A Virtual Machine Model for Accelerating Relational Database Joins using a General Purpose GPU

Kevin Angstadt
University of Virginia
Charlottesville, Virginia
kaa2nx@virginia.edu

Ed Harcourt
St. Lawrence University
Canton, New York
edharcourt@stlawu.edu
Databases Are Used Everywhere
SQLite: Most Deployed DB

Virtual Machine-based Database Engine

• 300 million copies of Firefox
• 20 million Apple computers
• 500 million iPhones*
• 1 billion Android Devices**
• 450 million registered Skype users
• 10 million Solaris 10 installations

Adapted from: https://www.sqlite.org/mostdeployed.html
*http://onforb.es/1gpk4Fs
**http://www.androidcentral.com/android-passes-1-billion-activations
GPUs as General Purpose Processors
GPUs as General Purpose Processors

G2 instance: 65¢/hr
GPUs as General Purpose Processors

G2 instance: 65¢/hr
GPUs as General Purpose Processors

G2 instance: 65¢/hr
Why don’t we combine SQLite and GPUs to improve performance?

Focus on JOINs
Overview

• Review of Database JOINs
• Why GPUs a good fit?
• Virtual Machine Implementation
  – Query Workflow
  – SQL Queries as Programs
  – Memory Concerns
  – VM Design
• Experimental Results
Database JOIN

Predefined, Static Relationship

JOIN
Cross-Join Example

• $T_1$ and $T_2$ share attribute $c_2$

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th></th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>$c_2$</td>
<td>$c_2$</td>
<td>$c_3$</td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td>x</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>z</td>
<td>z</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>w</td>
<td>7</td>
</tr>
</tbody>
</table>
Cross-Join Example

<table>
<thead>
<tr>
<th>$\mathbf{T}_1$</th>
<th>$\mathbf{T}_2$</th>
<th>$\mathbf{T}_1 \times \mathbf{T}_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>$c_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>3</td>
<td>w</td>
<td>w</td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td>z</td>
</tr>
<tr>
<td>1</td>
<td>w</td>
<td>w</td>
</tr>
<tr>
<td>2</td>
<td>z</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>2</td>
<td>z</td>
<td>w</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td>w</td>
</tr>
</tbody>
</table>
Cross-Join Example

\[
\begin{array}{c|c|c}
  \text{c}_1 & \text{c}_2 & \text{c}_3 \\
--- & --- & --- \\
  1 & w & x & 5 \\
  2 & z & x & 5 \\
  3 & z & x & 5 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
  \text{c}_1 & \text{c}_2 & \text{c}_2 & \text{c}_3 \\
--- & --- & --- & --- \\
  1 & w & z & 6 \\
  2 & z & z & 6 \\
  3 & z & w & 7 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
  \text{c}_1 & \text{c}_2 & \text{c}_3 \\
--- & --- & --- \\
  1 & w & w & 7 \\
  2 & z & z & 6 \\
  3 & z & w & 7 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
  \text{c}_1 & \text{c}_2 & \text{c}_2 & \text{c}_3 \\
--- & --- & --- & --- \\
  1 & w & x & 5 \\
  1 & w & z & 6 \\
  1 & w & w & 7 \\
  2 & z & x & 5 \\
  2 & z & z & 6 \\
  2 & z & w & 7 \\
  3 & z & x & 5 \\
  3 & z & z & 6 \\
  3 & z & w & 7 \\
\end{array}
\]
Restrict Result with Predicate

• SELECT * FROM T₁,T₂ WHERE T₁.c₂ = T₂.c₂

\[
\begin{array}{|c|c|}
\hline
T₁ & T₂ \\
\hline
\begin{array}{|c|c|}
\hline c₁ & c₂ \\
\hline 1 & w \\
2 & z \\
3 & z \\
\hline
\end{array} & \begin{array}{|c|c|c|}
\hline c₂ & c₃ \\
\hline x & 5 \\
z & 6 \\
w & 7 \\
\hline
\end{array}
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline c₁ & c₂ & c₂ & c₃ \\
\hline 1 & w & x & 5 \\
1 & w & z & 6 \\
1 & w & w & 7 \\
2 & z & x & 5 \\
2 & z & z & 6 \\
2 & z & w & 7 \\
3 & z & x & 5 \\
3 & z & z & 6 \\
3 & z & w & 7 \\
\hline
\end{array}
\]

