

PG-Strom v2.0 Release
Technical Brief
(17-Apr-2018)

PG-Strom Development Team
<pgstrom@heterodb.com>

What is PG-Strom?

PG-Strom: an extension module to accelerate analytic SQL workloads using GPU.



- ✓ Designed as PostgreSQL extension
- ✓ Transparent SQL acceleration
- ✓ Cooperation with fully transactional database management system
- ✓ Various comprehensive tools and applications for PostgreSQL

[For Advanced Analytics workloads]

- ❑ CPU+GPU hybrid parallel
- ❑ PL/CUDA user defined function
- ❑ GPU memory store (Gstore_fdw)



[For I/O intensive workloads]

- ❑ SSD-to-GPU Direct SQL Execution
- ❑ In-memory Columnar Cache
- ❑ SCAN+JOIN+GROUP BY combined GPU kernel



[GPU's characteristics]

- ✓ **Several thousands of processor cores per device**
- ✓ **Nearly terabytes per second memory bandwidth**
- ✓ **Much higher cost performance ratio**

PG-Strom is an open source extension module for PostgreSQL (v9.6 or later) to accelerate analytic queries, has been developed and incrementally improved since 2012. It provides PostgreSQL alternative query execution plan for SCAN, JOIN and GROUP BY workloads. Once optimizer chose the custom plans which use GPU for SQL execution, PG-Strom constructs relative GPU code on the fly. It means PostgreSQL uses GPU aware execution engine only when it makes sense, so here is no downside for transactional workloads.

PL/CUDA is a derivational feature that allows manual optimization with user defined function (UDF) by CUDA C; which shall be executed on GPU device. The PG-Strom v2.0 added many features to enhance these basis for more performance and wider use scenarios.

PG-Strom v2.0 features highlight

Storage Enhancement

- ❑ SSD-to-GPU Direct SQL Execution
- ❑ In-memory Columnar Cache
- ❑ GPU memory store (gstore_fdw)

Advanced SQL Infrastructure

- ❑ PostgreSQL v9.6/v10 support – CPU+GPU Hybrid Parallel
- ❑ SCAN+JOIN+GROUP BY combined GPU kernel
- ❑ Utilization of demand paging of GPU device memory

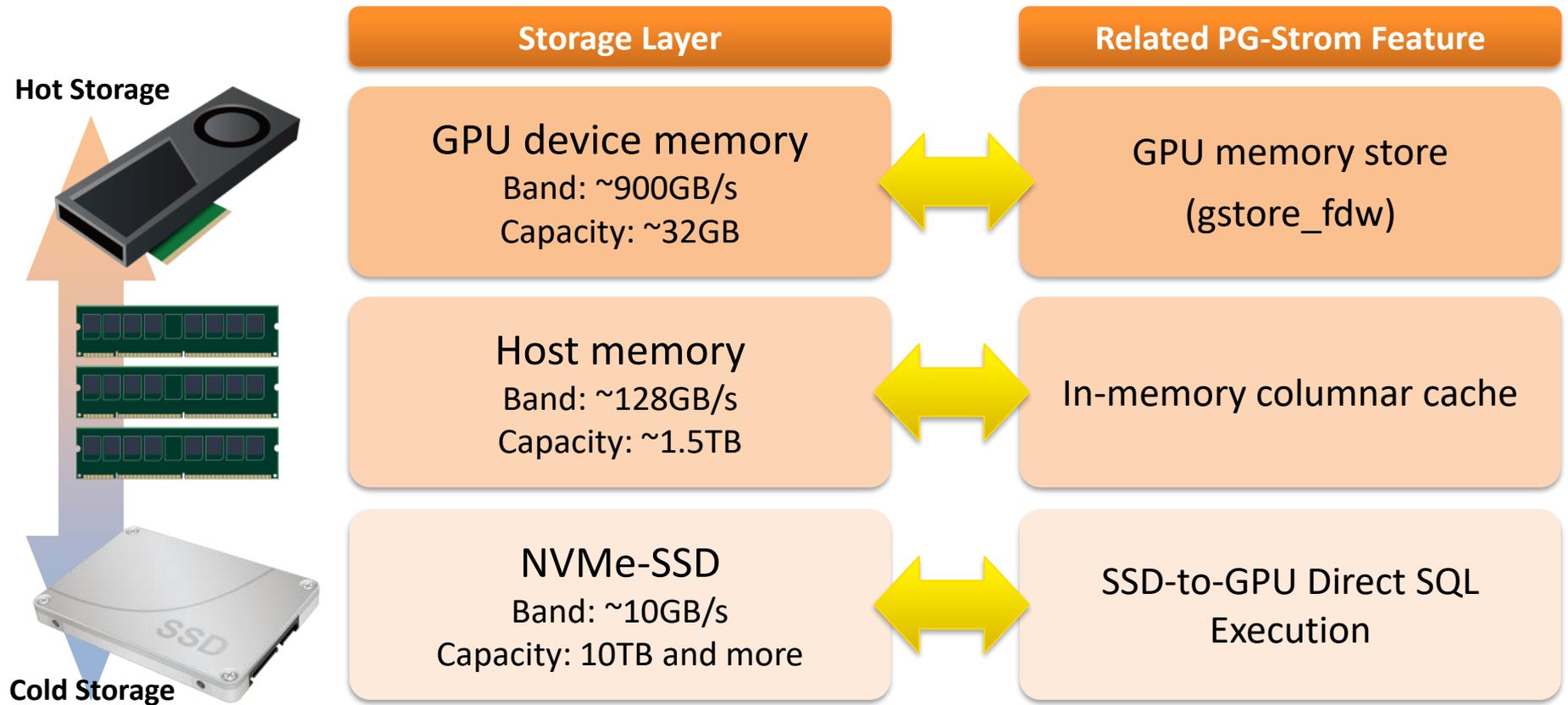
Miscellaneous

- ❑ PL/CUDA related enhancement
- ❑ New data type support
- ❑ Documentation and Packaging



Storage Enhancement

Storage enhancement for each layer



In general, key of performance is not only number of cores and its clock, but data throughput to be loaded for the processors also.

Individual storage layer has its own characteristics, thus PG-Strom provides several options to optimize the supply of data.

SSD-to-GPU Direct SQL Execution is unique and characteristic feature of PG-Strom. It directly loads the data blocks of PostgreSQL to GPU, and runs SQL workloads to reduce amount of data to be processed by CPU prior to the arrival. Its data transfer bypasses operating system software stacks, thus allows to pull out nearly wired performance of the hardware. **In-memory columnar cache** allows to keep data blocks in the optimal format for GPU to compute and transfer over the PCIe bus. **GPU memory store (gstore_fdw)** allows preload on the GPU device memory using standard SQL statement. It is an ideal data location for PL/CUDA function because it does not need to carry the data set for each invocation, and no size limitation of 1GB which is maximum length of the varlena.

