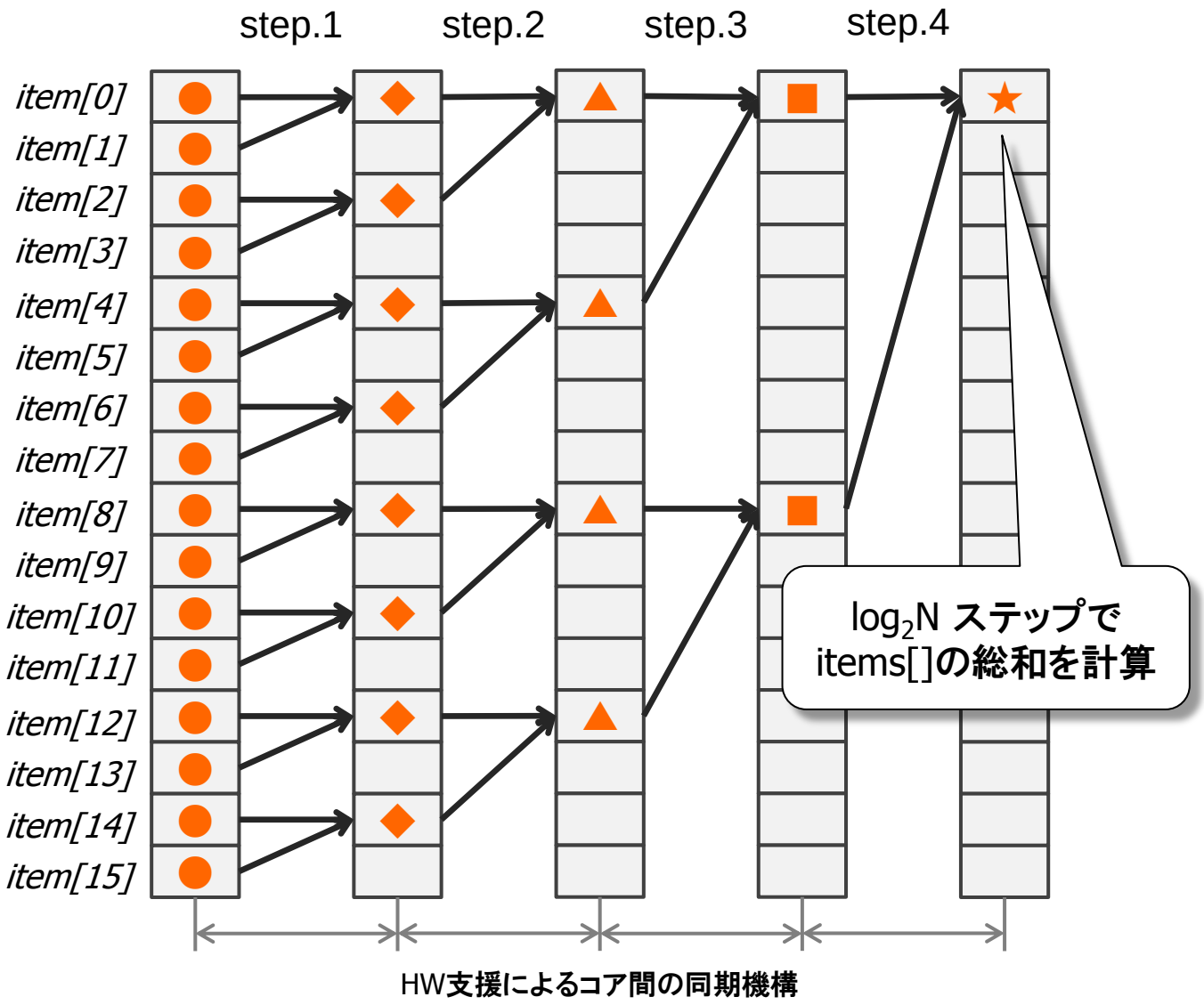
- 値段の割にコア数が多い
  - GTX750Ti(640core)で 1万5千円くらい

GPU活用による計算の例 – 縮約アルゴリズム

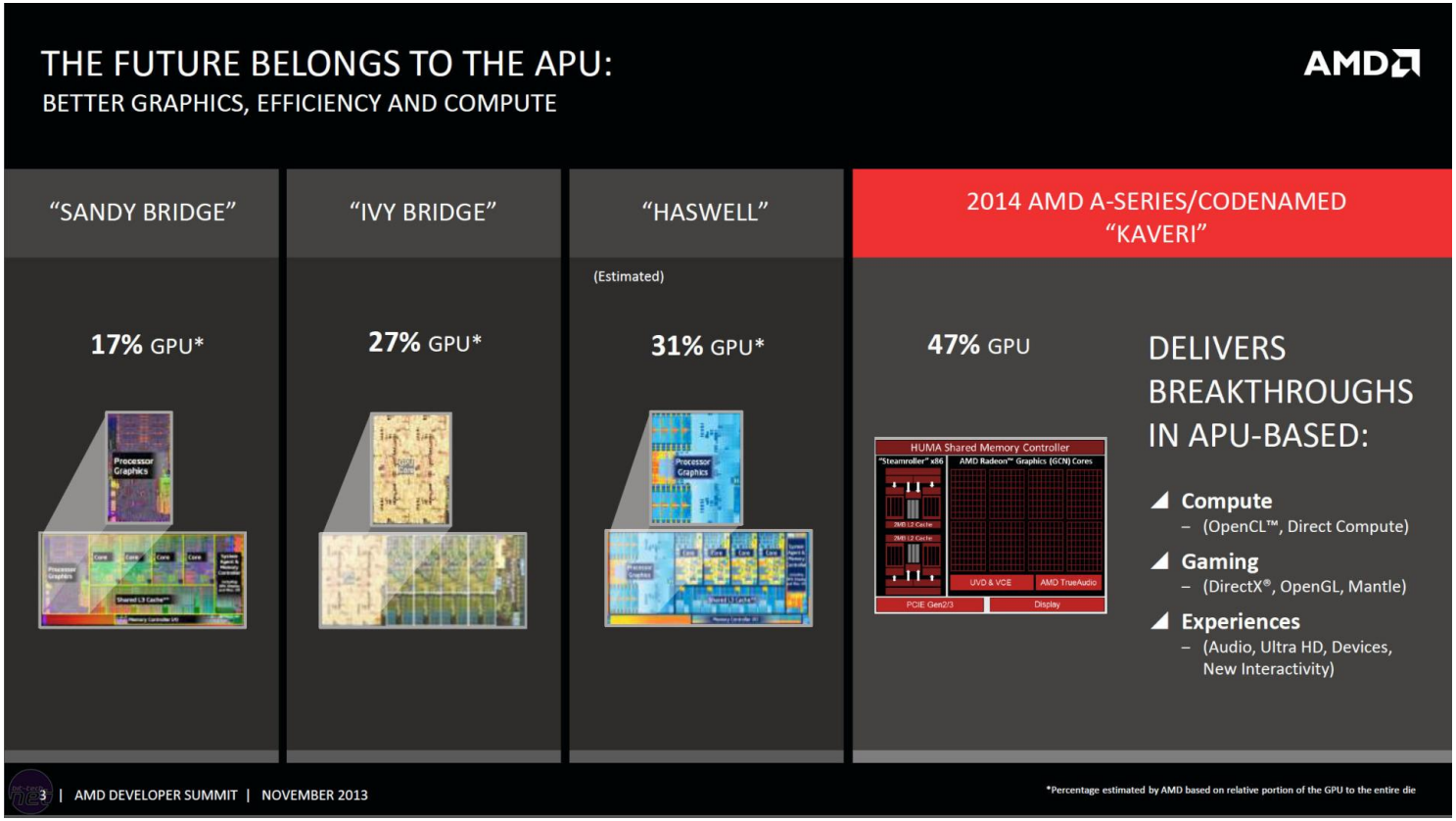
N個のGPUコアによる

$\Sigma_{i=0...N-1}$item[i]

配列総和の計算



半導体技術のトレンド (1/2) – ヘテロジニアス化

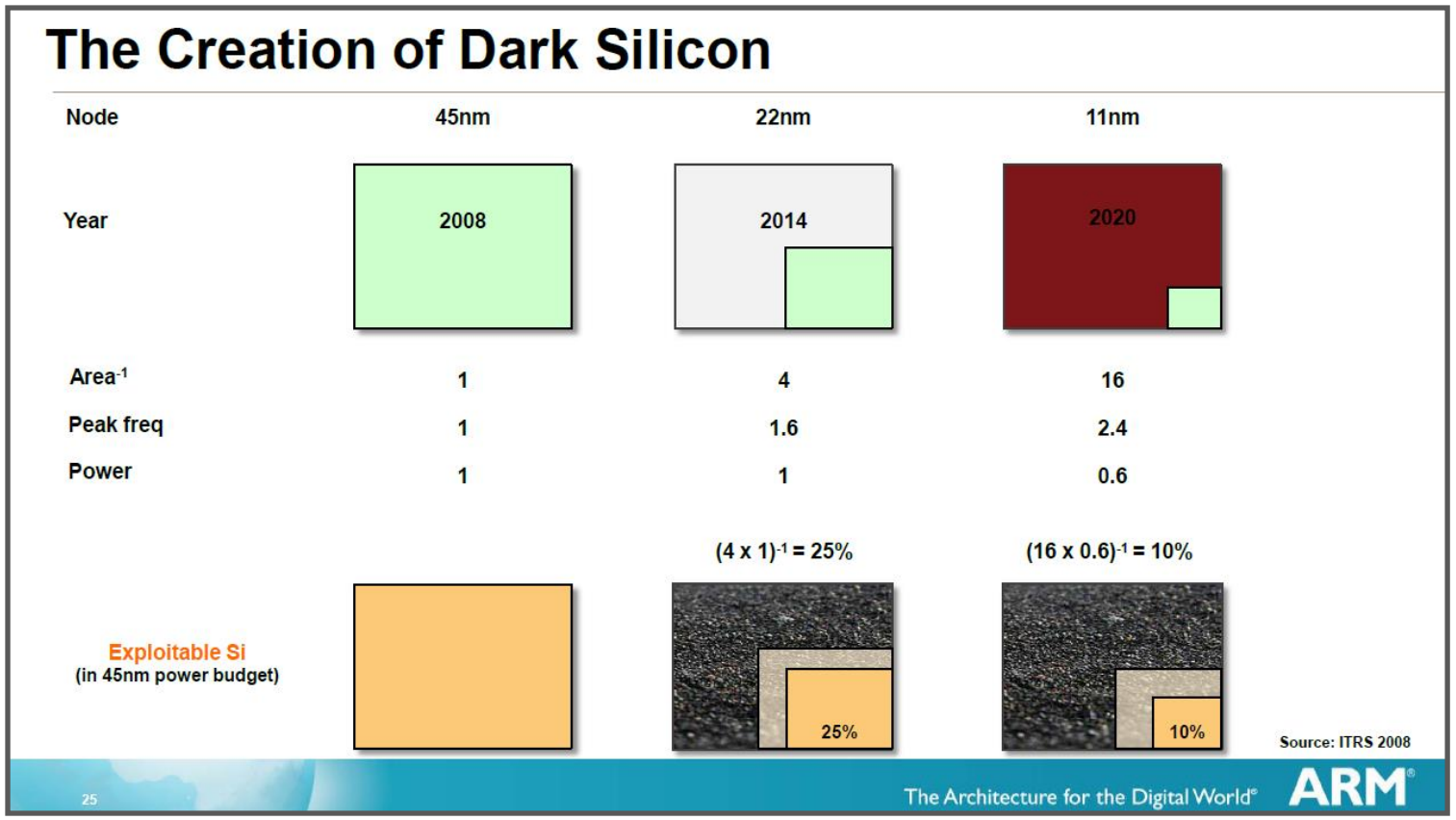


SOURCE: THE HEART OF AMD INNOVATION, Lisa Su, at AMD Developer Summit 2013

- マルチコアCPUから、CPU/GPU統合型アーキテクチャへの移行
- HW性能向上からSWがフリーランチを享受できた時代は終わりつつある。

➔ HW特性を意識してSWを設計しないと、半導体の能力を引き出せない。

半導体技術のトレンド (2/2) – ダークシリコン問題



SOURCE: Compute Power with Energy-Efficiency, Jem Davies, at AMD Fusion Developer Summit 2011

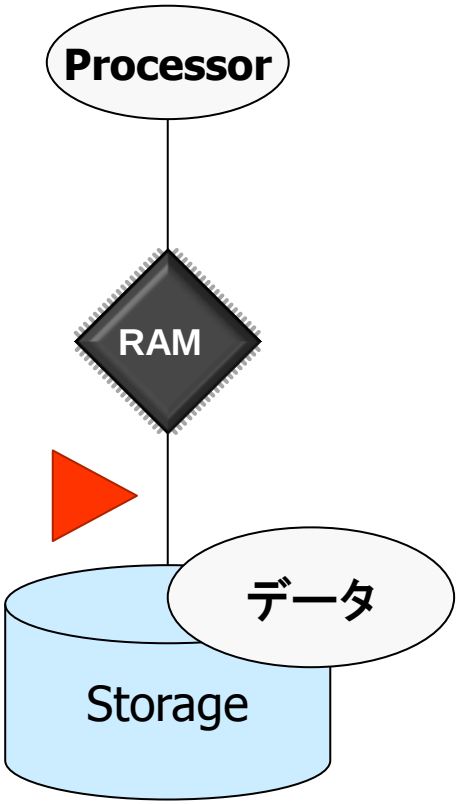
CPU/GPU統合型アーキテクチャの背景

- トランジスタ集積度向上のスピード > トランジスタあたり消費電力削減のスピード
- 排熱が追いつかないので、全ての回路に同時に電力供給するワケにはいかない。

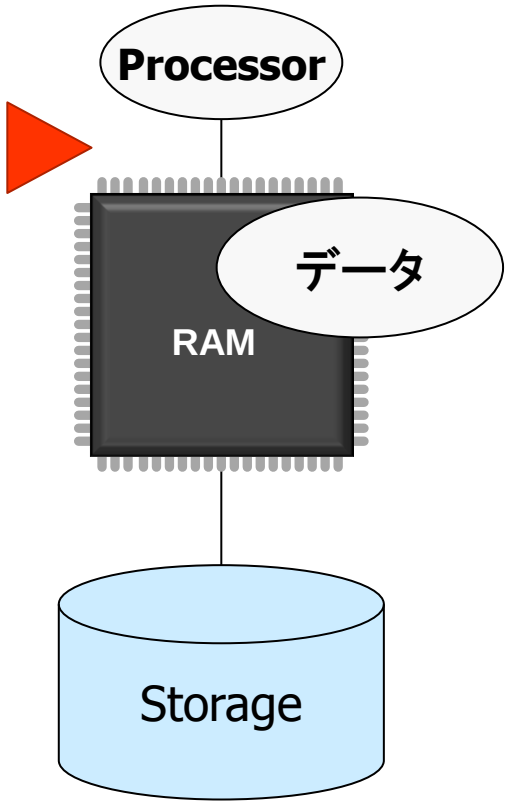
➔ 同じチップ上に特性の異なる回路を実装。ピーク電力消費を抑える。

RDBMSとボトルネックを考える (1/2)