T₁ ⊙ T₂
Restrict Result with Predicate

- **SELECT * FROM T₁, T₂ WHERE T₁.c₂ = T₂.c₂**

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c₁)</td>
<td>(c₂)</td>
</tr>
<tr>
<td>1</td>
<td>w</td>
</tr>
<tr>
<td>2</td>
<td>z</td>
</tr>
<tr>
<td>3</td>
<td>z</td>
</tr>
</tbody>
</table>

\(T₁ \times T₂\)
Restrict Result with Predicate

- SELECT * FROM $T_1, T_2$ WHERE $T_1.c_2 = T_2.c_2$

SIMD Execution
IMPLEMENTATION
Virginian Database

• Written by Peter Bakkum, NEC Labs, New Jersey
• Virtual Machine-based Implementation
  – Similar to SQLite
• Demonstrated speedup for single-table queries
• Presented “Tablet” data structure for efficient processing on CPU/GPU
# Table 1

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tablet 1</td>
<td>Tablet 2</td>
<td>Tablet 3</td>
<td>Tablet 4</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Tablet 1

Table 2
Query Workflow

Input Query
Query Workflow

Input Query

Virtual Machine Program
Query Workflow

Input Query

Virtual Machine Program

Set Up Result Table
Query Workflow

Input Query

Virtual Machine Program

Execute

Set Up Result Table
SELECT test.id, test1.uniformi, test.normali5 FROM test,test1
WHERE test1.uniformi > 60 AND test.normali5 < 0;
SELECT test.id, test1.uniformi, test.normali5 FROM test,test1
WHERE test1.uniformi > 60 AND test.normali5 < 0;