SSD-to-GPU Direct SQL Execution (1/3)

Pre-processing of SQL workloads to drop unnecessary rows prior to the data loading to CPU

SQL optimization stage

```
SELECT cat, count(*), avg(X)
FROM t0 JOIN t1 ON t0.id = t1.id
WHERE YMD >= 20120701
GROUP BY cat;
```

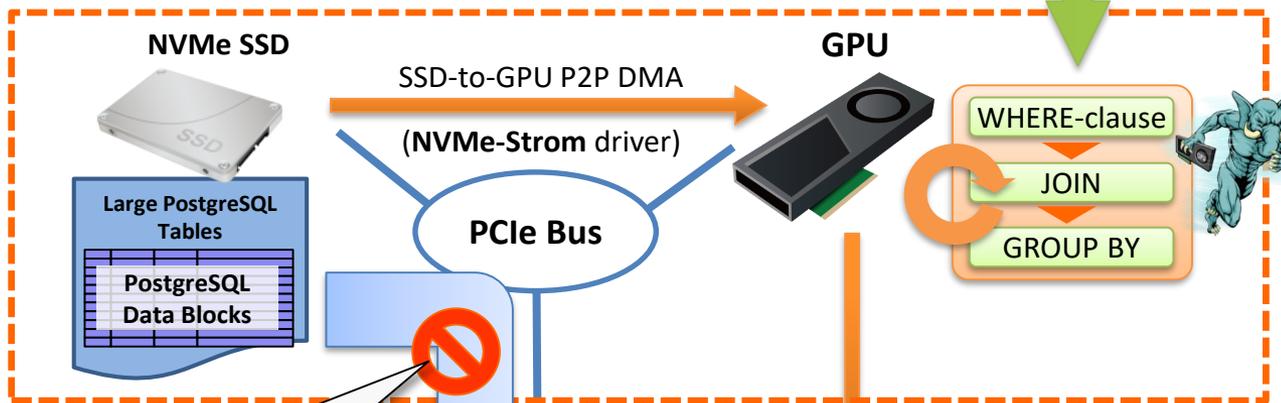
SQL-to-GPU
Program
Generator

GPU binary
Just-in-time
Compile

GPU code generation

PG-Strom automatically generate CUDA code. It looks like a transparent acceleration from the standpoint of users.

SQL execution stage



Traditional data flow

Even if unnecessary records, only CPU can determine whether these are necessary or not. So, we have to move any records including junks.

SSD-to-GPU Direct SQL

Once data blocks are loaded to GPU, we can process SQL workloads using thousands cores of GPU. It reduces the amount of data to be loaded and executed by CPU; looks like I/O performance acceleration.

SQL execution on GPU

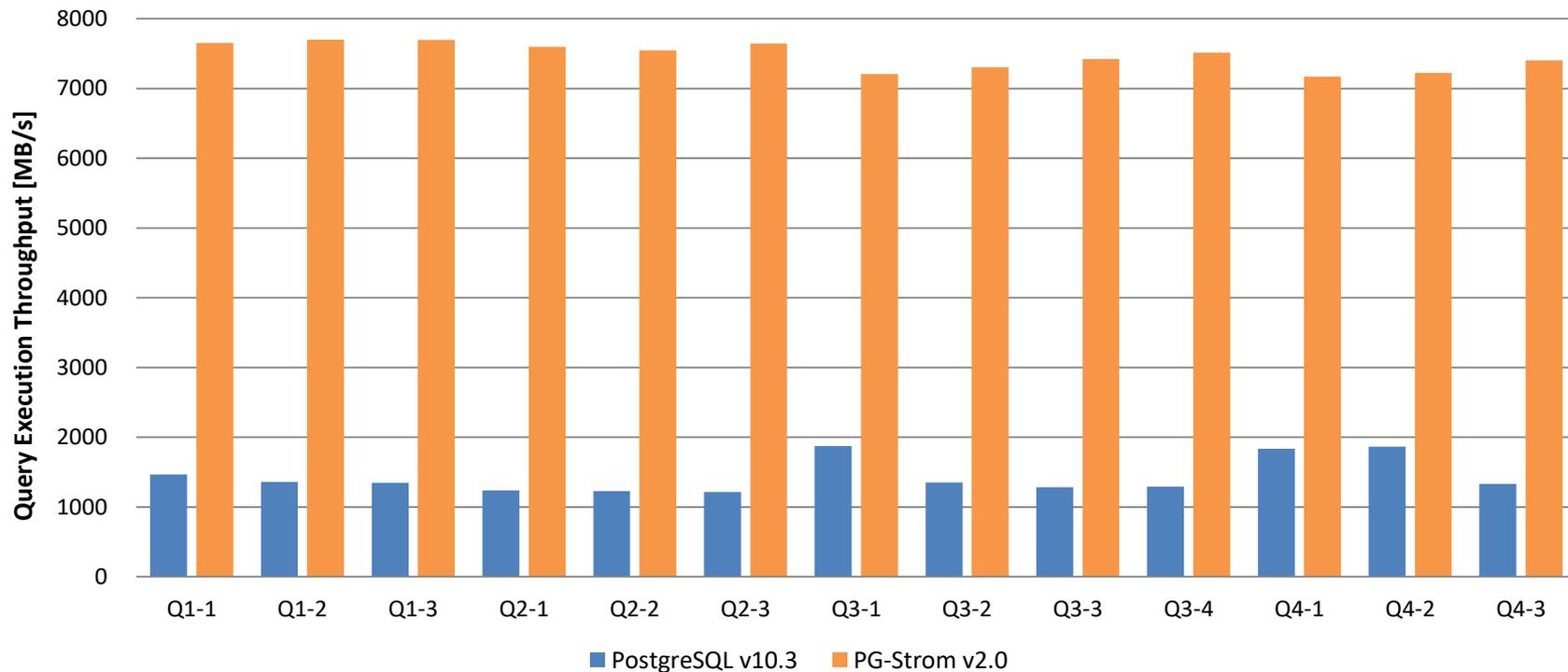
It loads data blocks of PostgreSQL to GPU using peer-to-peer DMA, then drops unnecessary data with parallel SQL execution by GPU.

SQL execution on CPU

CPU runs pre-processed data set; that is much smaller than the original. Eventually, it looks like GPU accelerated I/O also.

SSD-to-GPU Direct SQL Execution (2/3)

Star Schema Benchmark results on NVMe-SSDx3 with md-raid0



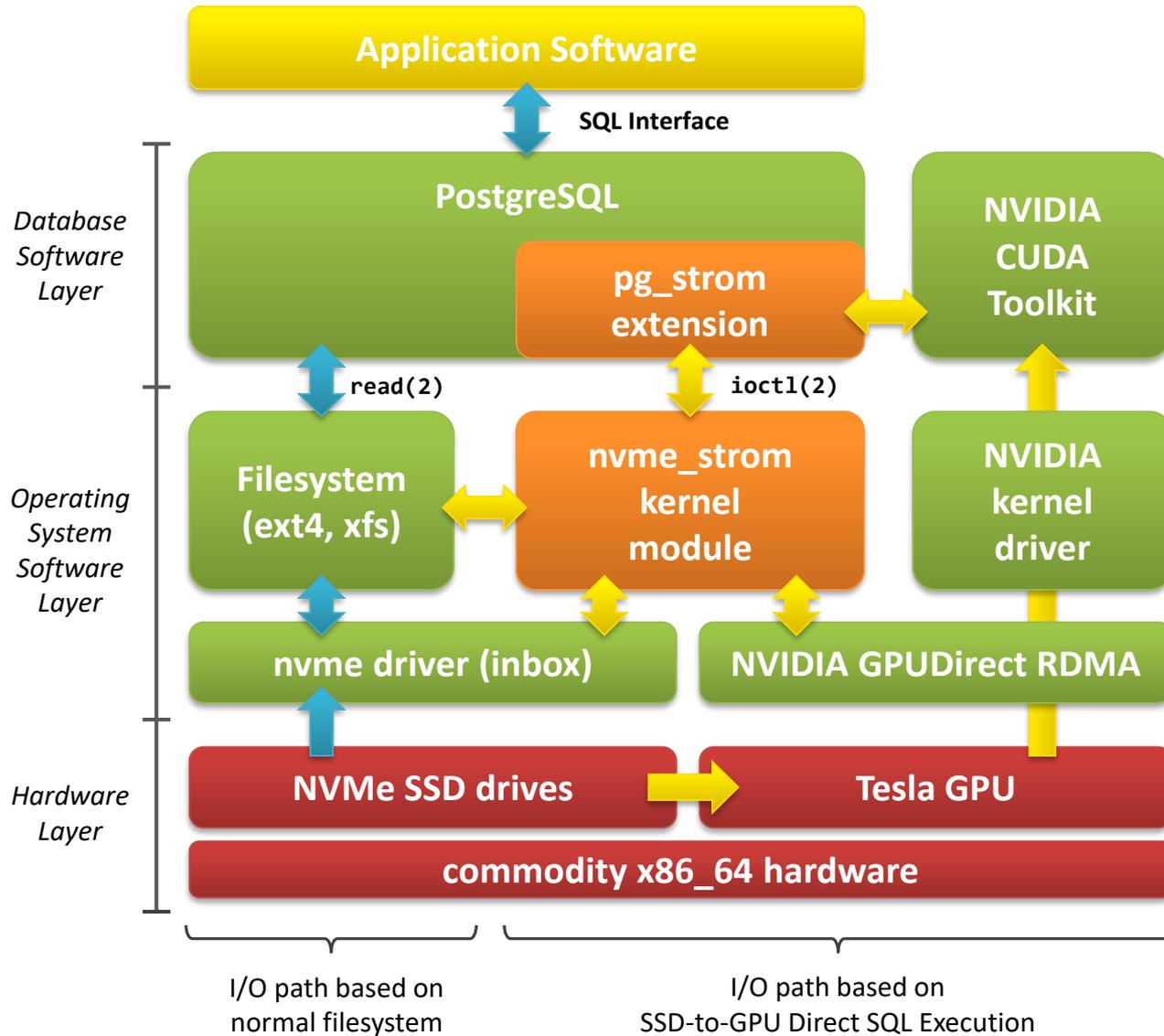
This result shows query execution throughput on 13 different queries of the Star Schema Benchmark. Host system mounts just 192GB memory but database size to be scanned is 353GB. So, it is quite i/o intensive workload.