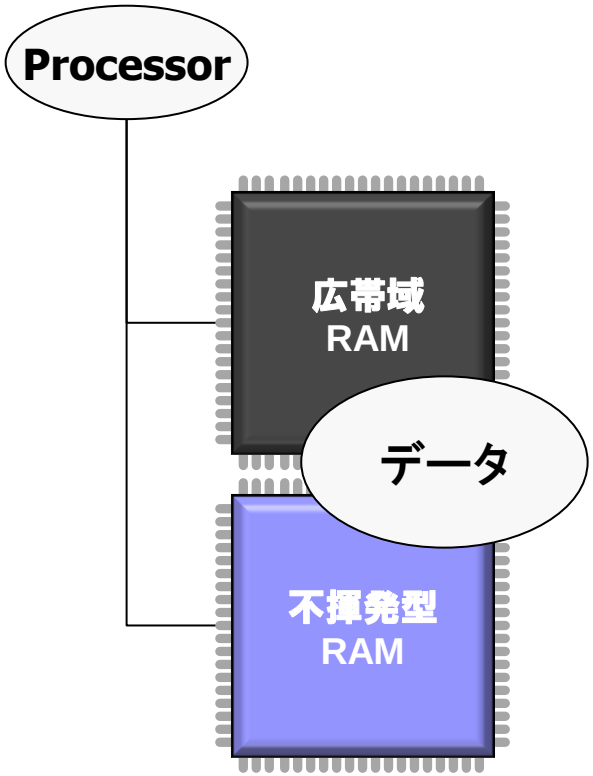
データサイズ > RAM容量



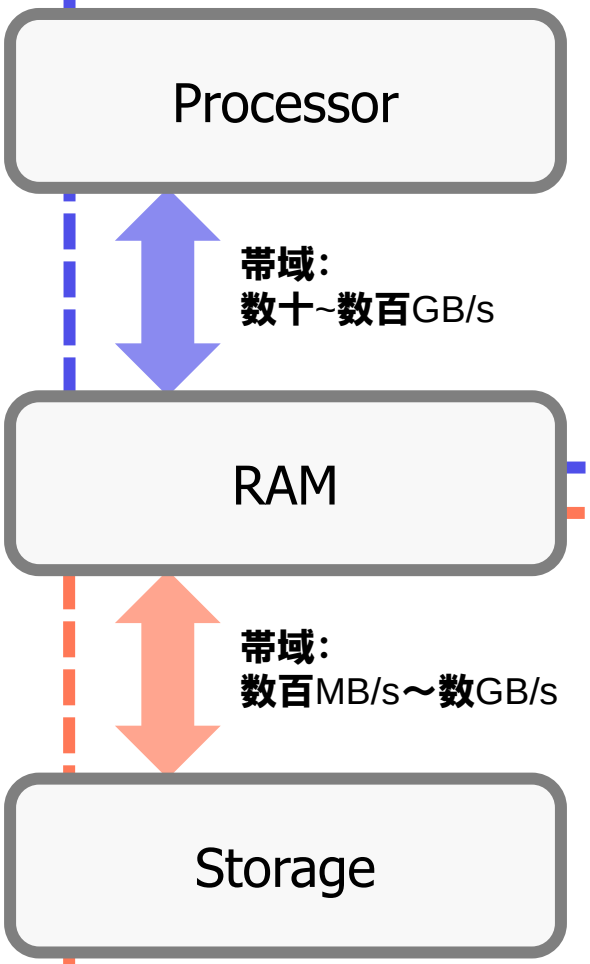
データサイズ < RAM容量



将来は？



RDBMSとボトルネックを考える (2/2)



現代のメモリボトルネックの世界

Join、Aggregation、Sort、Projectionなど

戦略

- バーストしやすいアクセスパターン
- 並列計算アルゴリズムの適用

伝統的なディスクI/Oボトルネックの世界

SeqScan、IndexScanなど

戦略

- I/Oの量 (サイズ、回数) を減らす
- ディスク分散 (RAID)

PG-Stromのアプローチ

PG-Stromとは

- PostgreSQL用の拡張モジュール
- SQL処理ワークロードの一部を**GPUで並列実行**し、高速化を実現。
- Full-Scan、Hash-Join、Aggregate の3種類のワークロードに対応

(2014年11月時点のβ版での対応状況)

コンセプト

- SQL構文からGPU用のネイティブ命令を動的に生成。JITコンパイル。
- CPU/GPU協調による非同期並列実行

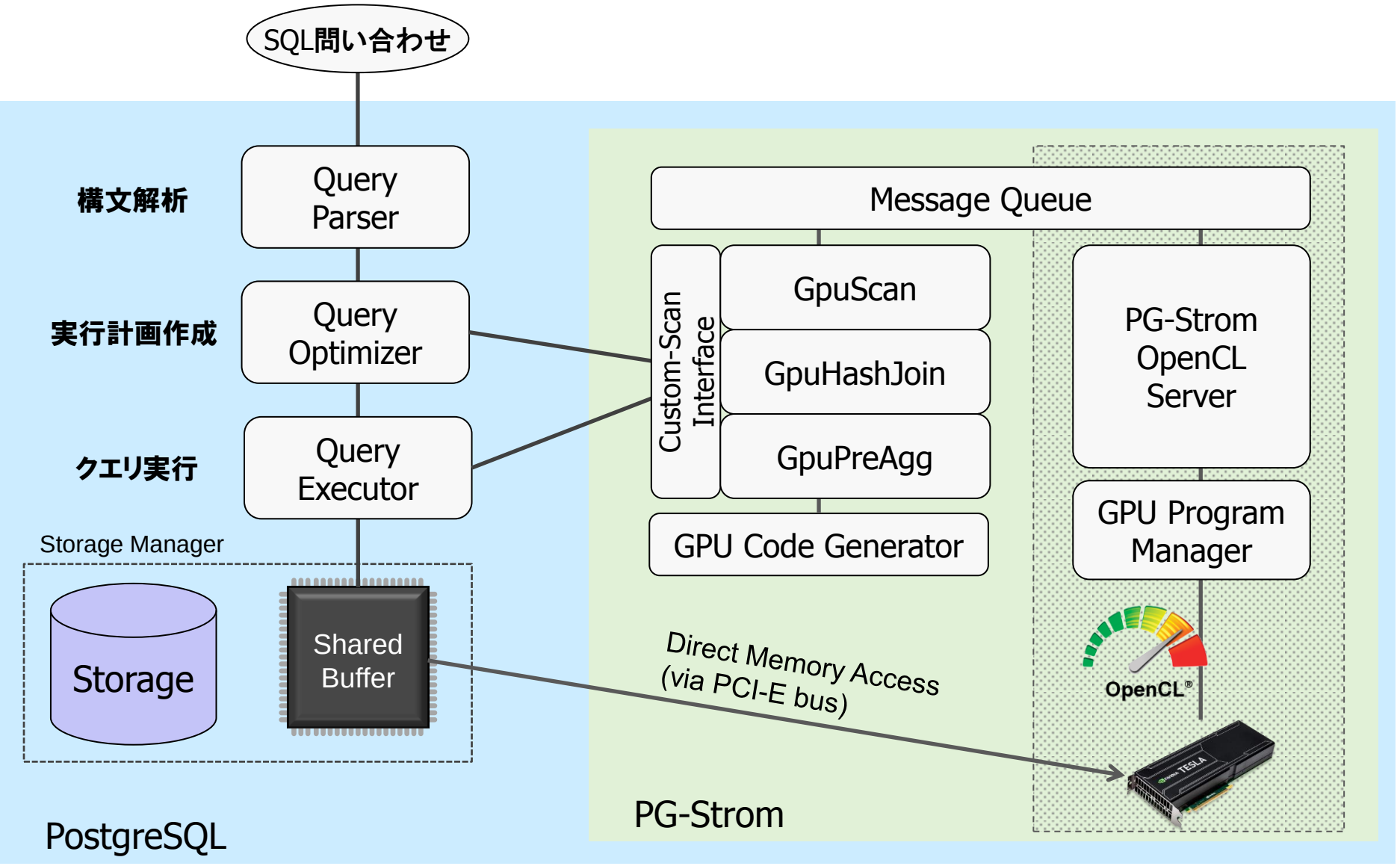
利点

- 利用者からは完全に透過的に動作。
  - SQL構文や周辺ツール、ドライバを含めPostgreSQL向けのソフトウェア資産を利用可
- GPU+OSSの組み合わせにより、低コストでの性能改善が計れる。

注意点

- ただし、現時点ではインメモリ・データストアが前提。

PG-Stromのアーキテクチャ (1/2)



PG-Stromのアーキテクチャ (2/2)

要素技術①　OpenCL

- ヘテロジニアス計算環境を用いた並列計算フレームワーク
- NVIDIAやAMDのGPUだけでなく、CPU並列にも適用可能
- 言語仕様にランタイムコンパイラを含む。

要素技術②　Custom-Scan Interface

- あたかもPostgreSQLのSQL処理ロジックであるかのように、拡張モジュールがスキャンやジョインを実装するための機能。
- PostgreSQL v9.5で一部機能がマージ。フル機能の標準化に向けて、開発者コミュニティで議論が進んでいる。

要素技術③　行指向データ構造

- PostgreSQLの共有バッファをDMA転送元として利用する。
- GPUの性能を引き出すには列指向データが最適ではあるものの、行⇔列変換が非常にコスト高で本末転倒に。

要素技術① OpenCL (1/2)

OpenCL の特徴

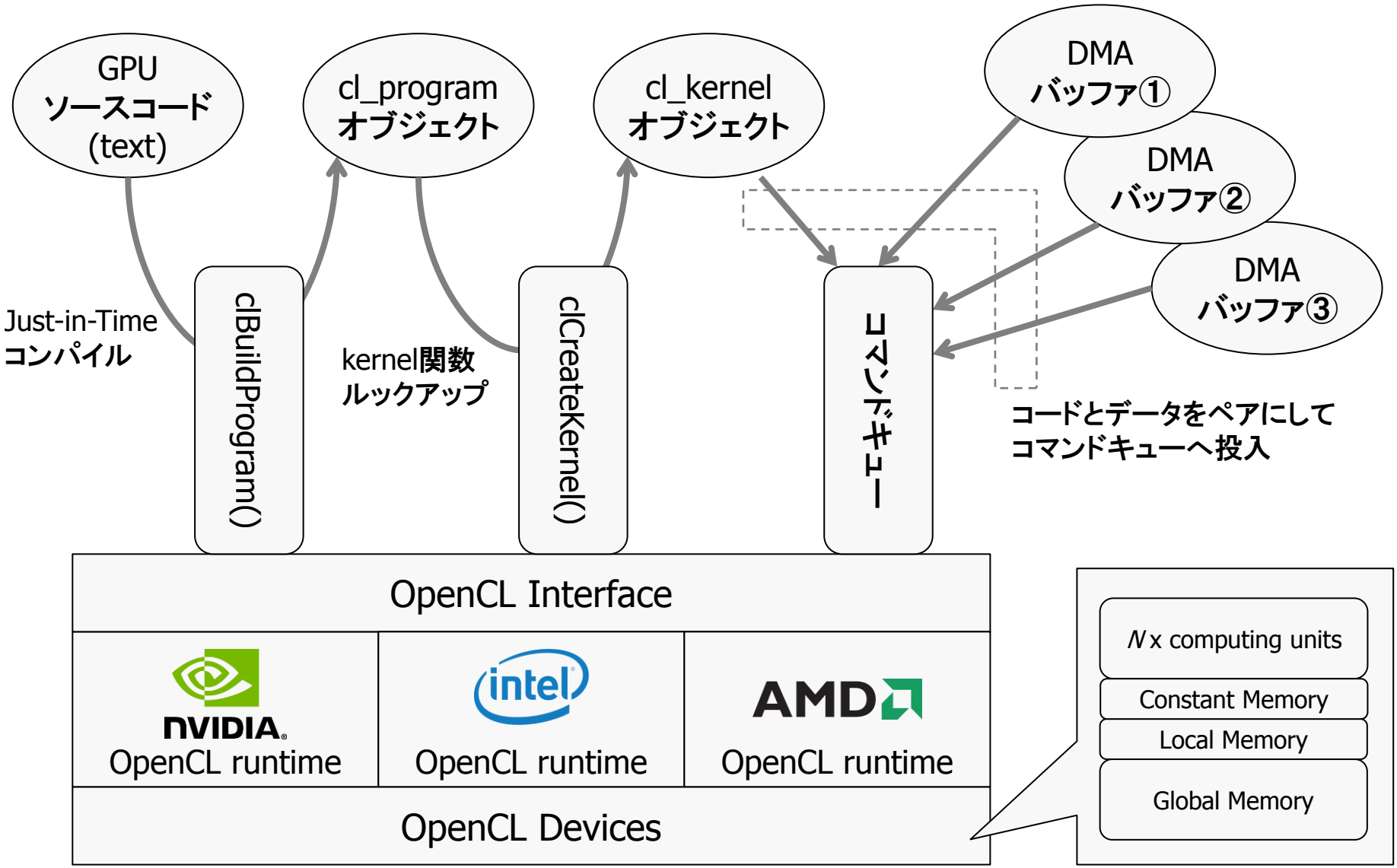
- GPUだけでなく、CPUやMICも含め “OpenCLデバイス” として抽象化。
  - とはいえ、概ねGPUの特徴を踏襲しているが…。
  - ✓三種類のメモリ階層 (global, local, constant)
  - ✓一定数のスレッドがプログラムカウンタを共有 (WARP; NVIDIAでは32個)
- C言語ライクなソースコードから、JITコンパイルによってプラットフォーム固有のネイティブ実行コードを生成。