0: Table 0 0 0 0
1: Table 1 0 1 0
2: ResultColumn 0 0 0 id
3: ResultColumn 0 0 0 uniformi
4: ResultColumn 0 0 0 normali5
5: Parallel 0 0 16 0
6: Column 3 0 1 0
7: Integer 0 60 0 0
8: Le 3 0 14 0
9: Column 4 1 0 0
10: Integer 1 0 0 0
11: Lt 4 1 13 1
12: Invalid 0 0 0 0
13: Rowid 2 0 0 0
14: Result 2 3 0 0
15: Converge 0 0 0 0
16: Finish 0 0 0 0
### SQL → OPCODE Program

```sql
SELECT test.id, test1.uniformi, test.normali5 FROM test,test1
WHERE test1.uniformi > 60 AND test.normali5 < 0;
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Table</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Table</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ResultColumn</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>ResultColumn</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>ResultColumn</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Parallel</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>Column</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Integer</td>
<td>0</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Le</td>
<td>3</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>Column</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>Integer</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>Lt</td>
<td>4</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>Invalid</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>Rowid</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>Result</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>Converge</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>Finish</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
SQL → OPCODE Program

SELECT test.id, test1.uniformi, test.normali5 FROM test,test1
WHERE test1.uniformi > 60 AND test.normali5 < 0;
Allocating Result Table

• How much memory must be allocated?

\[ ||TableA|| \cdot ||TableB|| \]

• If each table has 3500 rows, then we have...

  12 250 000 rows

• Memory limits of GPU: use mapped memory!
## Virtual Machine

<table>
<thead>
<tr>
<th>0: Table</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Table</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2: ResultColumn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>3: ResultColumn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>uniformi</td>
</tr>
<tr>
<td>4: ResultColumn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>normali5</td>
</tr>
<tr>
<td>5: Parallel</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>6: Column</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7: Integer</td>
<td>0</td>
<td>60</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8: Le</td>
<td>3</td>
<td>0</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>9: Column</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10: Integer</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11: Lt</td>
<td>4</td>
<td>1</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>12: Invalid</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13: Invalid</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14: Result</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15: Converge</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16: Finish</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Virtual Machine

![Intel Core i7 and Nvidia GeForce GTX 780 Graphics Card]

<table>
<thead>
<tr>
<th>Table 0:</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1:</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ResultColumns 2:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>id</td>
</tr>
<tr>
<td>ResultColumns 3:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>uniformi</td>
</tr>
<tr>
<td>ResultColumns 4:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>normali</td>
</tr>
<tr>
<td>Parallel 5:</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Column 6:</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Column 7:</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Column 8:</td>
<td>3</td>
<td>0</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>Column 9:</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Integer 10:</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Column 11:</td>
<td>4</td>
<td>1</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>Invalid 12:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Result 13:</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Result 14:</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Converge 15:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Finish 16:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Query Processing

- CPU: Nested Loop Join (NLJ) a la SQLite
- GPU: exploit 3D topological structure of CUDA threads
  - Assign source table to each thread dimension
  - Each thread represents a single entry in the cross-product
  - Write entries satisfying predicate to result table
Thread Mapping Scheme

\[
\begin{array}{|c|c|} \hline
T_1 & T_2 \\
\hline
c_1 & c_2 & c_2 & c_3 \\
1 & w & x & 5 \\
2 & z & z & 6 \\
3 & z & w & 7 \\
\hline
\end{array}
\]

\[
t_{1} \times t_{2} = \{(1,w,x,5), (1,w,z,6), (1,w,w,7), (2,z,x,5), (2,z,z,6), (2,z,w,7), (3,z,x,5), (3,z,z,6), (3,z,w,7)\}
\]
Thread Mapping Scheme

<table>
<thead>
<tr>
<th></th>
<th>$c_1$</th>
<th>$c_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$c_2$</th>
<th>$c_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

$T_1 \times T_2 =$

$\begin{align*}
(1, w, x, 5) & \times (1, w, z, 6) \times (1, w, w, 7) \\
(2, z, x, 5) & \times (2, z, z, 6) \times (2, z, w, 7) \\
(3, z, x, 5) & \times (3, z, z, 6) \times (3, z, w, 7)
\end{align*}$

Single Kernel Thread
Writing Results

Result Tablets

0 1 2 … 12 13

Tablet 0 → Tablet 1 → Tablet 2 → … → Tablet 12 → Tablet 13
Writing Results

Result Tablets

Grid of Threads

Thread Block
Writing Results

Result Tablets

Grid of Threads

Thread Block
Writing Results

Grid of Threads

Result Tablets

Thread Block
EXPERIMENTAL RESULTS
Test Machine

• Intel Core® i7 920 CPU @ 2.66 GHz
• NVIDIA GTX460 GPU
  – Fermi Microarchitecture
  – 336 CUDA Cores
  – 1 GB Memory
• NVIDIA GTX760 GPU
  – Kepler Microarchitecture
  – 1152 CUDA Cores
  – 2 GB Memory
• Linux 3.13.0-39-generic kernel
• CUDA 6.5, NVIDIA 340.29 Driver
Query Test Suite

• 10 Queries
  – 5 32-bit Integer / 5 32-bit Floating Point
• Random data (Both uniformly, normally distributed)
• All results based of 10 executions
Table 1: Running times in seconds for CPU and GPU execution of integer and floating-point arithmetic queries.

<table>
<thead>
<tr>
<th></th>
<th>CPU GTX460</th>
<th>GTX760</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1.183</td>
<td>0.304</td>
</tr>
<tr>
<td>Floating Pt.</td>
<td>1.127</td>
<td>0.279</td>
</tr>
<tr>
<td>All</td>
<td>1.155</td>
<td>0.617</td>
</tr>
</tbody>
</table>