PG-Strom with SSD-to-GPU Direct SQL Execution shows 7.2-7.7GB/s throughput which is more than x3.5 faster than the vanilla PostgreSQL for large scale batch processing. The throughput is calculated by (database size) / (query response time). So, average query response time of PG-Strom is later half of the 40s, and PostgreSQL is between 200s to 300s.

Benchmark environment:

Server: Supermicro 1019GP-TT, CPU: Intel Xeon Gold 6126T (2.6GHz, 12C), RAM: 192GB, GPU: NVIDIA Tesla P40 (3840C, 24GB), SSD: Intel DC P4600 (HHHL, 2.0TB) x3, OS: CentOS 7.4, SW: CUDA 9.1, PostgreSQL v10.3, PG-Strom v2.0

SSD-to-GPU Direct SQL Execution (3/3)



SSD-to-GPU Direct SQL Execution is a technology built on top of NVIDIA GPUDirect RDMA which allows P2P DMA between GPU and 3rd party PCIe devices.

The **nvme_strom** kernel module intermediates P2P DMA from NVMe-SSD to Tesla GPU [*1].

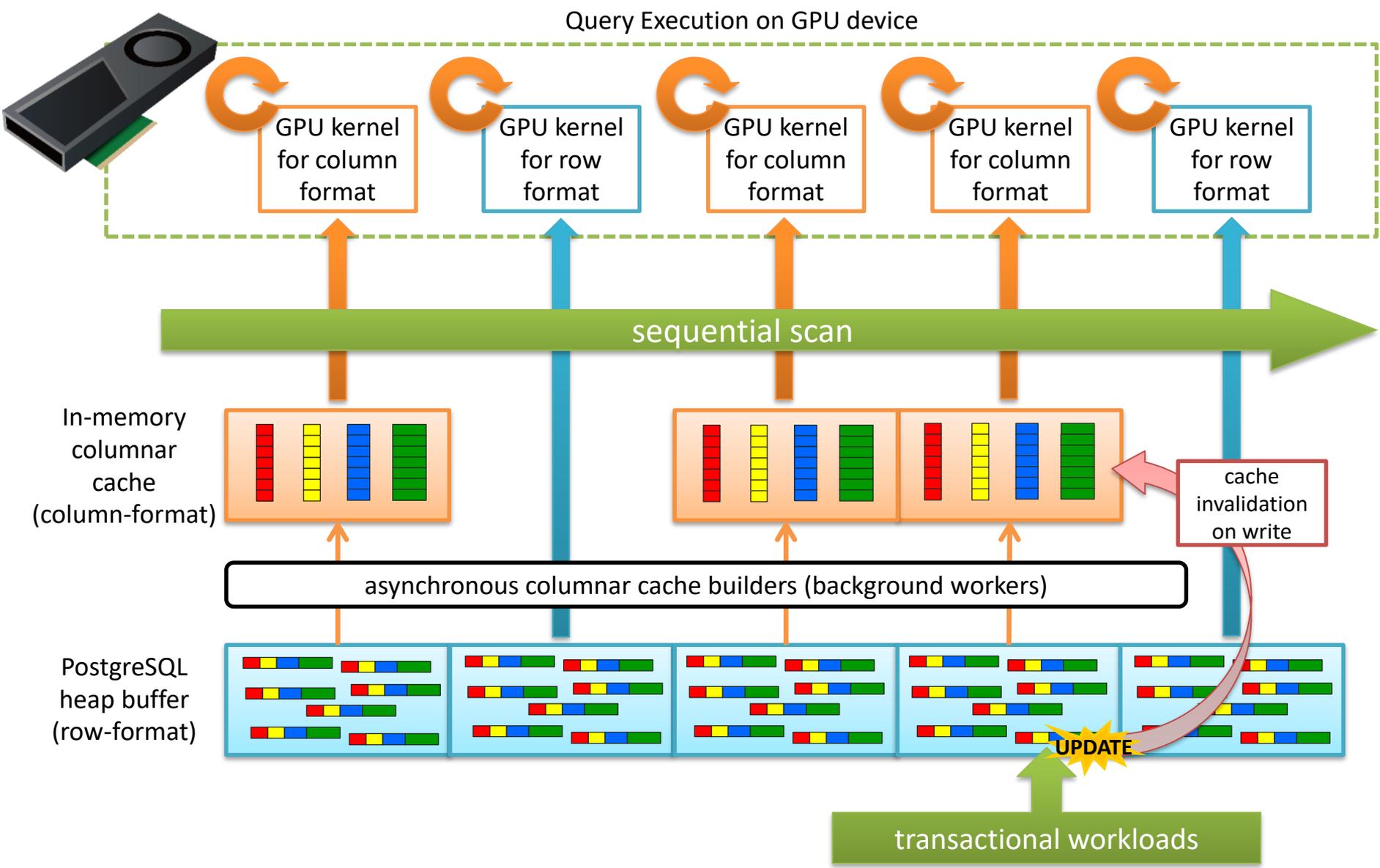
Once GPU accelerated query execution plan gets chosen by the query optimizer, then PG-Strom calls **ioctl(2)** to deliver the command of SSD-to-GPU Direct SQL Execution to **nvme_strom** in the kernel space.

This driver has small interaction with filesystem to convert file descriptor + file offset to block numbers on the device. So, only limited filesystems (Ext4, XFS) are now supported.

You can also use striping of NVMe-SSD using md-raid0 for more throughput, however, it is a feature of commercial subscription. Please contact HeteroDB, if you need multi-SSDs grade performance.

