PG-Strom
GPU Accelerated Asynchronous Query Execution Module

NEC Europe, Ltd
SAP Global Competence Center
KaiGai Kohei <kohei.kaigai@emea.nec.com>
Homogeneous vs Heterogeneous Computing

KPIs
- Computing Performance
- Power Consumption
- System Cost
- Variety of Applications
- Vendor Support
- Software Development

Homogeneous Scale-Up

Heterogeneous Scale-Up

Scale-out
(not a topic of today’s talk)
## Characteristics of GPU (1/2)

<table>
<thead>
<tr>
<th></th>
<th>Nvidia Kepler</th>
<th>AMD GCN</th>
<th>Intel SandyBridge</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model</strong></td>
<td>GTX 680 (*) (Q1/2012)</td>
<td>FirePro S9000 (Q3/2012)</td>
<td>Xeon E5-2690 (Q1/2012)</td>
</tr>
<tr>
<td><strong>Number of Transistors</strong></td>
<td>3.54 billion</td>
<td>4.3 billion</td>
<td>2.26 billion</td>
</tr>
<tr>
<td><strong>Number of Cores</strong></td>
<td><strong>1536</strong> Simple</td>
<td><strong>1792</strong> Simple</td>
<td><strong>16</strong> Functional</td>
</tr>
<tr>
<td><strong>Core clock</strong></td>
<td>1006MHz</td>
<td>925MHz</td>
<td>2.9GHz</td>
</tr>
<tr>
<td><strong>Peak FLOPS</strong></td>
<td>3.01Tflops</td>
<td>3.23TFlops</td>
<td>185.6GFlops</td>
</tr>
<tr>
<td><strong>Memory Size / TYPE</strong></td>
<td>2GB, GDDR5</td>
<td>6GB, GDDR5</td>
<td>up to 768GB, DDR3</td>
</tr>
<tr>
<td><strong>Memory Bandwidth</strong></td>
<td>~192GB/s</td>
<td>~264GB/s</td>
<td>~51.2GB/s</td>
</tr>
<tr>
<td><strong>Power Consumption</strong></td>
<td>~195W</td>
<td>~225W</td>
<td>~135W</td>
</tr>
</tbody>
</table>

(*) Nvidia shall release high-end model (Kepler K20) at Q4/2012
Characteristics of GPU (2/2)

Example)

\[ Z_i = X_i + Y_i \quad (0 \leq i \leq n) \]

\[
\begin{array}{cccc}
X_0 & X_1 & X_2 & \cdots & X_n \\
+ & + & + & & + \\
Y_0 & Y_1 & Y_2 & \cdots & Y_n \\
\downarrow & \downarrow & \downarrow & \cdots & \downarrow \\
Z_0 & Z_1 & Z_2 & \cdots & Z_n \\
\end{array}
\]

Assign a particular “core”

Nvidia’s GeForce GTX 680 Block Diagram (1536 Cuda cores)
Example) Parallel Execution of “sprt\( (X_i^2 + Y_i^2) < Z_i \)"

### GPU Code

```c
__kernel void
sample_func(bool result[], float x[], float y[], float z[]) {
    int i = get_global_id(0);

    result[i] = (bool)(sqrt(x[i]^2 + y[i]^2) < z[i]);
}
```

### Host Code

```c
#define N   (1<<20)
size_t g_itemsz = N / 1024;
size_t l_itemsz = 1024;

/* Acquire device memory and data transfer (host -> device) */
X = clCreateBuffer(cxt, CL_MEM_READ_WRITE, sizeof(float)*N, NULL, &r);
clEnqueueWriteBuffer(cmdq, X, CL_TRUE, sizeof(float)*N, ...);

/* Set argument of the kernel code */
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&X);

/* Invoke device kernel */
clEnqueueNDRangeKernel(cmdq, kernel, 1, &g_itemsz, &l_itemsz, ...);
```
1. Build & Load GPU Kernel

- Source Code
- OpenCL Compiler
- L2 Cache
- Host Memory
  - result buffer
  - X, Y, Z buffer
- Command Queue
- Device DRAM
  - GPU Kernel
Programming with GPU (2/2)

