2019: GPU ODYSSEY
Nikolay Sakharnykh, 8/26/19
MY LIFE AT NVIDIA

CUDA 1.0

FMA TFLOPS

Release year


G80
8800 GTX
0.7GB
86GB/s

Tesla
C1060
4GB
100GB/s

Fermi
C2070
6GB
144GB/s

Kepler
K40
12GB
288GB/s

Maxwell
M40
16GB
732GB/s

Volta
V100
32GB
900GB/s

Pascal
P100
16GB
32GB/s

Turing
RTX 8000
48GB
672GB/s

Pascal
RTX 8000
48GB
672GB/s

Volta
V100
32GB
900GB/s

Maxwell
M40
16GB
732GB/s

Kepler
K40
12GB
288GB/s

Fermi
C2070
6GB
144GB/s

Tesla
C1060
4GB
100GB/s

G80
8800 GTX
0.7GB
86GB/s


CUDA 10.1
AGENDA

- GPU chip
- Memory subsystem
- Multi-GPU
### VOLTA / TURING SM

<table>
<thead>
<tr>
<th></th>
<th>V100</th>
<th>TU102</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SMs</strong></td>
<td>80</td>
<td>72</td>
</tr>
<tr>
<td><strong>FP64</strong></td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td><strong>INT32</strong></td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td><strong>FP32</strong></td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td><strong>Tensor Cores</strong></td>
<td>8</td>
<td>8 (FP16 + INT)</td>
</tr>
<tr>
<td><strong>RT Core</strong></td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td><strong>Register File</strong></td>
<td>256 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td><strong>L1 and shmemb</strong></td>
<td>128 KB</td>
<td>96 KB</td>
</tr>
<tr>
<td><strong>Max threads</strong></td>
<td>2048</td>
<td>1024</td>
</tr>
</tbody>
</table>
GPU EXECUTION MODEL

Block of threads

SM 0

Block of threads

SM 1

Block of threads

SM 2

Grid of threads
WARP IMPLEMENTATION

Pre-Volta

32 thread warp

if (threadIdx.x < 16) {
    A;
    B;
}
else {
    X;
    Y;
}

Z;
WARP IMPLEMENTATION

Pre-Volta

Volta/Turing

PC, S
A; B;
X; Y;
Z;
A;
X;
B;
Y;
Z;
Z;
INDEPENDENT THREAD SCHEDULING

Communicating Algorithms

Pascal: Lock-Free Algorithms
Threads cannot wait for messages

Volta/Turing: Starvation Free Algorithms
Threads may wait for messages
ENABLE FAST MUTEXES FOR CONCURRENT DATA STRUCTURES

Concurrent Radix-Tree Lookup

- 1 thread
- 40 threads
- 163840 threads

Multi-threading (CPU)

Acceleration (RTX 2070)

CPU: Intel Xeon CPU E5-2690 v2 @ 3.00GHz (40 cores total)
GPU: Tesla V100-PCIe-16GB, ECC enabled

The diagram illustrates the layout of TENSOR CORE, a component designed for high-performance computing tasks. The matrix $A$ is represented in FP16 format, with elements $A_{i,j}$ where $i,j$ range from 0 to 3. Similarly, the matrix $B$ is also in FP16 format, with elements $B_{i,j}$, and the matrix $C$ is in FP16 or FP32 format, with elements $C_{i,j}$. The formula $D = AB + C$ is used to compute the result matrix $D$. The diagram visually demonstrates the interaction of these matrices within the TENSOR CORE architecture.
VOLTA TENSOR OPERATION

FP16 storage/input → Full precision product → Sum with FP32 accumulator → Convert to FP32 result

Also supports FP16 accumulator mode for inferencing

F16  
5-bit exponent 10-bit mantissa  
F32

F32
TENSOR SYNCHRONIZATION

Warp-synchronizing operation

Composed Matrix Multiply and Accumulate for 16x16 matrices

Result distributed across warp
COMPLETE GEMM HIERARCHY

Data reuse at each level of the memory hierarchy
CUTLASS
CUDA C++ Template Library for Matrix Algebra

SharedStoreIterator
SharedTileLoadIterator
MatrixMultiply
mma.sync
SharedStoreIterator
SharedLoadIterator
CUTLASS
CUDA C++ Template Library for Matrix Algebra

I am putting myself to the fullest possible use...