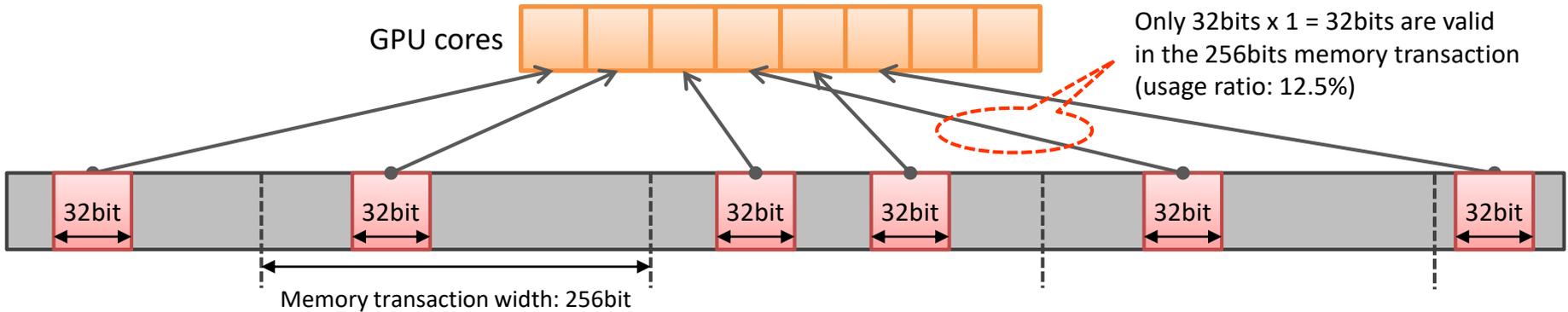
[*1] NVIDIA GPUDirect RDMA is available on only Tesla or Quadro, not GeForce.

In-memory Columnar Cache

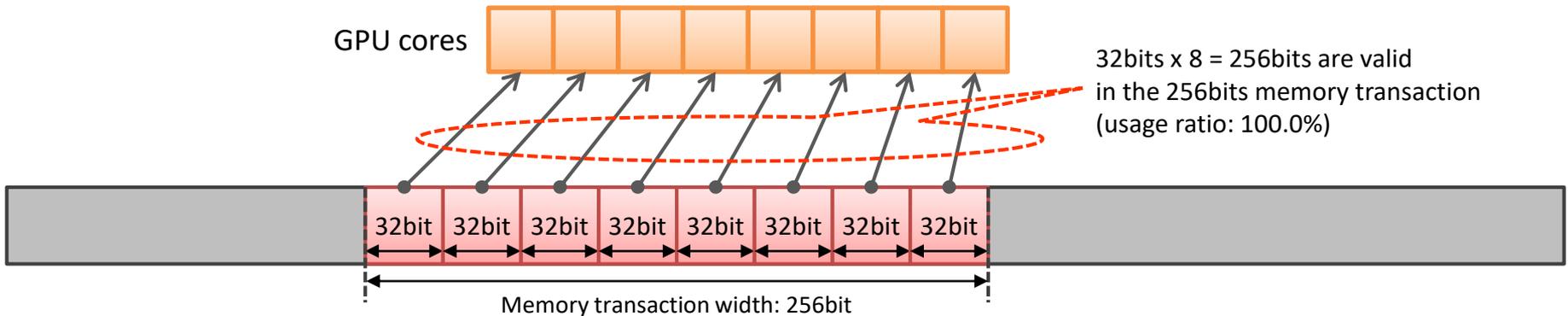


(Background) Why columnar-format is preferable for GPU

Row-format – *random memory access*



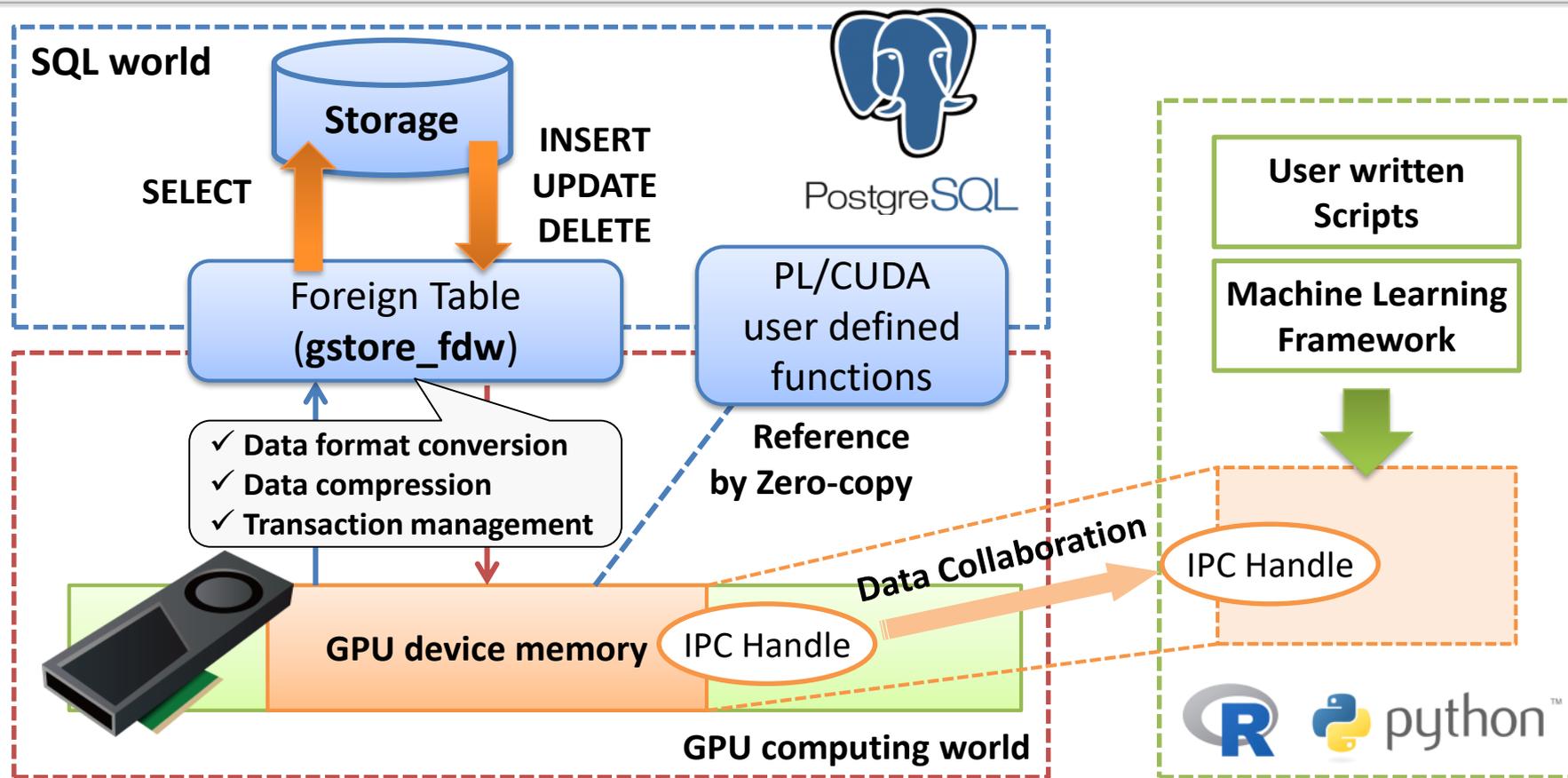
Column-format – *coalesced memory access*



Memory access pattern affects GPU's performance so much because of its memory sub-system architecture. GPU has relatively larger memory transaction width. If multiple co-operating cores simultaneously read continuous items in array, one memory transaction can load eight 32bit values (if width is 256bit) at once. It fully depends on the data format, and one of the most significant optimization factor.

Row-format tends to put values of a particular column to be scanned on random place, not continuous, thus it is hard to pull out maximum performance of GPU. **Column-format** represents a table like as a set of simple array. So, when column-X is referenced in scan, all the values are located very closely thus tend to fit the coalesced memory access pattern.