CUDAとの比較

	CUDA	OpenCL
利点	• 細かな最適化が可能 • NVIDIA GPUの最新機能 • ドライバの実績・安定性	• マルチプラットフォームへの対応 • JITコンパイラを言語仕様に含む • CPU並列へも対応が可能
課題	• AMD、Intel環境での利用不可	• ドライバの実績・安定性

➔ ① JITコンパイルの仕組みを自前で作らなくても良い事、
② 先ずは幅広く使用してもらう事、を優先して OpenCL を選択

要素技術① OpenCL (2/2)

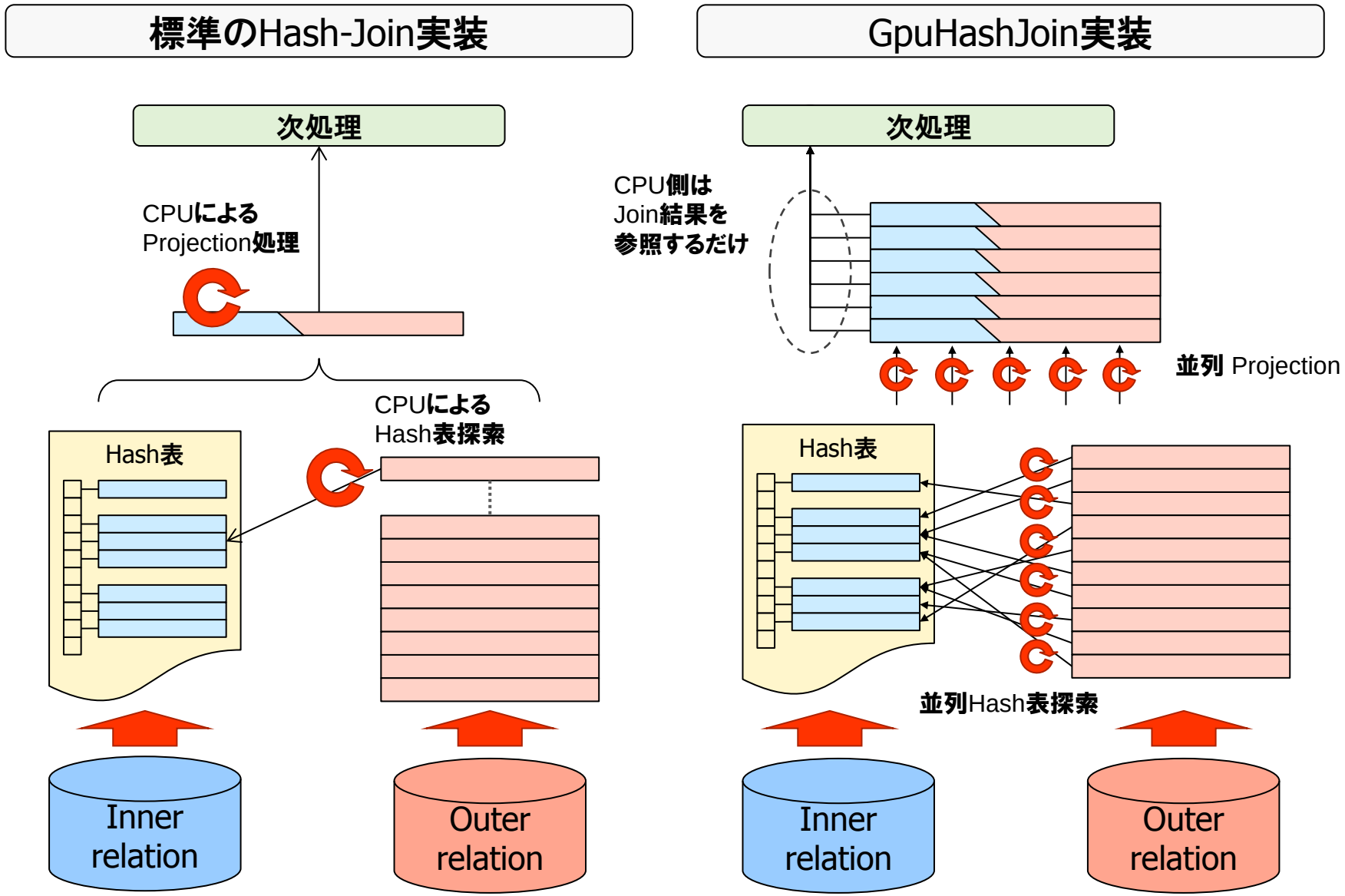


GPUコードの自動生成

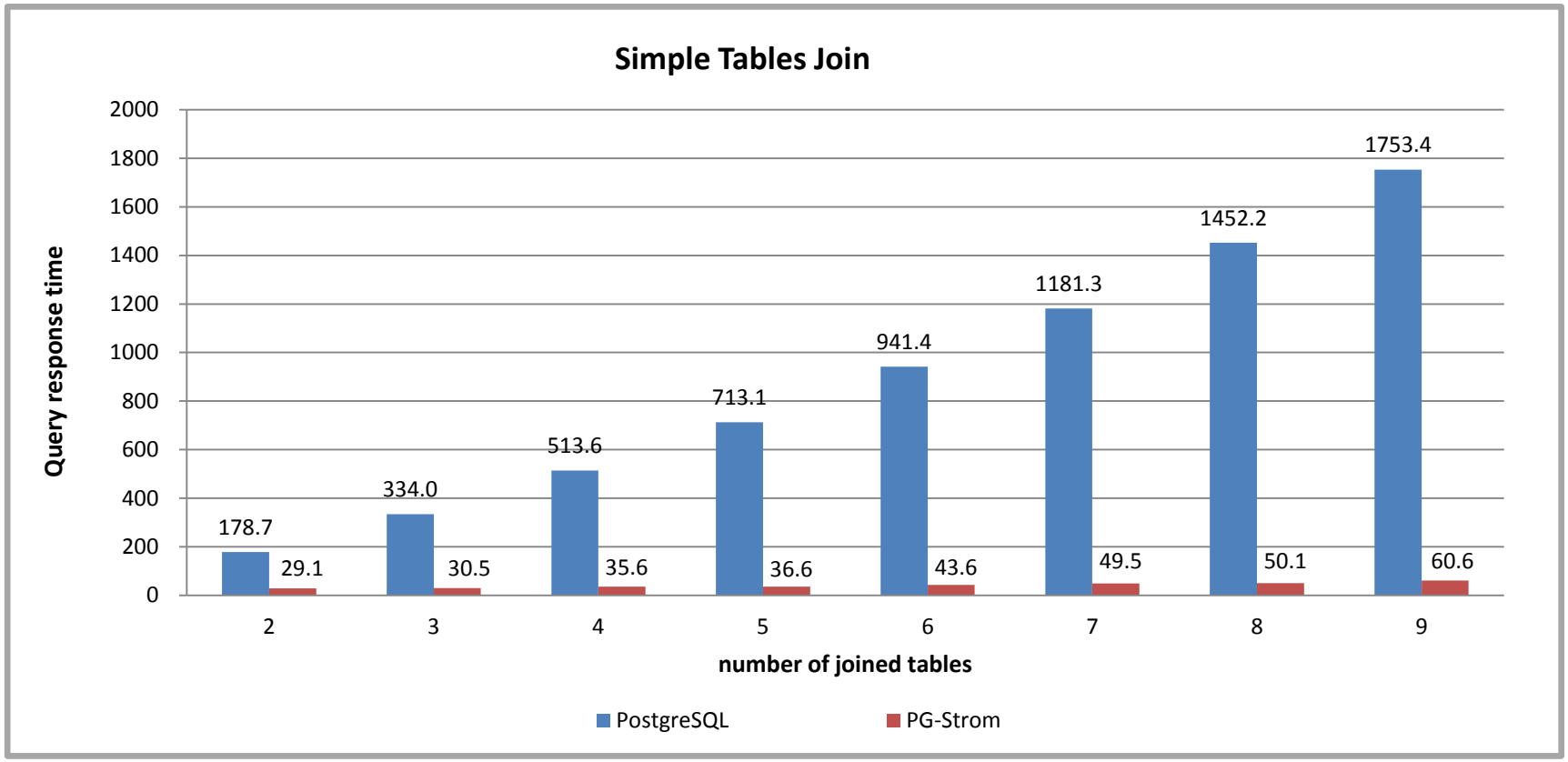
```
postgres=# SET pg_strom.show_device_kernel = on;
SET
postgres=# EXPLAIN (verbose, costs off) SELECT cat, avg(x) from t0 WHERE x < y GROUP BY cat;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 HashAggregate
   Output: cat, pgstrom.avg(pgstrom.nrows(x IS NOT NULL), pgstrom.psum(x))
   Group Key: t0.cat
   ->  Custom (GpuPreAgg)
         Output: NULL::integer, cat, NULL::integer, NULL::integer, NULL::integer, NULL::integer,
                 NULL::double precision, NULL::double precision, NULL::text,
                 pgstrom.nrows(x IS NOT NULL), pgstrom.psum(x)
         Bulkload: On
         Kernel Source: #include "opencl_common.h"
 #include "opencl_gpupreagg.h"
 #include "opencl_textlib.h"
         :  <...snip...>
 static bool
 gpupreagg_qual_eval(__private cl_int *errcode,
                     __global kern_parambuf *kparams,
                     __global kern_data_store *kds,
                     __global kern_data_store *ktoast,
                     size_t kds_index)
 {
   pg_float8_t KVAR_7 = pg_float8_vref(kds,ktoast,errcode,6,kds_index);
   pg_float8_t KVAR_8 = pg_float8_vref(kds,ktoast,errcode,7,kds_index);

   return EVAL(pgfn_float8lt(errcode, KVAR_7, KVAR_8));
 }
```

PG-Stromの処理イメージ① – Hash-Joinのケース



ベンチマーク (1/2) – Simple Tables Join

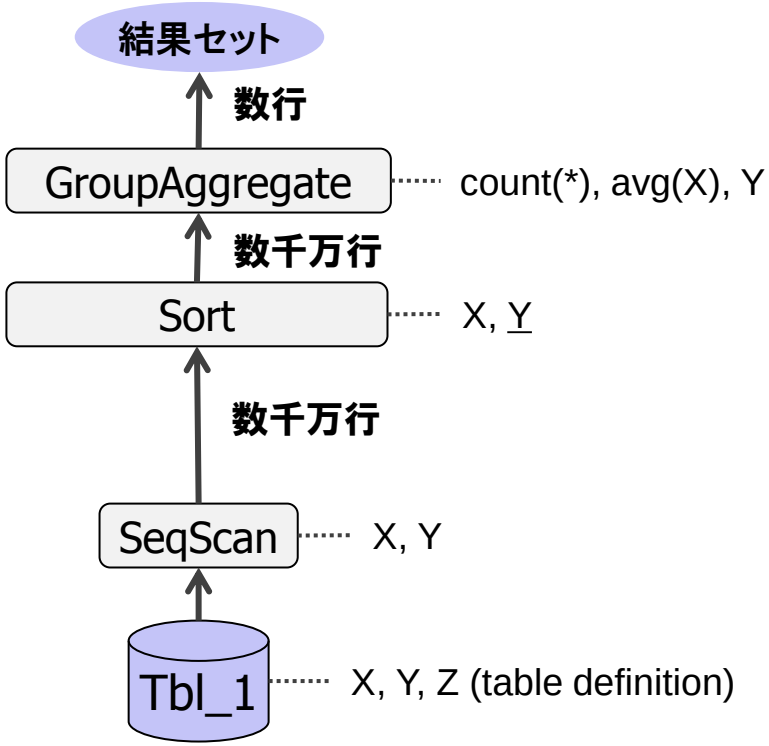


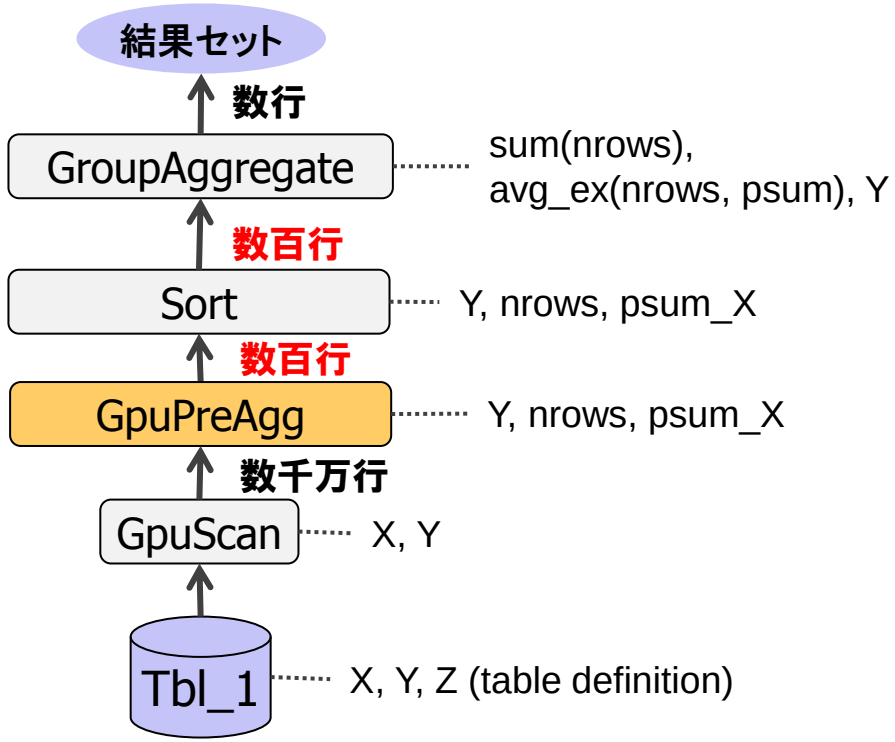
[測定条件]