1. Build & Load GPU Kernel
2. Allocate Device Memory
Programming with GPU (2/2)

1. Build & Load GPU Kernel
2. Allocate Device Memory
3. Enqueue DMA Transfer (host $\rightarrow$ device)
1. Build & Load GPU Kernel
2. Allocate Device Memory
3. Enqueue DMA Transfer (host ➔ device)
4. Setup Kernel Arguments
1. Build & Load GPU Kernel
2. Allocate Device Memory
3. Enqueue DMA Transfer (host $\rightarrow$ device)
4. Setup Kernel Arguments
5. Enqueue Execution of GPU Kernel
Programming with GPU (2/2)

1. Build & Load GPU Kernel
2. Allocate Device Memory
3. Enqueue DMA Transfer (host ➔ device)
4. Setup Kernel Arguments
5. Enqueue Execution of GPU Kernel
6. Enqueue DMA Transfer (device ➔ host)
Programming with GPU (2/2)

1. Build & Load GPU Kernel
2. Allocate Device Memory
3. Enqueue DMA Transfer (host $\rightarrow$ device)
4. Setup Kernel Arguments
5. Enqueue Execution of GPU Kernel
6. Enqueue DMA Transfer (device $\rightarrow$ host)
7. Synchronize the command queue
   - DMA Transfer (host $\rightarrow$ device)
Programming with GPU (2/2)

1. Build & Load GPU Kernel
2. Allocate Device Memory
3. Enqueue DMA Transfer (host \(\rightarrow\) device)
4. Setup Kernel Arguments
5. Enqueue Execution of GPU Kernel
6. Enqueue DMA Transfer (device \(\rightarrow\) host)
7. Synchronize the command queue
   - DMA Transfer (host \(\rightarrow\) device)
   - Execution of GPU Kernel
Programming with GPU (2/2)

1. Build & Load GPU Kernel
2. Allocate Device Memory
3. Enqueue DMA Transfer (host $\rightarrow$ device)
4. Setup Kernel Arguments
5. Enqueue Execution of GPU Kernel
6. Enqueue DMA Transfer (device $\rightarrow$ host)
7. Synchronize the command queue
   - DMA Transfer (host $\rightarrow$ device)
   - Execution of GPU Kernel
   - DMA Transfer (device $\rightarrow$ host)
8. Release Device Memory
Programming with GPU (2/2)

1. Build & Load GPU Kernel
2. Allocate Device Memory
3. Enqueue DMA Transfer (host ➔ device)
4. Setup Kernel Arguments
5. Enqueue Execution of GPU Kernel
6. Enqueue DMA Transfer (device ➔ host)
7. Synchronize the command queue
   - DMA Transfer (host ➔ device)
   - Execution of GPU Kernel
   - DMA Transfer (device ➔ host)
8. Release Device Memory
Basic idea to utilize GPU

- Simultaneous (Asynchronous) execution of CPU and GPU
- Minimization of data transfer between host and device

![Diagram showing CPU, GPU, memory, and data transfer connections.]

- **Memory**: DDR3-1600 (51.2GB/s)
- **CPU**: Xeon processor
- **IO Hub**: DMA Transfer
- **HBA**: SAS 2.0 (600MB/s)
- **PCI-E 3.0 x16**: (32.0GB/s)
- **GPGPU (non-integrated)**: DDR5 192.2GB/s
- **Device DRAM**: on-device buffer

**Super Parallel Execution**
Back to the PostgreSQL world

Don’t I forget I’m talking at PGconf.EU 2012?
Re-definition of SQL/MED

- SQL/MED (Management of External Data)
  - External data source performing as if regular tables
  - Not only “management”, but external computing resources also

![Diagram of SQL/MED](image)
Introduction of PG-Strom

PG-Strom is ...

- A FDW extension of PostgreSQL, released under the GPL v3.  
  [https://github.com/kaigai/pg_strom](https://github.com/kaigai/pg_strom)
- Not a stable module, please *don’t* use in production system yet.
- Designed to utilize GPU devices for CPU off-load according to their characteristics.