GPU memory store (Gstore_fdw)



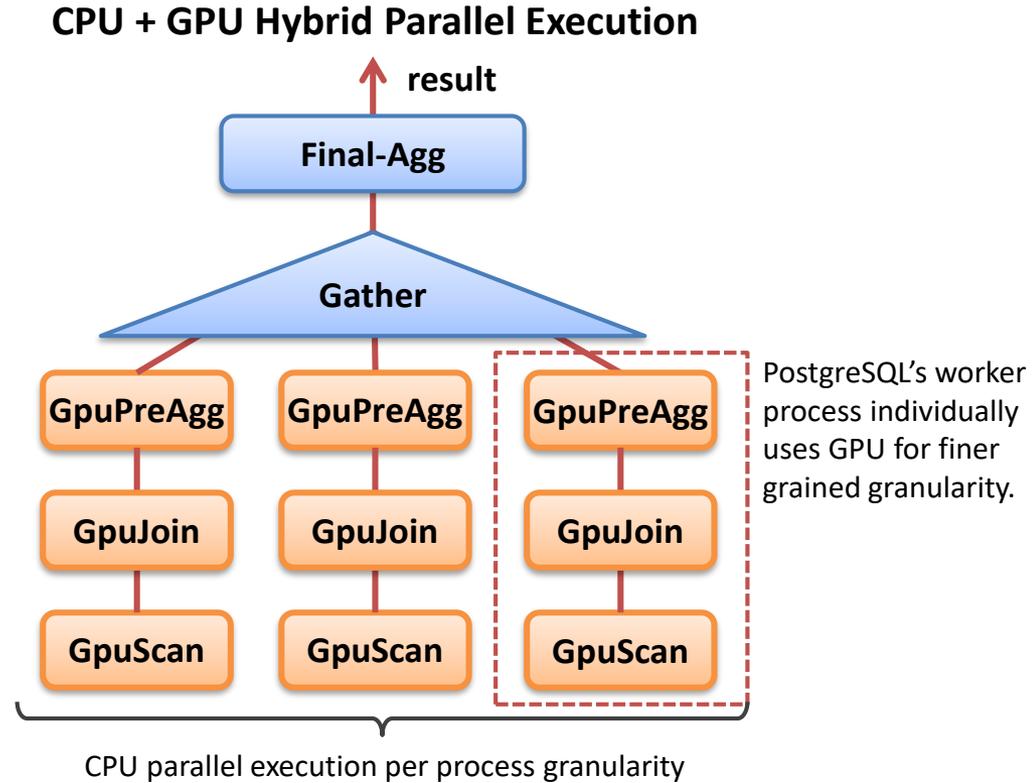
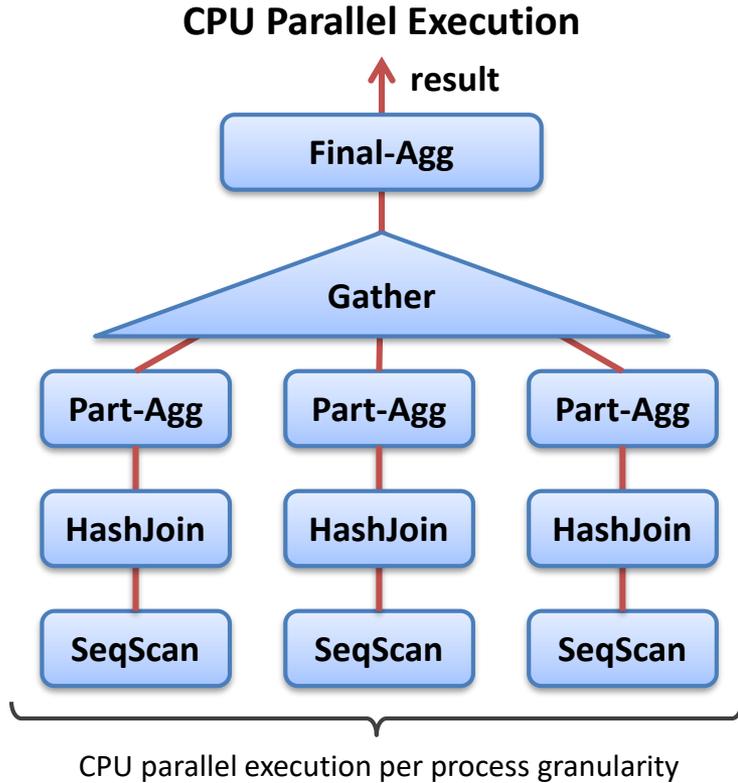
Gstore_fdw provides a set of interface to read/write GPU device memory using standard SQL. You can load bulk data into GPU's device memory using INSERT command, and so on. Because all the operations are handled inside PostgreSQL database system, data keep its binary form (so, no needs to dump as CSV file once and parse by Python script again). SQL is one of the most flexible tool for data management, so you can load arbitrary data-set from the master table, and apply pre-processing required by machine-learning algorithm on the fly.

One other significant feature is data-collaboration with external programs like Python scripts. GPU device memory can be shared with external program once identifier of the acquired memory region (just 64bytes token) is exported. It enables to use PostgreSQL as a powerful data management infrastructure for machine-learning usage. Even though Gstore_fdw now supports only 'pgstrom' internal format, we will support other internal format in the next or future version.



Advanced SQL Infrastructure

PostgreSQL v9.6/v10 support – CPU+GPU Hybrid Parallel

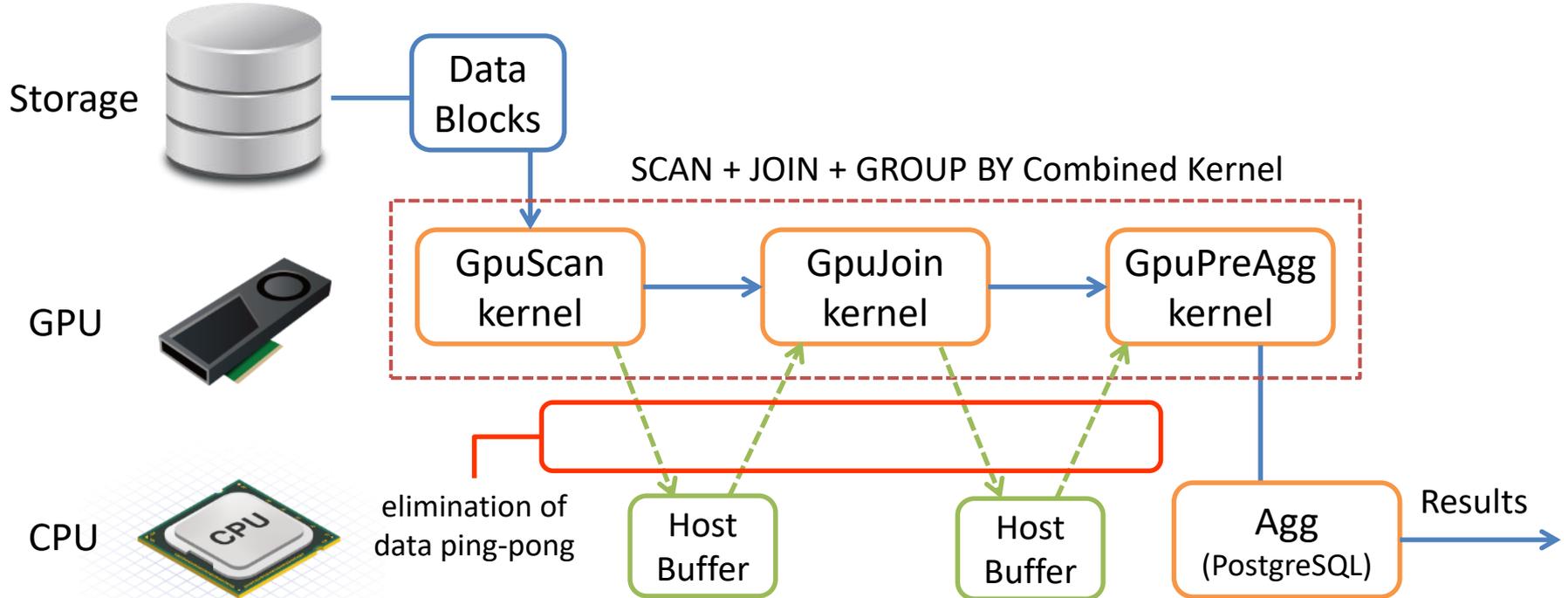


➔ **Multi-process pulls up GPU usage more efficiently**

One epoch-making feature at PostgreSQL v9.6 was parallel query execution based on concurrent multi-processes. It also extended the custom-scan interface for extensions to support parallel-query. PG-Strom v2.0 was re-designed according to the new interface set, then it enables GPU aware custom-plan to run on the background worker process.