- 2億件 x 10万件 x 10万件 ... のINNER JOINをテーブル数を変化させながら実行
- 使用したクエリ: `SELECT * FROM t0 natural join t1 [natural join t2 ...];`
- 全てのテーブルは事前にバッファにロード済み
- HW: Express5800 HR120b-1, CPU: Xeon E5-2640, RAM: 256GB, GPU: NVIDIA GTX980

PG-Stromの処理イメージ② – 集約演算のケース

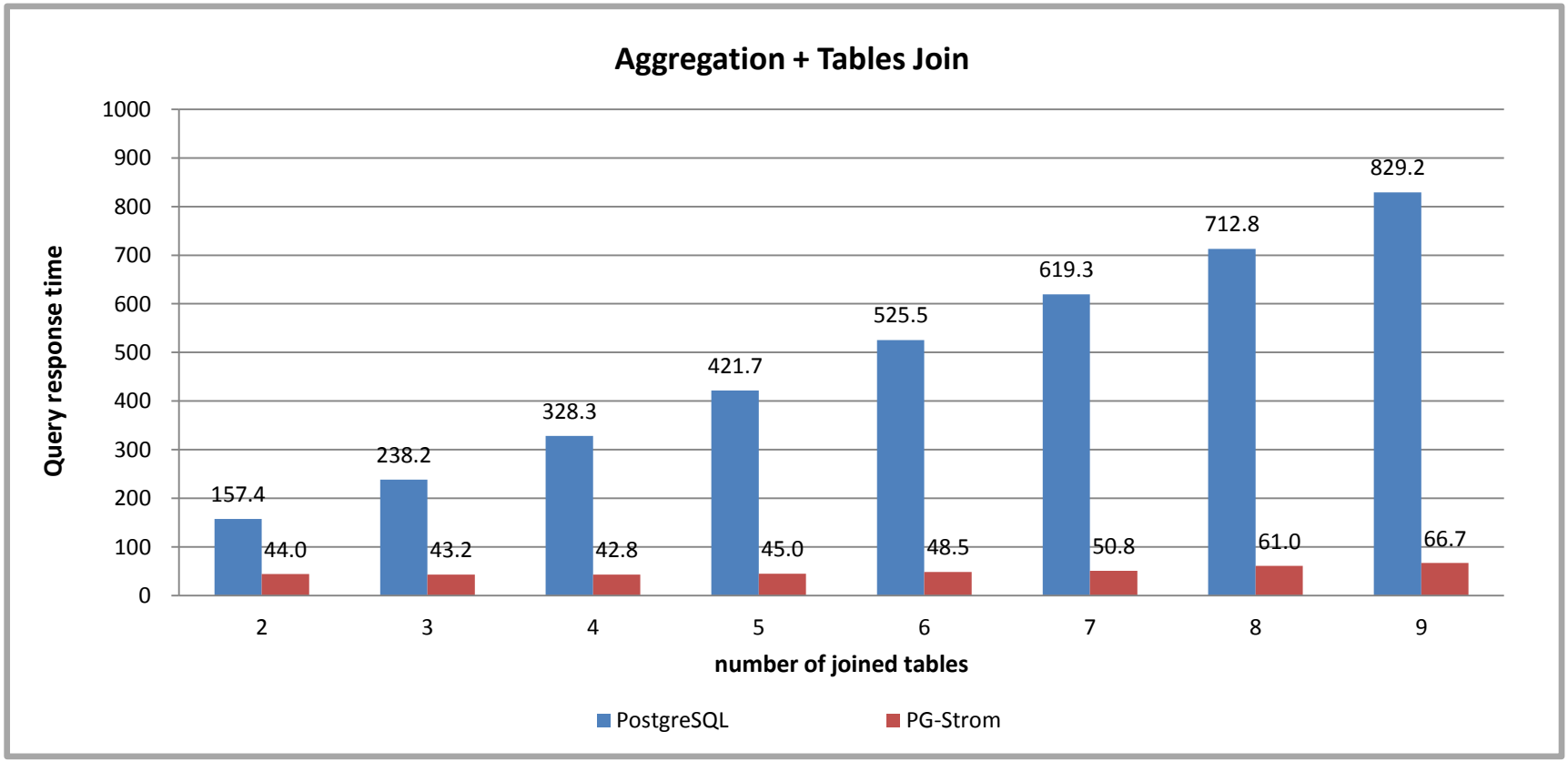
SELECT count(*), AVG(X), Y FROM Tbl_1 GROUP BY Y;





- Aggregate/Sortより前にGpuPreAggを置く事で、CPUが処理する行数を減らす。
- GPU搭載RAMの制約により、テーブル全体に集約演算を施す事は困難だが、数万行～数十万行単位での“部分集約”を作るのは容易。
- 高速化のポイントはCPUが処理しなければならない仕事を減らす事。

ベンチマーク (2/2) – Aggregation + Tables Join



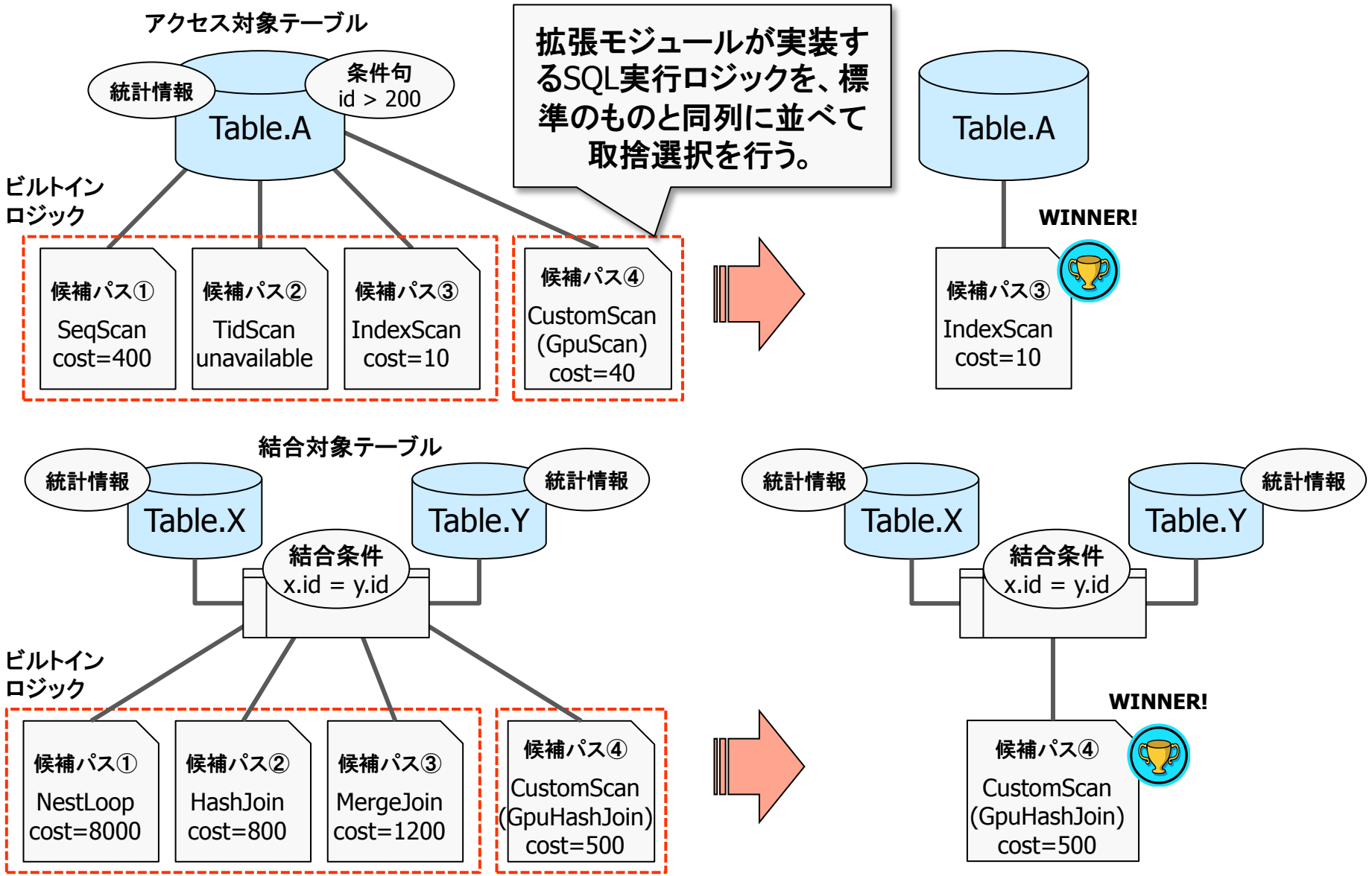
[測定条件]

- 2億件 x 10万件 x 10万件 ... のINNER JOINをテーブル数を変化させながら実行
- 使用したクエリ:

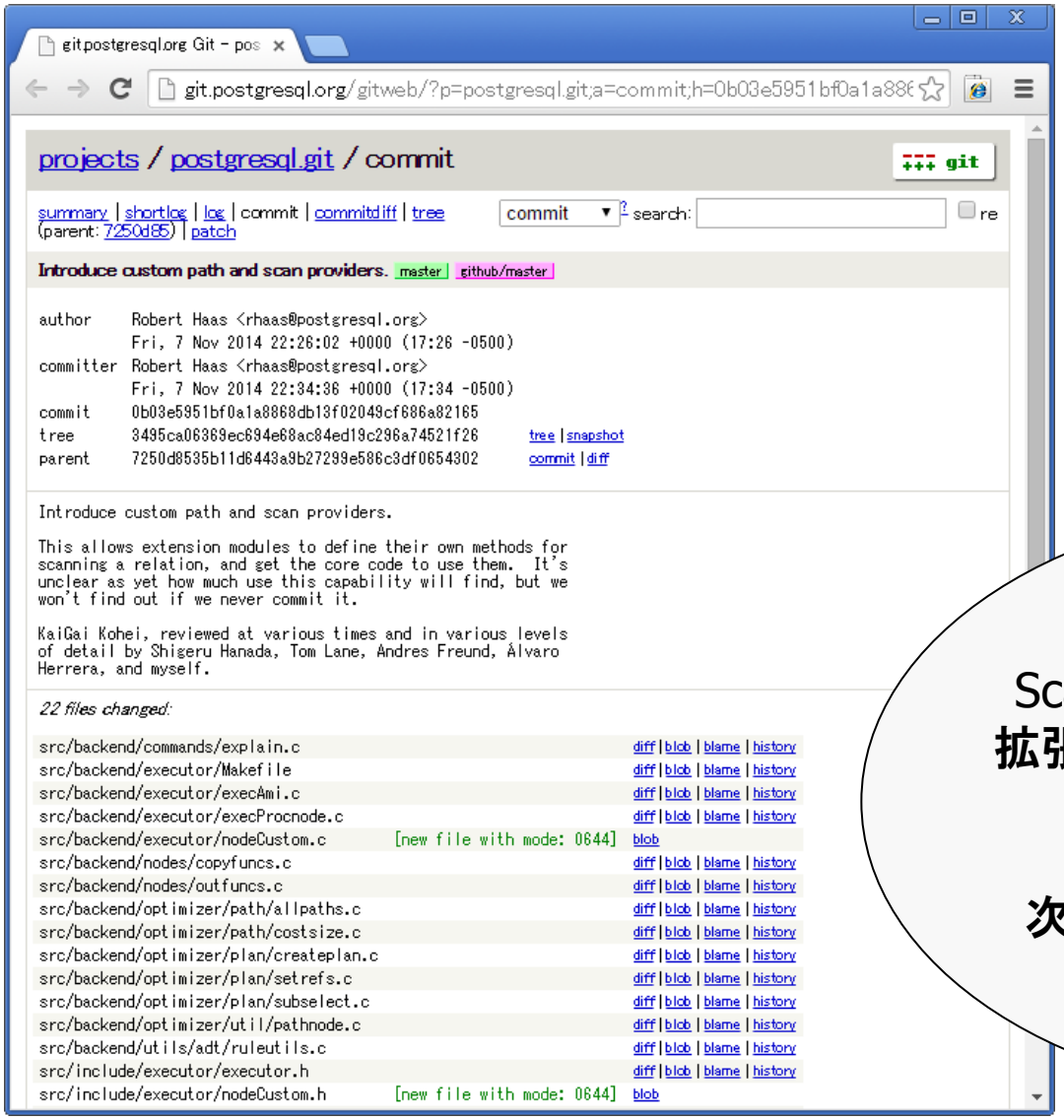
```
SELECT cat, AVG(x) FROM t0 natural join t1 [natural join t2 ...] GROUP BY CAT;
```