Key features of PG-Strom

- Just-in-time pseudo code generation for GPU execution
- Column-oriented internal data structure
- Asynchronous query execution
  - Reduction of response-time dramatically!
Asynchronous Execution using CPU/GPU (1/2)

CPU characteristics
- Complex Instruction, less parallelism
- Expensive & much power consumption per core
- I/O capability

GPU characteristics
- Simple Instruction, much parallelism
- Cheap & less power consumption per core
- Device memory access only (except for integrated GPU)

“Best Mix” strategy of PG-Stom
- CPU focus on I/O and control stuff.
- GPU focus on calculation stuff.
Asynchronous Execution using CPU/GPU (2/2)

vanilla PostgreSQL

PostgreSQL with PG-Strom

- Iteration of scan tuples and evaluation of qualifiers
- Asynchronous memory transfer and execution
- Larger “chunk” to scan the database at once
- Earlier than “Only CPU” scan

Red: Scan tuples on shared-buffers
Green: Execution of the qualifiers
So what, How fast is it?

CPU: Xeon E5-2670 (2.60GHz), GPU: NVidia GeForce GT640, RAM: 384GB

Both of regular `rtbl` and PG-Strom `ftbl` contain 20milion rows with same value
Architecture of PG-Strom

World of CPU

- Regular Tables
- Shadow Tables

Shared buffer

Chunk

- Data
- Pseudo codes

World of GPU

- GPU Device Memory
- Super Parallel Execution
- GPU Kernel Function

Async DMA Transfer

Preload

Result

- SeqScan, etc...
- ForeignScan

Query Executor

PG-Strom

Event Monitor

PG-Strom GPU Control Server

An extra daemon

Works according to given pseudo code

Postmaster

PostgreSQL Backend

Downloaded from https://www.datacamp.com/community/tutorials/
SELECT * FROM ftbl WHERE 
  c like '%xyz%' AND sqrt((x-256)^2+(y-100)^2) < 10;

contains unsupported operators / functions

Translation to pseudo code:

xreg10 = $(ftbl.x)
xreg12 = 256.000000::double
xreg8 = (xreg10 - xreg12)
xreg10 = 2.000000::double
xreg6 = pow(xreg8, xreg10)
xreg12 = $(ftbl.y)
xreg14 = 128.000000::double
Regularly, we should avoid branch operations on GPU code

```
__global__
void kernel_qual(const int commands[],...)
{
    const int *cmd = commands;
    while (*cmd != GPUCMD_TERMINAL_COMMAND)
    {
        switch (*cmd)
        {
            case GPUCMD_CONREF_INT4:
                regs[*(cmd+1)] = *(cmd + 2);
                cmd += 3;
                break;
            case GPUCMD_VARREF_INT4:
                VARREF_TEMPLATE(cmd, uint);
                break;
            case GPUCMD_OPER_INT4_PL:
                OPER_ADD_TEMPLATE(cmd, int);
                break;
        }
    }
}
```

result = 0;
if (condition)
{
    result = a + b;
}
else
{
    result = a - b;
}
return 2 * result;
Regularly, we should avoid branch operations on GPU code

```c
__global__
void kernel_qual(const int commands[], ...)
{
    const int *cmd = commands;
    while (*cmd != GPUCMD_TERMINAL_COMMAND)
    {
        switch (*cmd)
        {
            case GPUCMD_CONREF_INT4:
                regs[*((cmd + 1))] = *(cmd + 2);
                cmd += 3;
                break;
            case GPUCMD_VARREF_INT4:
                VARREF_TEMPLATE(cmd, uint);
                break;
            case GPUCMD_OPER_INT4_PL:
                OPER_ADD_TEMPLATE(cmd, int);
                break;
        }
    }
    return 2 * result;
}
```

result = 0;
if (condition)
{
    result = a + b;
}
else
{
    result = a - b;
}
return 2 * result;
Pseudo code generation (2/2)

Regularly, we should avoid branch operations on GPU code

```c
__global__
void kernel_qual(const int commands[],...)
{
    const int *cmd = commands;
    while (*cmd != GPUCMD_TERMINAL_COMMAND)
    {
        switch (*cmd)
        {
            case GPUCMD_CONREF_INT4:
                regs[*(cmd+1)] = *(cmd + 2);
                cmd += 3;
                break;
            case GPUCMD_VARREF_INT4:
                VARREF_TEMPLATE(cmd, uint);
                break;
            case GPUCMD_OPER_INT4_PL:
                OPER_ADD_TEMPLATE(cmd, int);
                break;
        }
    }
}

result = 0;
if (condition)
{
    result = a + b;
}
else
{
    result = a - b;
}
return 2 * result;
```