Heuristically, capacity of single CPU thread is not sufficient to supply enough amount of data stream to GPU. It is usually much narrower than GPU's computing capability. So, we can expect CPU parallel execution assists to pull up GPU usage with much higher data supply ratio.

SCAN + JOIN + GROUP BY combined GPU kernel

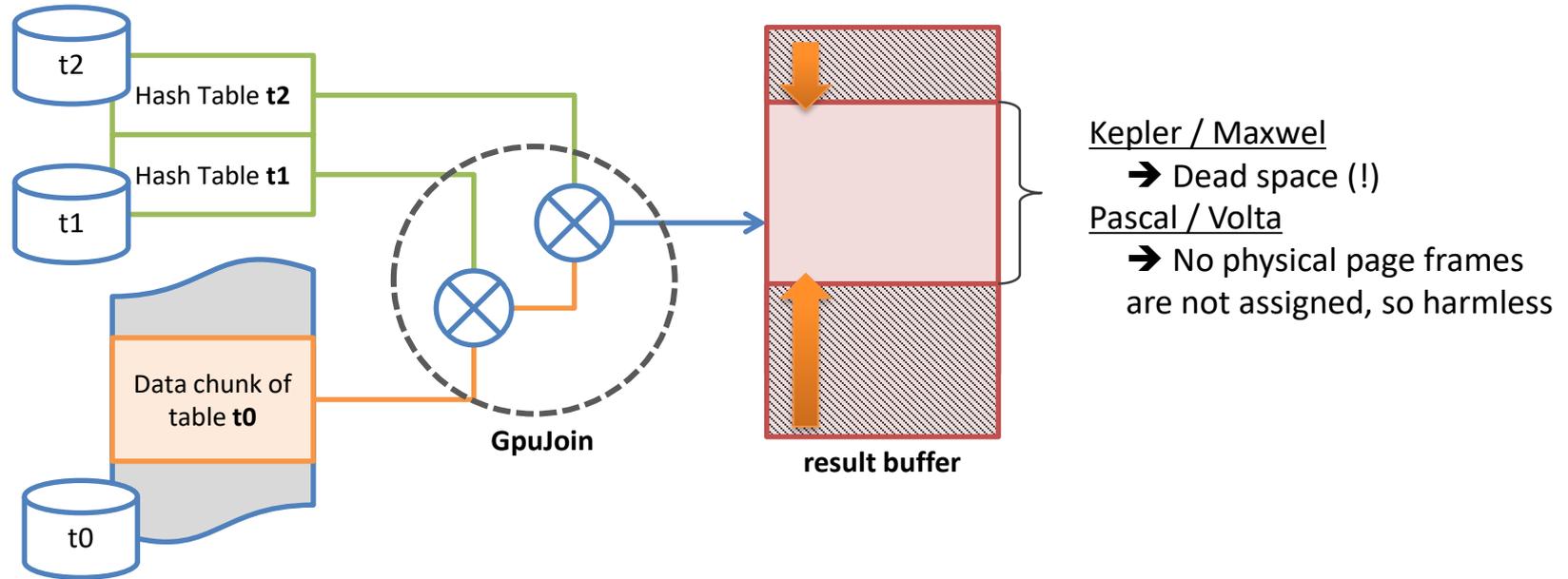


CustomScan interface allows extension to inject alternative implementation of query execution plan. If estimated cost is more reasonable than built-in implementation, query optimizer of PostgreSQL chooses the alternative plan. PG-Strom provides its custom logic for SCAN, JOIN and GROUP BY workloads, with GPU acceleration.

Usually, JOIN consumes the result of SCAN then generates records as its output. The output of JOIN often performs as input of GROUP BY, then it generates aggregation.

In case when GpuPreAgg, GpuJoin and GpuScan are continuously executed, we have an opportunity of further optimization by reduction of the data transfer over PCIe bus. The result of GpuScan can perform as GpuJoin's input, and the result of GpuJoin can also perform as GpuPreAgg's input, if all of them shall be continuously executed. PG-Strom tries to re-use the result buffer of the previous step as input buffer of the next step. It allows to eliminate the data ping-pong over the PCIe-bus. Once SCAN + JOIN + GROUP BY combined GPU kernel gets ready, it can run with the most efficient GPU kernel because it does not need data exchange over the query execution plan.

Utilization of demand paging of GPU device memory



Buffer size estimation is not an easy job for SQL workloads. In general, we cannot know exact number of result rows unless query is not actually executed, even though table statistics informs us rough estimation through the query planning.

GPU kernel needs result buffer for each operation, and it has to be acquired prior to its execution. It has been a problematic trade-off because a tight configuration with small margin often leads lack of the result buffer, on the other hands, buffer allocation with large margin makes unignorable dead space on the GPU device memory.

The recent GPU (Pascal / Volta) supports demand paging of GPU device memory. It assigns physical page frame on demand, thus, unused region consumes no physical device memory. It means that large margin configuration consumes no dead physical device memory.

It also allows to simplify the code to estimate the size of result buffer and to retry GPU kernel invocation with larger result buffer. These logics are very complicated and had many potential bugs around the error pass.

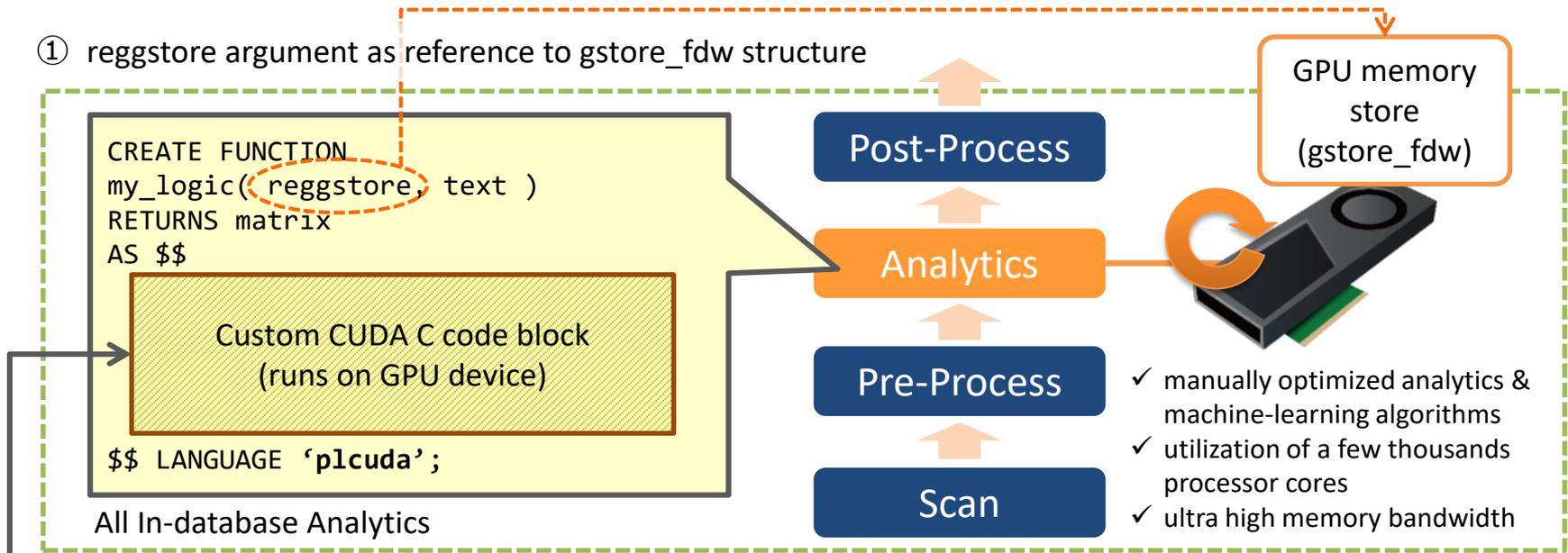
PG-Strom v2.0 fully utilized the demand paging of GPU device, thus we could eliminate the problematic code. It also contributed the stability of the software.