- 他の測定条件は前試験と同一

要素技術② – Custom-Scan Interface



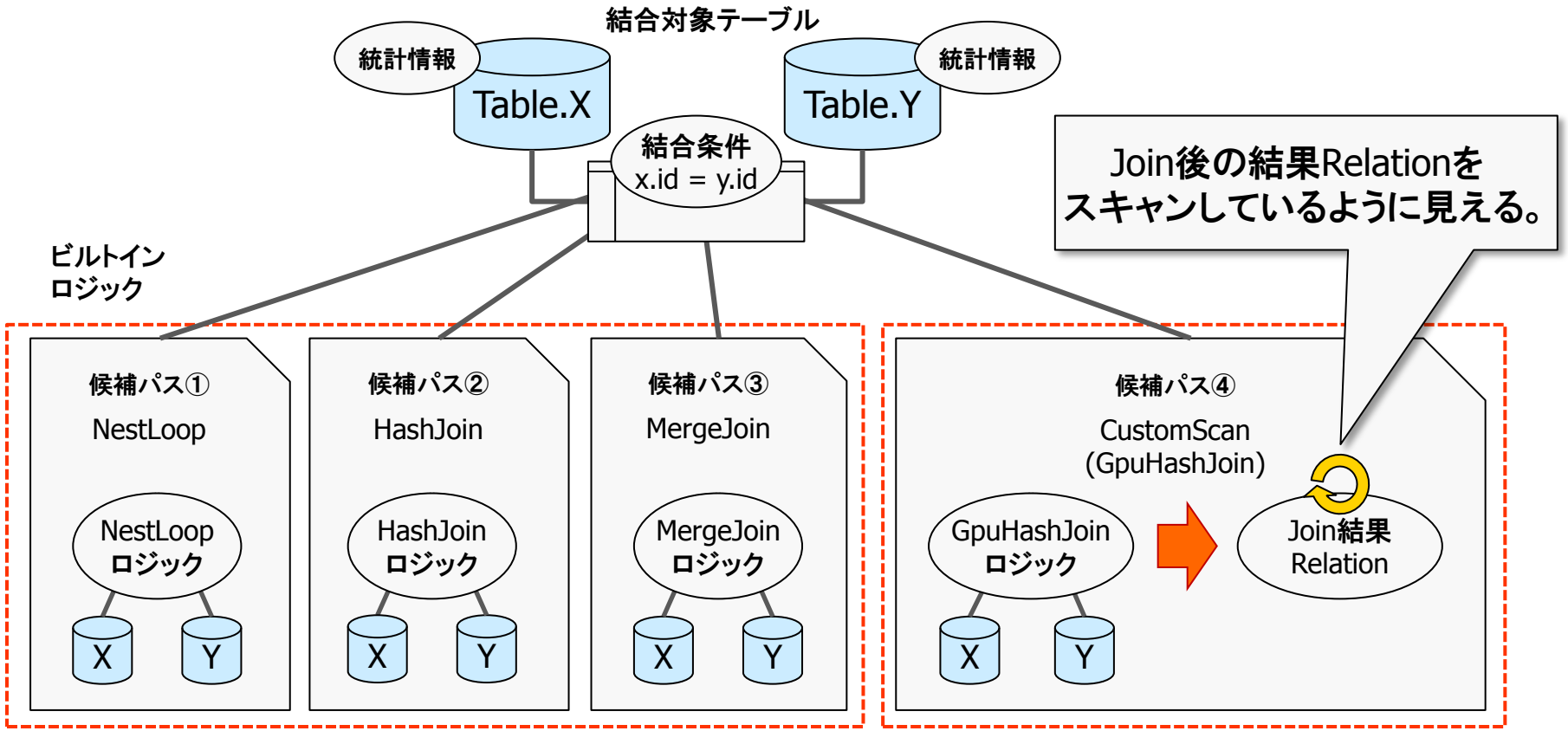
Yes!! Custom-Scan Interface got merged to v9.5



まず、コンセンサスである
Scan部分のカスタムロジックを
拡張モジュールで実装できるよう
インターフェースを定義。
↓
次にJoinのカスタムロジックを
開発者コミュニティで議論

Custom-Scan Interfaceの拡張

Join replacement by foreign-/custom-scan



- Joinが入るべき場所を、Foreign-/Custom-Scanで置き換える。
- 活用例：“Remote Joinの結果”をスキャンするように見えるFDWなど。

Custom-Scan Interfaceの斜め上な使い方

```
postgres=# EXPLAIN SELECT * FROM t0 NATURAL JOIN t1;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Custom (GpuHashJoin)  (cost=2234.00..468730.50 rows=19907950 width=106)
   hash clause 1: (t0.aid = t1.aid)
   Bulkload: On
   ->  Custom (GpuScan) on t0  (cost=500.00..267167.00 rows=20000024 width=73)
   ->  Custom (MultiHash)  (cost=734.00..734.00 rows=40000 width=37)
         hash keys: aid
         Buckets: 46000  Batches: 1  Memory Usage: 99.97%
         ->  Seq Scan on t1  (cost=0.00..734.00 rows=40000 width=37)
 Planning time: 0.220 ms
(9 rows)
```

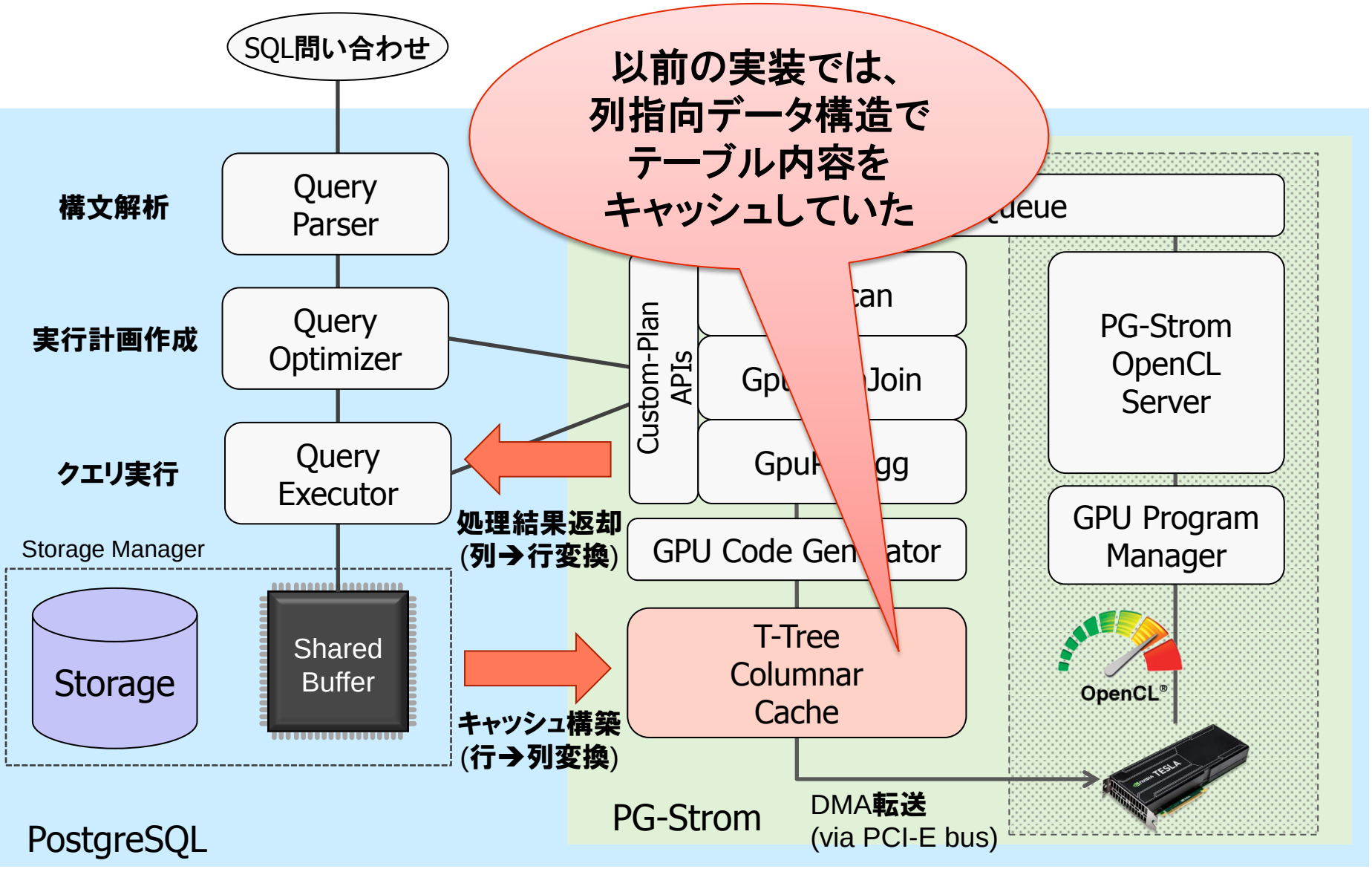
データの流れ

- GpuScanやSeqScanが行を読み出し、上位ノードで処理、さらに上位へ…
- 一行毎に TupleTableSlot を介してデータの受け渡しを行う。

プロトコルに従う必要がないケース

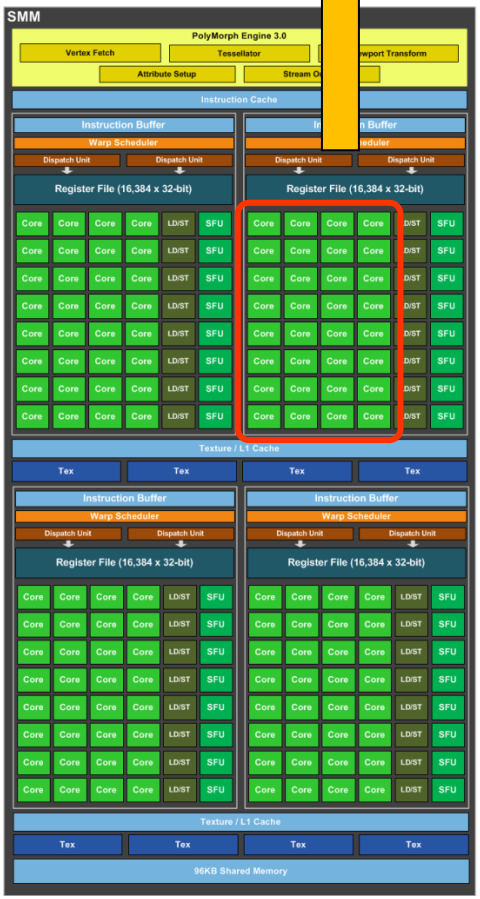
- 親ノード⇔子ノードが同一のモジュールによって管理されている場合。
- 独自のデータ形式を用いてデータの受け渡しを行っても問題ない。
- “Bulkload: On” ➔ 複数行(1万～10万行程度)を一回の呼び出しで受け渡す

要素技術③ – 行指向データ構造



なぜGPUと列指向データの相性が良いか

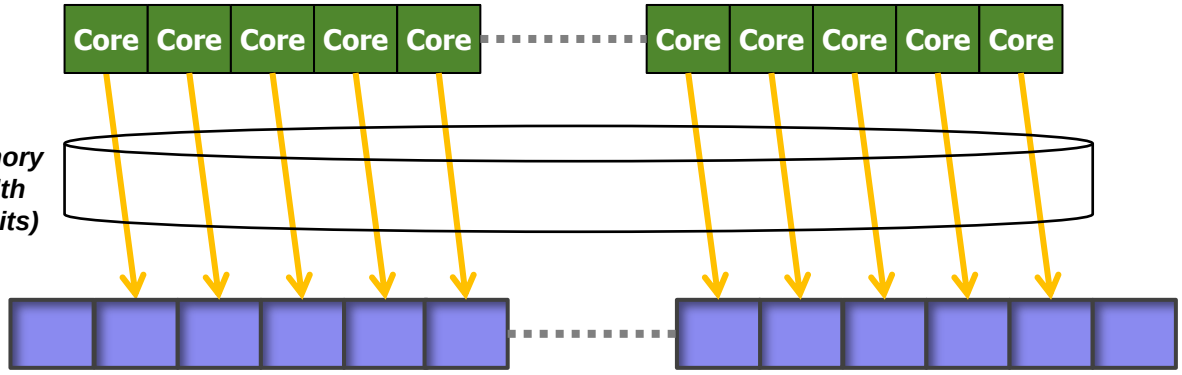
WARP:
命令ポインタを共有する
GPUの命令処理単位。
32個単位である事が多い。



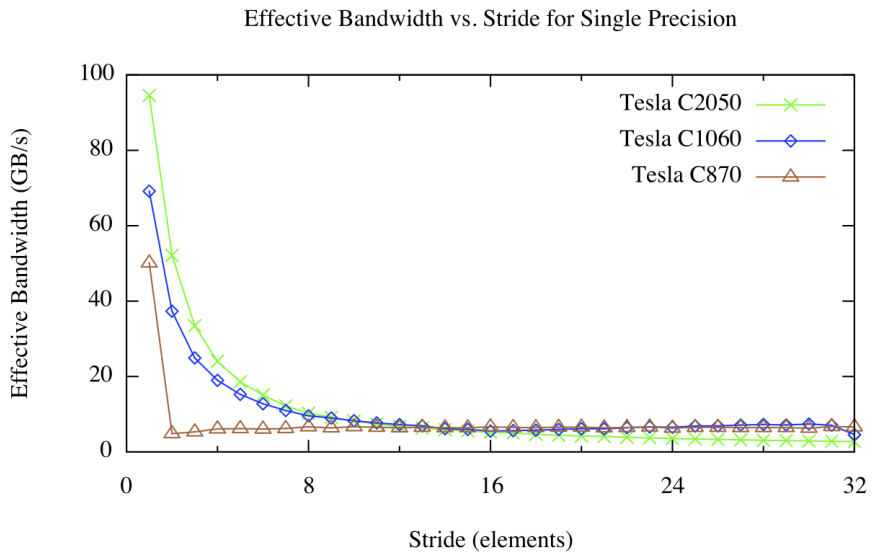
SOURCE: Maxwell: The Most Advanced CUDA GPU Ever Made

coalesced memory access

Wide Memory Bandwidth (256-384bits)