Regularly, we should avoid branch operations on GPU code.
Regularly, we should avoid branch operations on GPU code

```c
__global__
void kernel_qual(const int commands[],...)
{
    const int *cmd = commands;
    while (*cmd != GPUCMD_TERMINAL_COMMAND)
    {
        switch (*cmd)
        {
            case GPUCMD_CONREF_INT4:
                regs[*((cmd+1)]) = *(cmd + 2);
                cmd += 3;
                break;
            case GPUCMD_VARREF_INT4:
                VARREF_TEMPLATE(cmd, uint);
                break;
            case GPUCMD_OPER_INT4_PL:
                OPER_ADD_TEMPLATE(cmd, int);
                break;
        }
    }
}
```

result = 0;
if (condition)
{
    result = a + b;
}
else
{
    result = a - b;
}
return 2 * result;
Regularly, we should avoid branch operations on GPU code

\[
\text{result} = 0;
\]

\[
\text{if (condition)}
\quad \{
\quad \text{result} = \text{a} + \text{b};
\quad \}
\]

\[
\text{else}
\quad \{
\quad \text{result} = \text{a} - \text{b};
\quad \}
\]

\[
\text{return 2 * result;}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]

\[
\text{\textbf{\_global\_}}
\]
OT: Why “pseudo”, not native code

Initial design at Jan-2012

SQL Query

Query Parser

PostgreSQL Core

Query Planner

PG-Strom module

Query Executor

PG-Strom Planner

Run-time GPU Code Generator

GPU Source

Pre-Compiled Binary Cache

GPU Source

GPU Binary

Compile Time

nvcc

PG-Strom module

PG-Strom Executor

Qualifier

Init

Load

GPU

Async Memcpy

Async Execute

GPU

Async Memcpy

Async Memcpy

Num of kernels to load

Compile Time

Columns used to Qualifiers

Columns used to Target-List

Regular Databases

pg_strom schema

Qualifiers

Scan

PostgreSQL Core

Num of kernels to load

PostgreSQL Core
Save the bandwidth of PCI-Express bus

E.g) SELECT name, tel, email, address FROM address_book
    WHERE sqrt((pos_x - 24.5)^2 + (pos_y - 52.3)^2) < 10;

→ No sense to fetch columns being not in use

- : Scan tuples on the shared-buffers
- : Execution of the qualifiers
- : Columns being not used the qualifiers

Reduction of data-size to be transferred via PCI-E
### Data density & Column-oriented structure (1/3)

#### FOREIGN TABLE ft

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>float</td>
<td>text</td>
<td>Z</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td></td>
</tr>
</tbody>
</table>

#### FOREIGN TABLE ft

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rowid</td>
<td>nitems</td>
<td>isnull</td>
<td>values</td>
</tr>
<tr>
<td>4000</td>
<td>2000</td>
<td>{0,0,0,1,0,0,...}</td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td>2000</td>
<td>{0,0,0,0,0,0,...}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

#### (shadow) TABLE "public.ft.rowid"

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rowid</td>
<td>nitems</td>
<td>isnull</td>
<td>values</td>
</tr>
<tr>
<td>4000</td>
<td>15</td>
<td>{0,0,...}</td>
<td></td>
</tr>
<tr>
<td>4015</td>
<td>20</td>
<td>{0,0,...}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

#### (shadow) TABLE "public.ft.z.cs"

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rowid</td>
<td>nitems</td>
<td>isnull</td>
<td>values</td>
</tr>
<tr>
<td>4000</td>
<td>250</td>
<td>{0,0,...}</td>
<td>{ 'hello', 'world', ... }</td>
</tr>
<tr>
<td>4015</td>
<td>250</td>
<td>{0,0,...}</td>
<td>{ 'aaa', 'bbb', 'ccc', ... }</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

#### (shadow) TABLE "public.ft.y.cs"

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rowid</td>
<td>nitems</td>
<td>isnull</td>
<td>values</td>
</tr>
<tr>
<td>4000</td>
<td>500</td>
<td>{0,0,...}</td>
<td>{ 1.38, 6.45, 2.15, ... }</td>
</tr>
<tr>
<td>4500</td>
<td>500</td>
<td>{0,1,...}</td>
<td>{ 4.32, 5.46, 3.14, ... }</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