Miscellaneous Improvement

PL/CUDA related enhancement

① reggstore argument as reference to gstore_fdw structure



`#plcuda_include my_distance`

```
CREATE FUNCTION my_distance(reggstore, text)
RETURNS text
AS $$
  if ($2 = "manhattan")
    return "#define dist(X,Y)  abs(X-y)";
  if ($2 = "euclid")
    return "#define dist(X,Y)  ((X-y)^2)"
  $$ ...
```

② Inclusion of other function's result as a part of CUDA C code.

If "reggstore" type is supplied as argument of PL/CUDA function, user defined part of this PL/CUDA function receives this argument as pointer to the preserved GPU device memory for gstore_fdw. It allows to reference the data preliminary loaded onto the Gstore_fdw. The `#plcuda_include` directive is newly supported to include the code which is returned from the specifies function. You can switch the code to be compiled according to the arguments, not to create multiple but similar variations.

New data type support

Numeric

- ▣ float2

Network address

- ▣ macaddr, inet, cidr

Range data types

- ▣ int4range, int8range, tsrange, tstzrange, daterange

Miscellaneous

- ▣ uuid, reggstore

The “float2” data type is implemented by PG-Strom, not a standard built-in data type. It represents half-precision floating point values. People in machine-learning area often use FP16 for more short representation of matrix; less device memory consumption and higher computing throughput.

Documentation and Packaging

PG-Strom official documentation

The screenshot shows the PG-Strom Manual website. The left sidebar contains navigation links: Home, Install, Tutorial, Advanced Features, PL/CUDA, References, and Release Note. The main content area is titled 'Docs > Install' and includes a search bar, a language selector (Japanese | English), and a 'Checklist' section. The checklist lists requirements for Server Hardware, GPU Device, Operating System, PostgreSQL, and CUDA Toolkit. Below the checklist is the 'OS Installation' section, which provides instructions for choosing a Linux distribution and installing the system. At the bottom, there are links for 'Previous' and 'Next'.

HeteroDB Software Distribution Center

The screenshot shows the HeteroDB Software Distribution Center website. The page title is 'HeteroDB Software Distribution Center'. Below the title, there are sections for 'yum repository configurations', 'PG-Strom Source Tarball', and 'All RPM Files'. The 'yum repository configurations' section lists a single repository: [heterodb-swdc-1.0-1.el7.noarch.rpm](https://heterodb.github.io/swdc/). The 'PG-Strom Source Tarball' section lists a single tarball: [pg_strom-1.9-180412.tar.gz](https://heterodb.github.io/swdc/pg_strom-1.9-180412.tar.gz). The 'All RPM Files' section lists several RPM packages, including [heterodb-swdc-1.0-1.el7.noarch.rpm](https://heterodb.github.io/swdc/heterodb-swdc-1.0-1.el7.noarch.rpm), [nvme_strom-0.98-1.el7.x86_64.rpm](https://heterodb.github.io/swdc/nvme_strom-0.98-1.el7.x86_64.rpm), [pg_strom-PG10-1.9-180412.el7.x86_64.rpm](https://heterodb.github.io/swdc/pg_strom-PG10-1.9-180412.el7.x86_64.rpm), [pg_strom-PG96-1.9-180412.el7.x86_64.rpm](https://heterodb.github.io/swdc/pg_strom-PG96-1.9-180412.el7.x86_64.rpm), [postgresql-alternatives-1.0-1.el7.noarch.rpm](https://heterodb.github.io/swdc/postgresql-alternatives-1.0-1.el7.noarch.rpm), [heterodb-swdc-1.0-1.el7.src.rpm](https://heterodb.github.io/swdc/heterodb-swdc-1.0-1.el7.src.rpm), [pg_strom-PG10-1.9-180412.el7.src.rpm](https://heterodb.github.io/swdc/pg_strom-PG10-1.9-180412.el7.src.rpm), [pg_strom-PG96-1.9-180412.el7.src.rpm](https://heterodb.github.io/swdc/pg_strom-PG96-1.9-180412.el7.src.rpm), [nvme_strom-debuginfo-0.98-1.el7.x86_64.rpm](https://heterodb.github.io/swdc/nvme_strom-debuginfo-0.98-1.el7.x86_64.rpm), [pg_strom-PG10-debuginfo-1.9-180412.el7.x86_64.rpm](https://heterodb.github.io/swdc/pg_strom-PG10-debuginfo-1.9-180412.el7.x86_64.rpm), and [pg_strom-PG96-debuginfo-1.9-180412.el7.x86_64.rpm](https://heterodb.github.io/swdc/pg_strom-PG96-debuginfo-1.9-180412.el7.x86_64.rpm).

Documentation was totally rewritten with markdown that is much easier for timely update than raw HTML based one.

It is now published at <http://heterodb.github.io/pg-strom/>

RPM packages are also available for RHEL7.x / CentOS 7.x. PG-Strom and related software are available on the HeteroDB Software Distribution Center (SWDC) at <https://heterodb.github.io/swdc/>

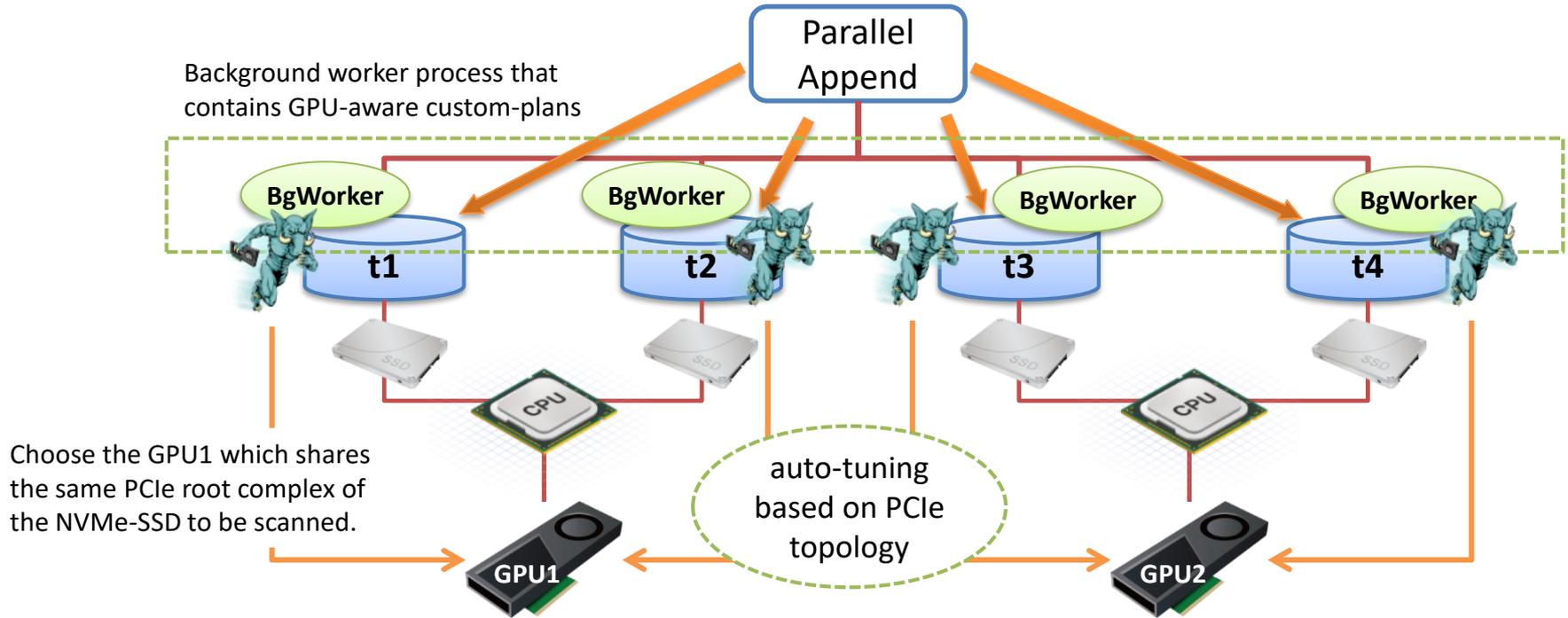
You can use the SWDC as yum repository source.