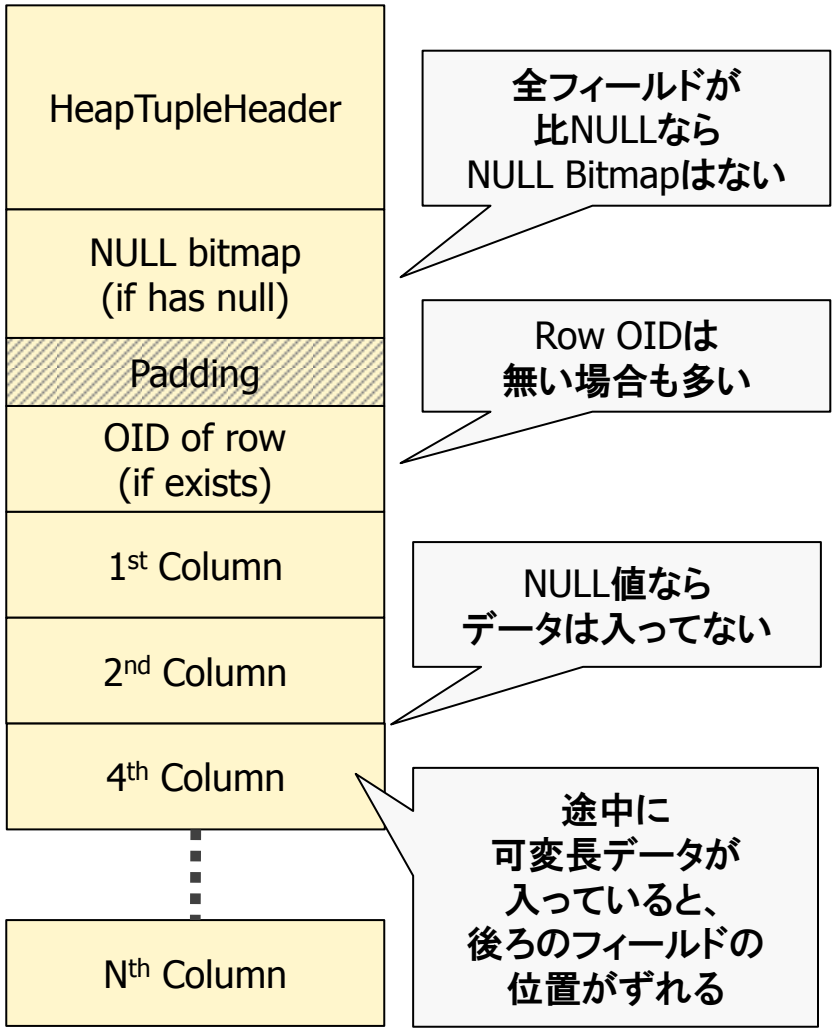
Global Memory (DRAM)



SOURCE: How to Access Global Memory Efficiently in CUDA C/C++ Kernels

PostgreSQLのタプル形式

```
struct HeapTupleHeaderData
{
  union
  {
    HeapTupleFields t_heap;
    DatumTupleFields t_datum;
  }    t_choice;
  /* current TID of this or newer tuple */
  ItemPointerData t_ctid;
  /* number of attributes + various flags */
  uint16       t_infomask2;
  /* various flag bits, see below */
  uint16       t_infomask;
  /* sizeof header incl. bitmap, padding */
  uint8        t_hoff;
  /* ^ - 23 bytes - ^ */
  /* bitmap of NULLs -- VARIABLE LENGTH */
  bits8        t_bits[1];
  /* MORE DATA FOLLOWS AT END OF STRUCT */
};
```



GPU向けとしては、
割と最悪に近いデータ構造

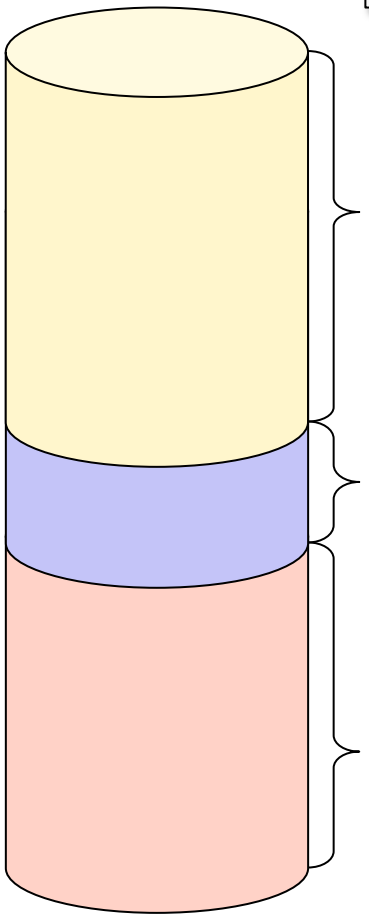
列指向キャッシュの悪夢 (1/2)

```
postgres=# explain (analyze, costs off)
           select * from t0 natural join t1 natural join t2;
                                QUERY PLAN
 ------------------------------------------------------------------------------
Custom (GpuHashJoin) (actual time=54.005..9635.134 rows=20000000 loops=1)
   hash clause 1: (t0.aid = t1.aid)
   hash clause 2: (t0.bid = t2.bid)
   number of requests: 144
   total time to load: 584.67ms
   total time to materialize: 7245.14ms  <-- 全体の70%!!
   average time in send-mq: 37us
   average time in recv-mq: 0us
   max time to build kernel: 1us
   DMA send: 5197.80MB/sec, len: 2166.30MB, time: 416.77ms, count: 470
   DMA recv: 5139.62MB/sec, len: 287.99MB, time: 56.03ms, count: 144
   kernel exec: total: 441.71ms, avg: 3067us, count: 144
   ->  Custom (GpuScan) on t0 (actual time=4.011..584.533 rows=20000000 loops=1)
   ->  Custom (MultiHash) (actual time=31.102..31.102 rows=40000 loops=1)
         hash keys: aid
         ->  Seq Scan on t1 (actual time=0.007..5.062 rows=40000 loops=1)
         ->  Custom (MultiHash) (actual time=17.839..17.839 rows=40000 loops=1)
               hash keys: bid
               ->  Seq Scan on t2 (actual time=0.019..6.794 rows=40000 loops=1)
Execution time: 10525.754 ms
```

列指向キャッシュの悪夢 (2/2)

GPU内のロジックを最適化するために、足回り (= PostgreSQLとのインターフェース部分) で無視できない処理コストが発生している。

処理時間の内訳



テーブル (行指向データ)
⇩
列指向キャッシュへの変換
（最初の一回だけ）

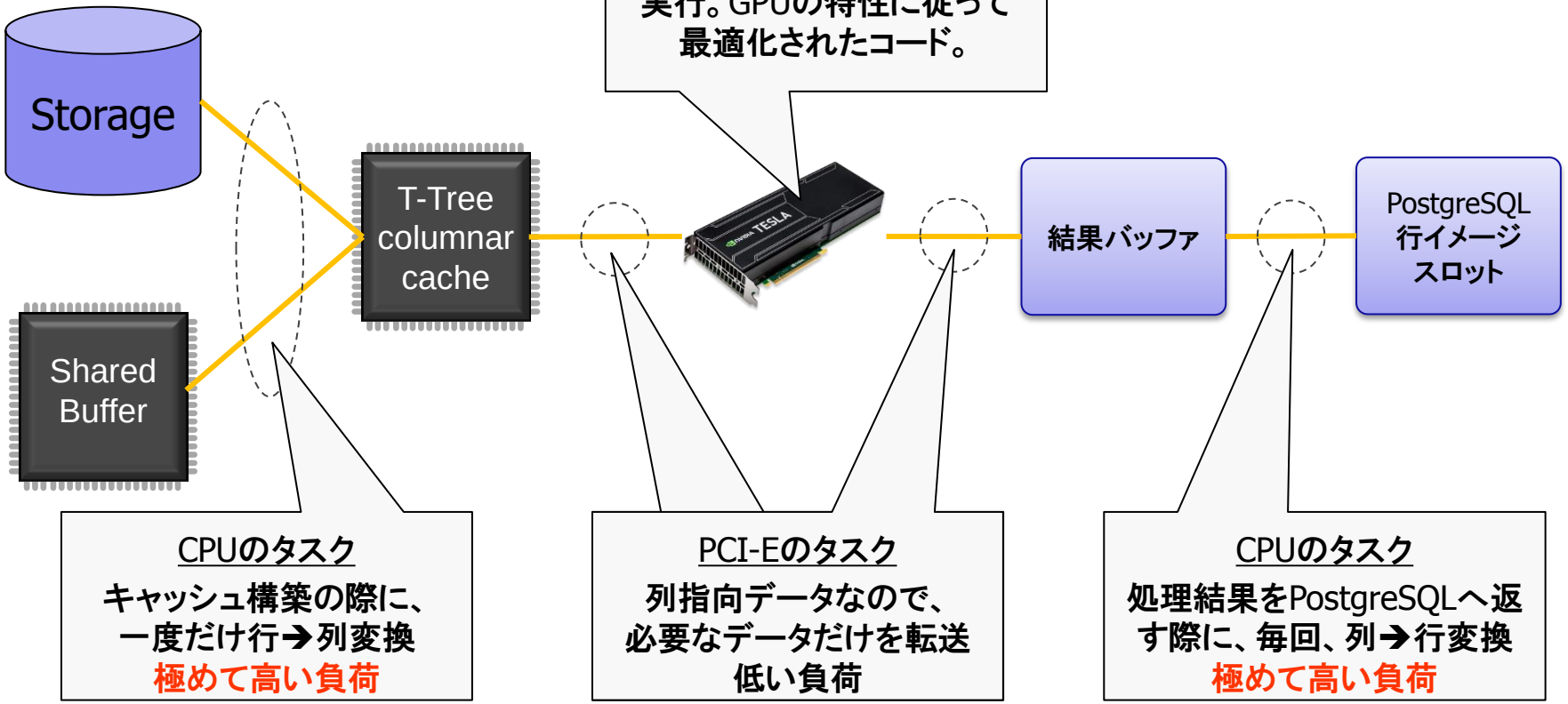
[Hash-Join処理]
テーブル間で対応する
レコードを探索する処理

PG-Strom内部形式
(列指向データ)
⇩
PostgreSQLの
データ受渡し形式
(行指向データ)

GPUアクセラレーションのポイント (1/2)

- CPUは希少リソース。いかにCPUに仕事をさせないかがポイント。
- CPUに仕事をさせる場合、メモリ性能を発揮しやすいパターンを意識する。

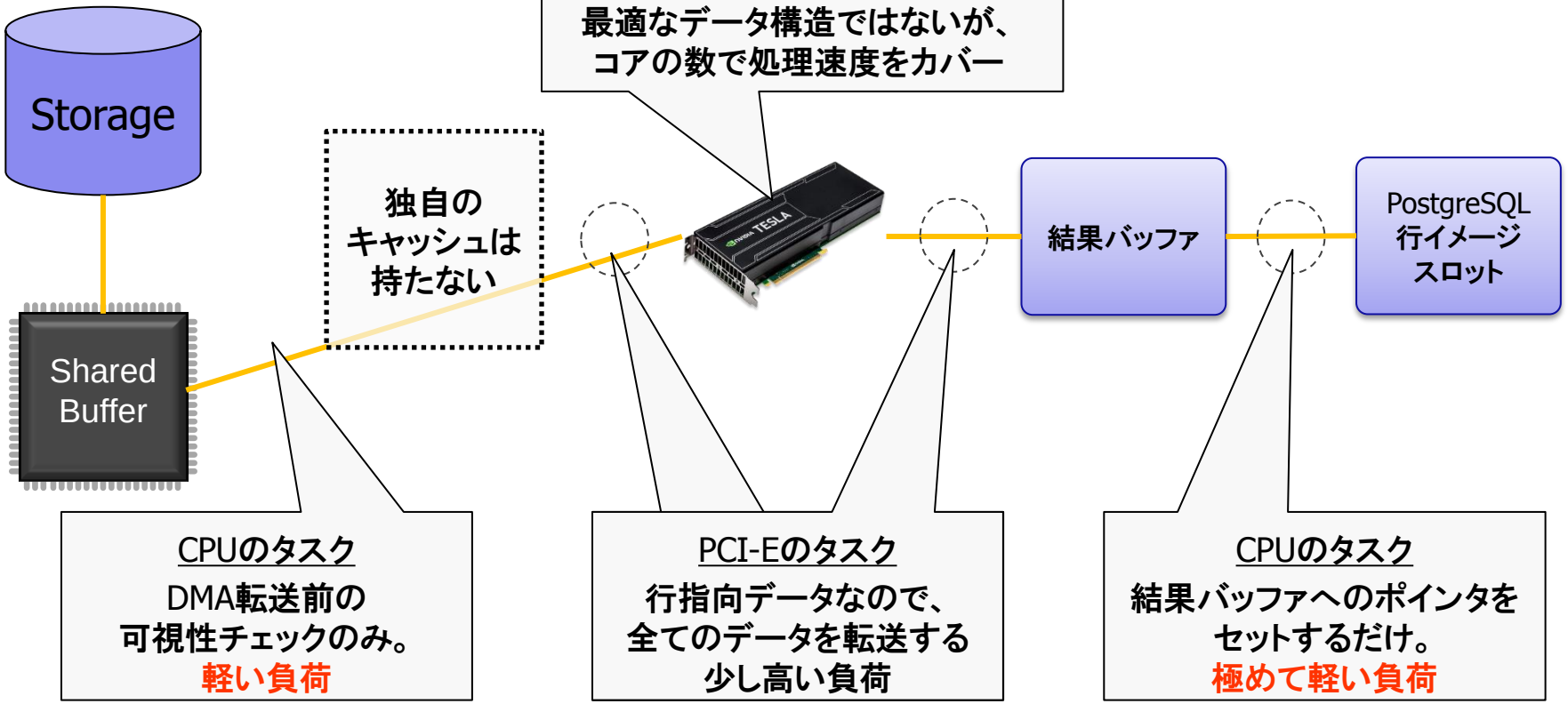
列指向データの利用時



GPUアクセラレーションのポイント (2/2)