#### (shadow) TABLE "public.ft.a.cs"

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rowid</td>
<td>nitems</td>
<td>isnull</td>
<td>values</td>
</tr>
<tr>
<td>4000</td>
<td>500</td>
<td>{0,0,...}</td>
<td>{10, 20, 30, 40, 50, ...}</td>
</tr>
<tr>
<td>4500</td>
<td>500</td>
<td>{0,1,...}</td>
<td>{11, 0, 31, 41, 51, ...}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>14200</td>
<td>200</td>
<td>{0,0,...}</td>
<td>{19, 29, 39, 49, 59, ...}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Data density & Column-oriented structure (2/3)

postgres=# CREATE FOREIGN TABLE example
   (a int, b text) SERVER pg_strom;
CREATE FOREIGN TABLE

postgres=# SELECT * FROM pgstrom_shadow_relations;
    oid   |     relname           | relkind | relsize
----------+----------------------+---------+----------
     16446 | public.example.rowid  | r       | 0
     16449 | public.example.idx    | i       | 8192
     16450 | public.example.a.cs   | r       | 0
     16453 | public.example.a.idx  | i       | 8192
     16454 | public.example.b.cs   | r       | 0
     16457 | public.example.b.idx  | i       | 8192
     16462 | public.example.seq    | S       | 8192
(9 rows)

postgres=# SELECT * FROM pg_strom."public.example.a.cs" ;
    rowid | nitems | isnull | values
----------+--------+--------+----------
(0 rows)
Data density & Column-oriented structure (2/3)

② Calculation
① Transfer
③ Write-Back

Table: my_schema.ft1.b.cs
- 10100: \{2.4, 5.6, 4.95, ... \}
- 10300: \{10.23, 7.54, 5.43, ... \}

Table: my_schema.ft1.c.cs
- 10100: \{'2010-10-21', ...\}
- 10200: \{'2011-01-23', ...\}
- 10300: \{'2011-08-17', ...\}

PgStromChunkBuffer
- opcode
- rowmap
- value a[]
- value b[]
- value c[]
- value d[]

Pseudo Code
<not used>
<not used>

Less bandwidth consumption
Also, suitable for data compression
Demonstration
Key features towards upcoming v9.3 (1/2)

Extra Daemon

- It enables extension to manage background worker processes.
- Pre-requisites to implement PG-Strom’s GPU control server
- Alvaro submitted this patch on CommitFest:Nov.
Writable Foreign Table

- It enables to use usual INSERT, UPDATE or DELETE to modify foreign tables managed by PG-Strom.
- KaiGai submitted a proof-of-concept patch to CommitFest:Sep.
- In-core postgresql_fdw is needed for working example.
More Rapidness (1/2) – Parallel Data Load

- Regular Tables
- Shadow Tables

World of CPU

- SeqScan etc...
- ForeignScan
- Query Executor

PostgreSQL Backend

- PG-Strom Data Loader

Postmaster

World of GPU

- GPU Device Memory
- Super Parallel Execution
- GPU Kernel Function

Works according to given pseudo code

Async DMA Transfer

Preload

Chunk to be loaded

chunk

shared buffer

PG-Strom

GPU Server Control Function
More Rapidness (2/2) – TargetList Push-down

### SELECT

```
SELECT ((a + b) * (c - d))^2 FROM ftbl;
```

### SELECT

```
SELECT pseudo_col FROM ftbl;
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>pseudo_col</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>144</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>144</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

- Pseudo column hold “computed” result, to be just referenced
- Performs as if extra columns exist in addition to table definition

Computed during ForeignScan
We need you getting involved

Project was launched from my personal curiousness,
So, it is uncertain how does PG-Strom fit “real-life” workload.
We definitely have to find out attractive usage of PG-Strom

Which region?
Which problem?
How to solve?

More feedback makes more improvement!
Summary

Characteristics of GPU device
- Inflexible instructions, but much higher parallelism
- Cheap & small power consumption per computing capability

PG-Strom
- Utilization of GPU device for CPU off-load and rapid response
- Just-in-time pseudo code generation according to the given query
- Column-oriented data structure for data density on PCI-Express bus
  ➔ In the result, dramatic shorter response time

Upcoming development
- Upstream
  - Extra daemons, Writable Foreign Tables
- Extension
  - Move to OpenCL rather than CUDA

Your involvement can lead future evolution of PG-Strom
Any Questions?
Thank you

ありがとうございました
THANK YOU
DĚKUJEME
DANKE
MERCI
GRAZIE
GRACIAS
Empowered by Innovation

NEC