Post-v2.0 Development Roadmap

PostgreSQL v11 support (1/2) – Parallel Append & Multi-GPUs

- PG10 Restriction: GPUs not close to SSD must be idle during the scan across partition table.
- PG11 allows to scan partitioned children in parallel. It also makes multiple-GPUs active simultaneously.



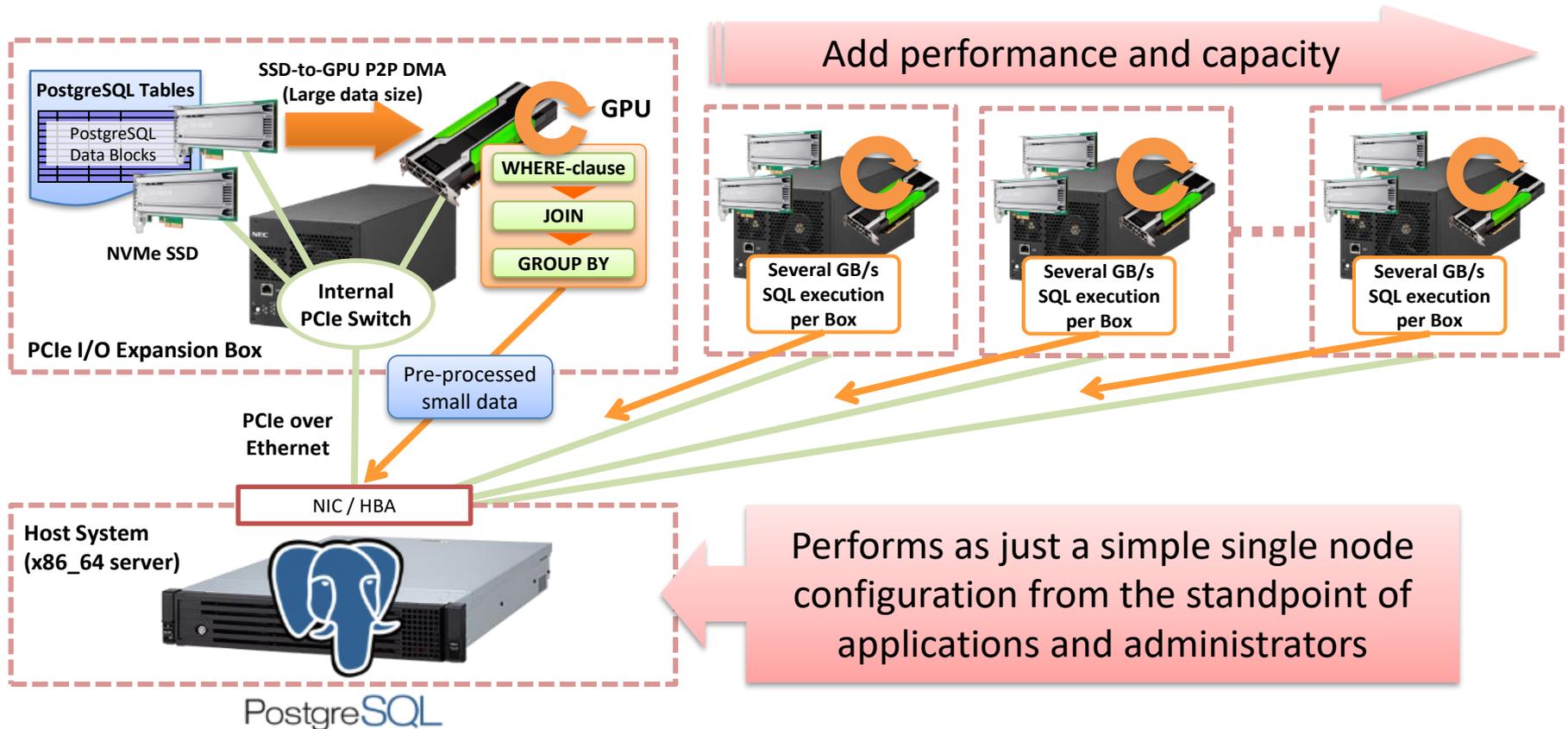
NVIDIA GPUDirect RDMA, is a basis technology of SSD-to-GPU Direct SQL Execution, requires GPU and SSD should share the PCIe root complex, thus P2P DMA route should not traverse QPI link. It leads a restriction on multi-GPUs configuration with partitioned table.

When background worker scans partitioned child tables across multiple SSDs, GPU-aware custom plan needs to choose its neighbor GPU to avoid QPI traversal. In other words, the target table for scan determines the GPU to be attached on the background workers.

In PG10, partitioned child tables are sequentially picked up to scan, so we cannot activate more than one GPUs simultaneously because the secondary GPU will cause QPI traverse on usual hardware configuration (1:CPU – 1:GPU + n:SSDs).

PG11 supports parallel scan across partitioned child tables. It allows individual background worker activate its neighbor GPU for the tables they are scanning. It enables to utilize multiple GPUs under SSD-to-GPU Direct SQL Execution for larger data processing.

PostgreSQL v11 support (2/2) – In-box distributed query execution



Multi-GPUs capability will expand the opportunity of big data processing for more bigger data set, probably, close to 100TB.

Multiple vendors provide PCIe I/O expansion box solution that allows to install PCIe devices on the physically separated box which is connected to the host system using fast network. In addition, some of the solution have internal PCIe switch that can route P2P DMA packet inside of the I/O box. It means PG-Strom handles SSD-to-GPU Direct SQL Execution on the I/O box with little interaction to the host system, and runs the partial workload on the partitioned table per box in parallel once multi-GPUs capability get supported at PG11.

From the standpoint of applications and administrators, it is just a simple single node configuration even though many GPUs and SSDs are installed, thus, no need to pay attention for distributed transaction. It makes application design and daily maintenance so simplified.

Other significant features

- cuPy data format support of Gstore_fdw
- BRIN index support
- Basic PostGIS support
- NVMe over Fabric support
- GPU device function that can return varlena datum
- Semi- / Anti- Join support
- MVCC visibility checks on the device side
- Compression support of in-memory columnar cache

See [003: Development Roadmap](#) for more details.



**Run! Beyond
the Limitations**