- CPUは希少リソース。いかにCPUに仕事をさせないかがポイント。
- CPUに仕事をさせる場合、メモリ性能を発揮しやすいパターンを意識する。

行指向データの利用時



PostgreSQLのshared_bufferと統合した結果

```
postgres=# explain (analyze, costs off)
           select * from t0 natural join t1 natural join t2;
                                    QUERY PLAN
-----------------------------------------------------------------
 Custom (GpuHashJoin) (actual time=111.085..4286.562 rows=20000000 loops=1)
   hash clause 1: (t0.aid = t1.aid)
   hash clause 2: (t0.bid = t2.bid)
   number of requests: 145
   total time for inner load: 29.80ms
   total time for outer load: 812.50ms
   total time to materialize: 1527.95ms  <-- 大幅に削減
   average time in send-mq: 61us
   average time in recv-mq: 0us
   max time to build kernel: 1us
   DMA send: 5198.84MB/sec, len: 2811.40MB, time: 540.77ms, count: 619
   DMA recv: 3769.44MB/sec, len: 2182.02MB, time: 578.87ms, count: 290
   proj kernel exec: total: 264.47ms, avg: 1823us, count: 145
   main kernel exec: total: 622.83ms, avg: 4295us, count: 145
   ->  Custom (GpuScan) on t0 (actual time=5.736..812.255 rows=20000000 loops=1)
   ->  Custom (MultiHash) (actual time=29.766..29.767 rows=80000 loops=1)
         hash keys: aid
         ->  Seq Scan on t1 (actual time=0.005..5.742 rows=40000 loops=1)
         ->  Custom (MultiHash) (actual time=16.552..16.552 rows=40000 loops=1)
               hash keys: bid
               ->  Seq Scan on t2 (actual time=0.022..7.330 rows=40000 loops=1)
 Execution time: 5161.017 ms  <-- 応答性能は大幅改善
```

やや、性能劣化

PG-Stromの現在と今後 (1/2) – 現在

対応済みのロジック

- GpuScan ... 条件句付き全件スキャンのGPU処理
- GpuHashJoin ... Hash-JoinのGPU実装
- GpuPreAgg ... GPUによる集約関数の前処理

対応しているデータ型

- 整数型 (smallint, integer, bigint)
- 浮動小数点型 (real, float)
- 文字列型 (text, varchar(n), char(n))
- 日付時刻型 (date, time, timestamp)
- NUMERIC型

対応している関数

- 各データ型の四則演算オペレータ
- 各データ型の大小比較オペレータ
- 浮動小数点型の数値計算関数
- 集約演算: MIN, MAX, SUM, AVG, 標準偏差, 分散, 共分散

PG-Stromの現在と今後 (2/2) – 今後

今後対応するロジック

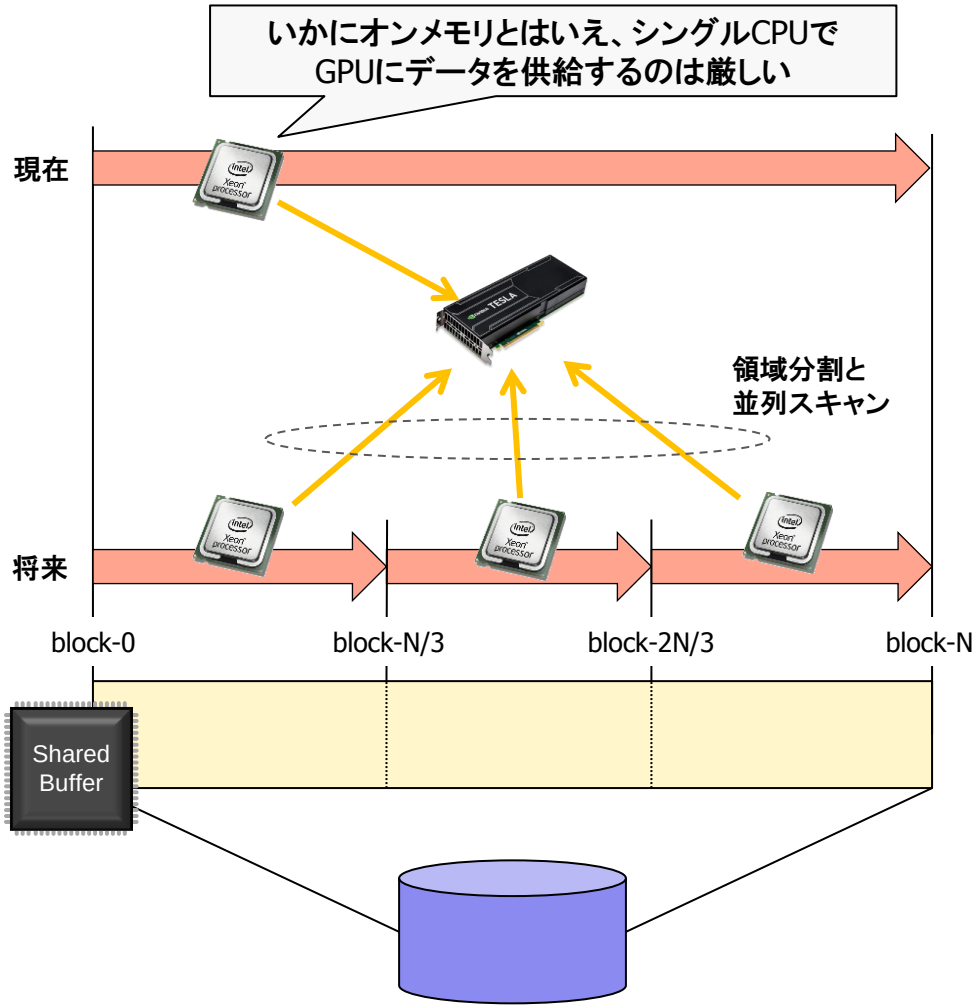
- Outer Join
- Sort
- Aggregate Push-down
- … その他？

対応するデータ型？

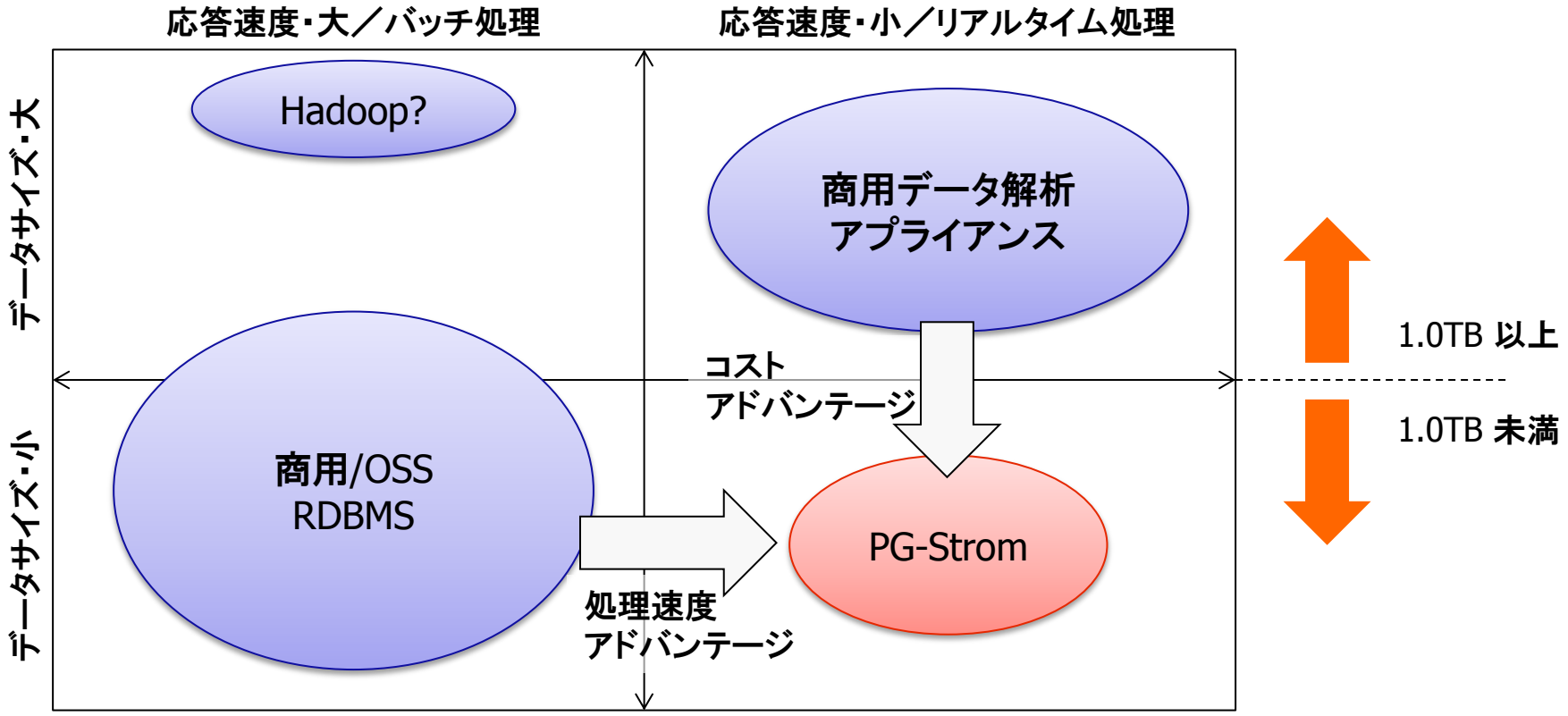
対応する関数？

- LIKE句、正規表現？
- 日付/時刻関数
- PostGIS関数？？
- 幾何関数？
- ユーザ定義関数？

足回りの強化

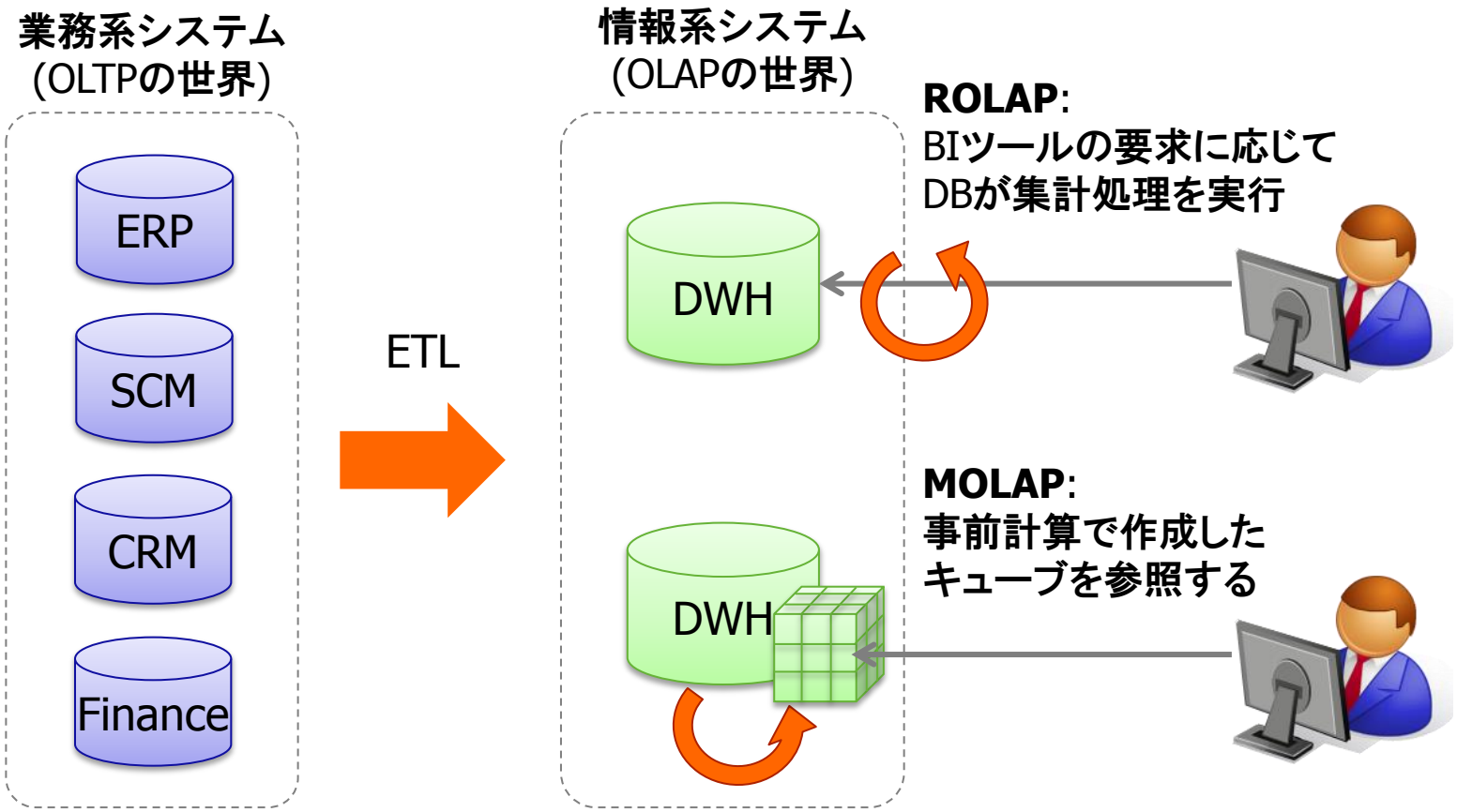


どういう領域で使いたいか



- アプリケーション： 1st ターゲットとしてBIツールを想定
- データサイズは高々1.0TB未満、中堅企業／大企業の部門
  - 現状、In-memoryでデータを持たねばならないため
- GPU と PostgreSQL、どちらも安価な組み合わせ。