Figure 5 graphically demonstrates the differences in running times on both the CPU and GPU for ten different join queries. The mean execution time for all ten queries on the CPU was 1.155 seconds, and the mean GPU execution time was 0.617 seconds and 0.291 seconds for the GTX460 and GTX760 respectively. Even numbered queries contain predominately integer arithmetic, with each subsequent odd-numbered query executing the same query with floating-point data. Table 1 lists the average running times for each of these two categories on both the GPUs and the CPU. There is no significant difference in values between these integer and floating-point queries, which indicates that speedup is independent of the data type. Additionally, the GPU executed faster than the CPU on average for both tests.

Figure 6(a) depicts the average running time for our suite of ten queries for increasing source table sizes, and figure 6(b) represents this data as speedups. Performance on smaller table sizes is less due to memory writes making up a greater portion of the total execution time. Nevertheless, the GPU implementation of the SQL virtual machine executes approximately twice to four times as fast as the CPU virtual machine on average for this query suite.

Queries 4 and 5 in Figure 5 executed more slowly on the GTX460 than the CPU, and the GTX760 executed in approximately equal time compared with the CPU. We hypothesize that this is due to result table sizes. While all other queries output 2.25 million rows or fewer, these two queries output approximately 6 million rows each. The additional time required to write these results across the PCI bus to the host memory significantly slowed execution time. We then conducted additional benchmarking in order to verify that the memory writes are the limiting step during query execution. To test this hypothesis, we incrementally increase the number of result rows in a cross-join of two, 3,000 row tables. As represented by Figure 7, the GPU becomes less efficient until the GPU executes in the same time as the CPU. For the GTX460, this occurs at approximately 1.8 million rows, and at 4.5 million rows for the GTX760. These values will vary with the computation required by the query and the layout of the desired data in the source tables. The PCI bus was also a limiting factor in our tests because the test machine only supported PCIe2. The GTX760 is designed to utilize the additional throughput of PCIe3, and this additional throughput would push the break-even point even closer to a full Cartesian product.
Figure 5: Queries show varied execution times (a) and speedup (b) on the GPU for different queries in the test suite.
Figure 5: Queries show varied execution times (a) and speedup (b) on the GPU for different queries in the test suite.
The Virginian Database System 

The Virginian database system is a distributed system that can be used to query large datasets. Our findings indicate that using a hardware virtual machine can improve the performance of the system by up to an order of magnitude faster than non-hardware virtual machines. We achieved this by using a virtual machine to distribute query processing across multiple GPUs. Our implementation demonstrated the benefits of using a virtual machine, but there are some limitations that need to be addressed. For example, memory writes are not currently optimized, and there is a PCI bottleneck that affects performance.

**Figure 7:** Proportion of Cross Product vs. Execution Time

**Figure 8:** Proportion of Rows Returned vs. Execution Time

- **CPU**
- **GTX460**
- **GTX760**

Proportion of Cross Product

Proportion of Rows Returned

Execution Time (s)

The current implementation technique is also limited to joining up to three tables. This is because of the three-dimensional nature of CUDA thread blocks. Because joins in SQL are commutative (the result of joining two tables is another table) in relational algebra, we can join more than three tables using a similar technique. In addition to improving the efficiency of the software implementation, we also demonstrated the benefits of using a virtual machine to distribute query processing across multiple GPUs.
Figure 7: As fewer rows in the Cartesian product are filtered, performance improves with lower cost of GPU hardware and its ubiquitous nature in modern computer systems. A framework such as this provides a low cost alternative to distributed RDBMSs for accelerating query processing.

Figure 8: Average running times with restrictive predicate for natural join shows substantial speedups. Compared with more expensive higher performance graphics cards, our hardware was relatively inexpensive with lower performance. Contrastive with the overall performance of query execution, but still benefit from GPU speedup in the general case.

In the current implementation, the limiting step in queries with joins, we anticipate such an advantageous with lower performance. Contrastive with the overall performance of query execution, but still benefit from GPU speedup in the general case. However, due to the high memory access nature of CUDA thread blocks, joins in SQL are highly dependent on the hardware configuration of the test machine; using different hardware may demonstrate different results.

The benchmarks associated with the Virginian framework are consistent for larger source table sizes, or will tablet management become less efficient. Because memory writes are coalesced writes to mapped memory, our multi-dimensional framework does not currently support coalesced memory accesses.

Tablet management becomes less efficient. Because memory writes are coalesced writes to mapped memory, our multi-dimensional framework does not currently support coalesced memory accesses. Although our framework does not currently support coalesced memory accesses, a feature which can be up to an order of magnitude faster than non-coalesced writes to mapped memory, our multi-dimensional framework includes an implementation to improve the performance of the GPU virtual machine.

In addition to improving the efficiency of the software implementation, another interesting research path would be multi-card dual use of the CPU- and GPU-based virtual machines. In such scenarios, pre-processing of the data could help to determine the most efficient virtual machine for query execution. In queries resulting in large amounts of data with relational nature of CUDA thread blocks, joins in SQL are highly dependent on the hardware configuration of the test machine; using different hardware may demonstrate different results.
Conclusions / Future Directions

• GPU implementation of VM-based query processor can be used to accelerate relational database joins
  – 2x-4x on average; 20x-60x in many common cases

• Scalability of a VM-based approach?
• Multiple GPUs to process queries
• Dynamically choose between CPU and GPU
Thank You!

Questions