...which is all, I think, that any conscious entity can ever hope to do.
CUTLASS PERFORMANCE

GEMM kernels targeting Volta Tensor Cores natively with `mma.sync`

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Performance Relative to cuBLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>F16 accum, NN</td>
<td>79%</td>
</tr>
<tr>
<td>F16 accum, NT</td>
<td>71%</td>
</tr>
<tr>
<td>F16 accum, TN</td>
<td>78%</td>
</tr>
<tr>
<td>F16 accum, TT</td>
<td>68%</td>
</tr>
<tr>
<td>F32 accum, NN</td>
<td>93%</td>
</tr>
<tr>
<td>F32 accum, NT</td>
<td>63%</td>
</tr>
<tr>
<td>F32 accum, TN</td>
<td>57%</td>
</tr>
<tr>
<td>F32 accum, TT</td>
<td>57%</td>
</tr>
</tbody>
</table>
MAXIMIZING PERF WITH TENSOR CORE

125 TFLOPS FP16 performance in Tesla V100

Up to 5x throughput of FP32 on real DL workloads


<table>
<thead>
<tr>
<th></th>
<th>CUDA Cores</th>
<th>Tensor Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP64</td>
<td>FP32</td>
</tr>
<tr>
<td>Volta</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Turing</td>
<td>2</td>
<td>64</td>
</tr>
</tbody>
</table>

throughput (multiply-accumulates per SM per clock)
RT CORE

New in Turing

Ray Tracing acceleration
Exposed in NVIDIA OptiX
Easy interop with CUDA
Non-raytracing usecases
ANATOMY OF A RAY-TRACING APP

Intersection of rays and geometry

Arbitrary new rays started at arbitrary locations

Arbitrary operations at intersection points

Hierarchical spatial decomposition as acceleration structure
OPTIX
NVIDIA GPU Ray Casting API

High-level GPU accelerated ray-casting API

C (and C++)-API to setup scene and data

Multiple program domains, control per-ray payload

Flexible single ray programming model

Supports multi-GPU and NVLINK

Develop “to the algorithm“

https://developer.nvidia.com/optix
HOW IS THIS PROGRAMMED ON A GPU?

Ray Generation
Intersection
Bounding Box
Closest Hit
Any Hit

* per geometric primitive type
* per entry point
* per ray type
USING RT FOR YOUR APPLICATION

Integrating OptiX into CUDA app

Define geometry
Define hit/miss/intersect programs
Define ray gen program
Launch rays
Get data processed from rays

* per geometric primitive type
* per entry point
* per ray type
GENERATING SCENE

```cpp
optix::Geometry triangleMesh = context->createGeometry();

optix::Material material = context->createMaterial();
material->setAnyHitProgram(0,
    context->createProgramFromPTXFile(
        "kernels.ptx",
        "AnyHit"));

optix::GeometryInstance instance = context->createGeometryInstance();
instance->setGeometry(triangleMesh);
instance->setMaterialCount(1);
instance->setMaterial(0, material);

optix::GeometryGroup geometryGroup = context->createGeometryGroup();
geometryGroup->setAcceleration(
    context->createAcceleration("Trbvh", "Bvh"));
geometryGroup->addChild(instance);

collection["topObject"]->set(geometryGroup);
```
#include <optix.h>

rtDeclareVariable(uint, launchIndex, rtLaunchIndex, );
rBuffer<float3, 1> rayOrigins;
rBuffer<float3, 1> rayDirections;
rtDeclareVariable(rtObject, topObject, );

RT_PROGRAM void RayGeneration() {
    optix::Ray ray = optix::make_Ray(
        rayOrigins[launchIndex],
        rayDirections[launchIndex],
        0, 0.0f, RT_DEFAULT_MAX);
    rtTrace(topObject, ray, 0);
}
#include <optix.h>

rtDeclareVariable(uint, launchIndex, rtLaunchIndex, );
rtBuffer<float3, 1> rayOrigins;

RT_PROGRAM void AnyHit()
{
    float3 launchOrig = rayOrigins[launchIndex];
    float3 pos = launchOrig - ray.origin;
    // do shading or any other work..
    rtIgnoreIntersection();
}
LOCATING NEIGHBORS WITHIN A RANGE

Intersect Rays With Bounding Box Around Points Of Interest

For any arbitrary set of points

For a point P, find neighbors within a shape enclosed in a Bounding Box

Ray-based solution

1. Attach a box of width R to each point
2. Shoot one ray from P in arbitrary direction, \( t_{\text{max}} = 2R \)
3. Neighbors boxes will have either entry/exit intersection but never both.
4. Refine result points to any shape within the box in SM.
RAY TRACED NEAREST NEIGHBOUR SEARCH

Using RT-Cores Through OptiX RTX

KD-Tree vs. OptiX-RTX Neighbour Search

- Kdtree (Billion Search/s)
- Optix rtx (Billion search/s)
GPU CHIP: WRAP UP

- Independent thread scheduling
- TensorCores
- RTCores
GPU MEMORY SUBSYSTEM
## HOW FAST IS GPU MEMORY?