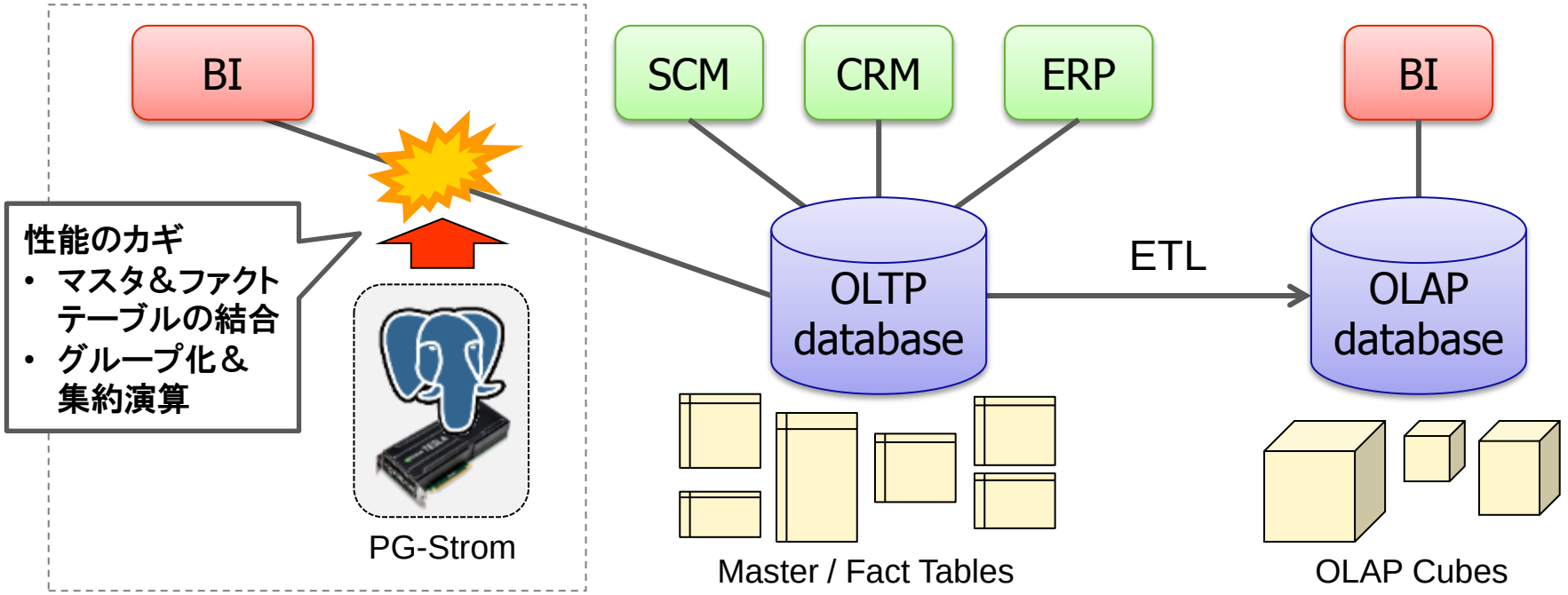
(参考) ROLAPとMOLAP



PG-Stromの利用シーンは

- ROLAP向けバックエンドRDBMS / 応答速度高速化
- MOLAP向けキューブ作成処理 / バッチ処理高速化

想定利用シーン (1/2) – OLTP/OLAP統合



OLTP/OLAPデータベースを分ける理由

- 複数のデータソースを統合する
- 参照系ワークロードへの最適化

PG-Stromの適用により…

- 性能のカギとなるJoinやAggregateを並列処理、リアルタイム化
- OLAPとETLが不要とする事で、システムコストを圧縮

想定利用シーン (2/2) – 一緒にトライしませんか？

- どういった領域に適用可能だろうか？
- どういった用途で使えるだろうか？
- どういったワークロードに困っているだろうか？

➔ PG-Stromプロジェクトはフィールドから学びたいと考えています

PG-Strom Roadmap

How to use (1/3) – インストールに必要なのは

- OS: Linux (RHEL 6.x で動作確認)
- PostgreSQL 9.5devel (with Custom-Plan Interface)
- PG-Strom 拡張モジュール
- OpenCL ドライバ (NVIDIAランタイムなど)

PG-Stromを使用するために最低限必要な設定

```
shared_preload_libraries = '$libdir/pg_strom'
shared_buffers = <DBサイズと同程度>
```

実行時に PG-Strom の有効/無効を切り替える

```
postgres=# SET pg_strom.enabled = on;
SET
```

How to use (2/3) – ビルド、インストール、起動

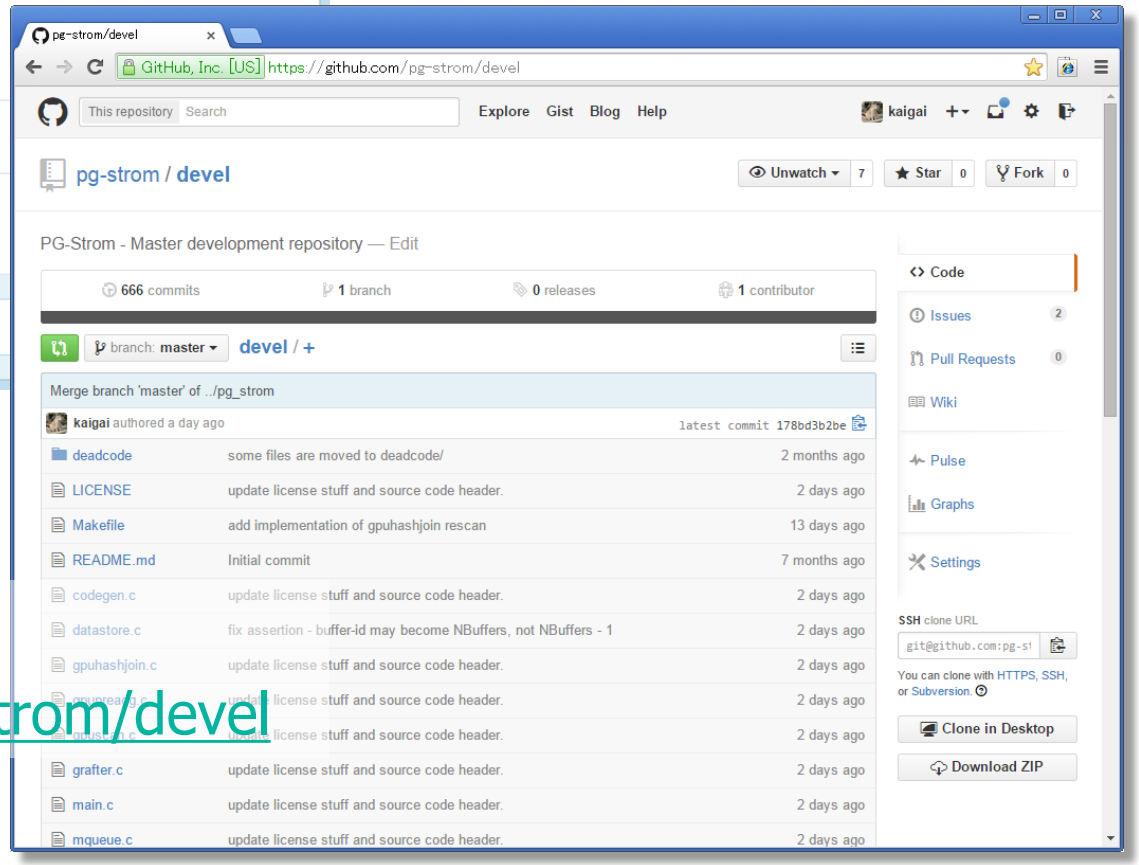
```
[kaigai@saba ~]$ git clone https://github.com/pg-strom/devel.git pg_strom
[kaigai@saba ~]$ cd pg_strom
[kaigai@saba pg_strom]$ make && make install
[kaigai@saba pg_strom]$ vi $PGDATA/postgresql.conf
```

```
[kaigai@saba ~]$ pg_ctl start
server starting
[kaigai@saba ~]$ LOG:  registering background worker "PG-Strom OpenCL Server"
LOG:  starting background worker process "PG-Strom OpenCL Server"
LOG:  database system was shut down at 2014-11-09 17:45:51 JST
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections
LOG:  PG-Strom: [0] OpenCL Platform: NVIDIA CUDA
LOG:  PG-Strom: (0:0) Device GeForce GTX 980 (1253MHz x 16units, 4095MB)
LOG:  PG-Strom: (0:1) Device GeForce GTX 750 Ti (1110MHz x 5units, 2047MB)
LOG:  PG-Strom: [1] OpenCL Platform: Intel(R) OpenCL
LOG:  PG-Strom: Platform "NVIDIA CUDA (OpenCL 1.1 CUDA 6.5.19)" was installed
LOG:  PG-Strom: Device "GeForce GTX 980" was installed
LOG:  PG-Strom: shmem 0x7f447f6b8000-0x7f46f06b7fff was mapped (len: 10000MB)
LOG:  PG-Strom: buffer 0x7f34592795c0-0x7f44592795bf was mapped (len: 65536MB)
LOG:  Starting PG-Strom OpenCL Server
LOG:  PG-Strom: 24 of server threads are up
```

How to use (3/3) – AWSで楽々デプロイ

AWS GPUインスタンス仕様 (g2.2xlarge)	
CPU	Xeon E5-2670 (8 xCPU)
RAM	15GB
GPU	NVIDIA GRID K2 (1536core)
Storage	60GB of SSD
Price	$0.898/hour、$646.56/mon (*) 2014年11月8日現在の東京リージョン オンデマンドインスタンス価格

補足① – 自分で試してみたい人へ



check it out!

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

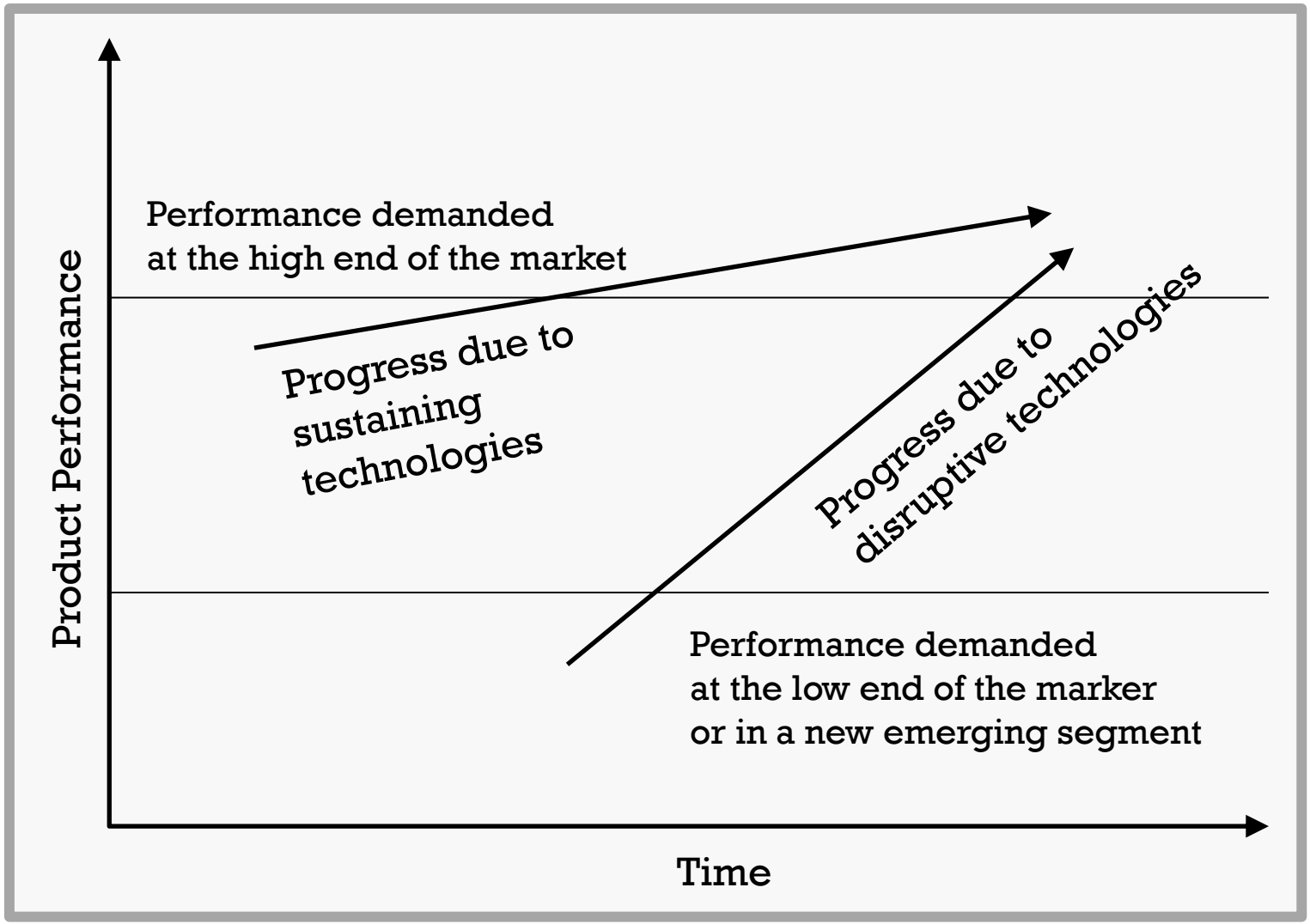
補足② – 一緒に試してみたい人へ

- PG-Stromの検証環境を用意しました。
- 一定の条件下で、共同検証に無償利用する事ができます。
  - ✓ ウチのシステムのSQLが早くなるかどうか試してみたい。
  - ✓ こんな使い道だと効果が大きそう。試してみたい。

PG-Strom検証環境	
CPU	Intel Xeon E5-2670 v3 (12C, 2.3GHz) x2
RAM	768GB
GPU	NVIDIA Tesla K20C x1 NVIDIA GTX980 x1 NVIDIA GTX750Ti x1
HDD	2.1TB (300GB SAS 10krpm, x8; RAID5)
OS	Red Hat Enterprise Linux 6.x
DB	PostgreSQL 9.5devel + PG-Strom
その他	**応相談**

➔興味がある方は pg-strom@bid.jp.nec.com まで

イノベーションのジレンマ



SOURCE: The Innovator's Dilemma, Clayton M. Christensen

コミュニティと共に進む

Orchestrating a brighter world

世界の想いを、未来へつなげる。

未来に向かい、人が生きる、豊かに生きるために欠かせないもの。
それは「安全」「安心」「効率」「公平」という価値が実現された社会です。

NECは、ネットワーク技術とコンピューティング技術をあわせ持つ
類のないインテグレーターとしてリーダーシップを発揮し、
卓越した技術とさまざまな知見やアイデアを融合することで、
世界の国々や地域の人々と協奏しながら、
明るく希望に満ちた暮らしと社会を実現し、未来につなげていきます。

Empowered by Innovation