<table>
<thead>
<tr>
<th>GB/s</th>
<th>Peak bandwidth (sequential access)</th>
<th>GUPS 8B random atomic</th>
<th>GUPS 8B random read</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA Tesla V100</td>
<td>900</td>
<td>12.7</td>
<td>46.8</td>
</tr>
<tr>
<td>Intel® Xeon® Processor E5-2697A v4 (Broadwell)</td>
<td>76.8</td>
<td>2.5</td>
<td>4.9</td>
</tr>
</tbody>
</table>
GPU ACCELERATION OF DB OPERATIONS

32GB HBM

1TB+ DDR

Hash Table(s)

<table>
<thead>
<tr>
<th>Key</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
</tr>
</tbody>
</table>

Build & Probe

DB
GPU HASH TABLE PERFORMANCE

Bandwidth (GB/s) vs. Number of key-payload pairs (HT size = 2x the input size)

- Insert
- Query

- Workload fits in L2
- random read
- random CAS

Tesla V100

8B key + 8B payload, fixed hash table occupancy - 50%
select
  o_orderpriority,
  count(o_orderkey) as order_count,
from
  orders
where
  o_orderdate >= date '[DATE]' and
  o_orderdate < date '[DATE]' + interval '3' month and
  exists (select * from lineitem
    where l_orderkey = o_orderkey and
    l_commitdate < l_receiptdate)

  group by
    o_orderpriority,
order by
  o_orderpriority;
TPC-H Q4 - MAP TO GPU

```
query

select
  o_orderpriority,
  count(o_orderkey) as order_count,
from
  orders
where
  o_orderdate >= date '[DATE]' and 
  o_orderdate < date '[DATE]' + interval '3' month and
  exists (select * from lineitem
    where l_orderkey = o_orderkey and
    l_commitdate < l_receiptdate)

group by
  o_orderpriority,
order by
  o_orderpriority;
```

operations
- filter
- join
- groupby
- sort
TPC-H Q4 - MAP TO GPU

```sql
select
    o_orderpriority,
    count(o_orderkey) as order_count,
from
    orders
where
    o_orderdate >= date '[DATE]' and
    o_orderdate < date '[DATE]' + interval '3' month and
    exists (select * from lineitem
        where l_orderkey = o_orderkey and
        l_commitdate < l_receiptdate)

group by
    o_orderpriority,
order by
    o_orderpriority;
```
Q4 PERFORMANCE AND BOTTLENECK

SF1000

Bandwidth (GB/s)

CPU mem

GPU mem

0 20 40 60 80 100 120 140 160 180

0 20 40 60 80 100 120 140 160 180

DRAM BW
3x difference

PCIe bottleneck

V100  T4

152.5

54.8

12.2

11.5

11.5
HOW CAN WE IMPROVE?

Faster interconnect

- NVLINK2 CPU-GPU interconnect up to 150GB/s bidirectional
- V100
- P9

Compression

- l_orderkey decompression (GB/s) reading from CPU memory
- Uncompressed (8B)
- RLE+bp: 62
- RLE+Delta+RLE+: 113

Hybrid CPU-GPU

- CPU sequential BW is much higher than PCIe
- Filter data on the CPU before sending to the GPU
Q4: OVERCOMING PCIE BOTTLENECK

SF1000

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 + V100</td>
<td>12.6</td>
<td>22.3</td>
</tr>
<tr>
<td>P9 + V100</td>
<td>45.2</td>
<td>55.3</td>
</tr>
</tbody>
</table>
RAPIDS

- cuDF
- cuML
- cuGraph

DASK / SPARK

CUDA

PYTHON

DL FRAMEWORKS

cuDNN

APACHE ARROW on GPU Memory

Pandas-like

ScikitLearn-like

NetworkX-like
RAPIDS CUDF

**cuIO:**
- CSV, Parquet, ORC readers

**cuDF core:**
- Sort, join, groupby, transpose, filter
- Supported column types: int, dates, strings

**Reusable components:**
- Memory manager, concurrent hash map

---

**Fast readers:** GPU speed-up is 4x-6x

<table>
<thead>
<tr>
<th></th>
<th>Parquet (70M rows)</th>
<th>ORC (55M rows)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuDF</td>
<td>1.6</td>
<td>2.4</td>
</tr>
<tr>
<td>pyarrow</td>
<td>7.0</td>
<td>14.0</td>
</tr>
</tbody>
</table>

GPU: 1x NVIDIA Tesla P100
CPU: 2x Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz (40c)
SPATIAL DATA PROCESSING ON GPU
(COMING TO RAPIDS SOON)

27 polygons as ROIs and 1.3 million locations from AI City Challenge Dataset from City of Dubuque

Spatial refinement: theta join on a cartesian product

<table>
<thead>
<tr>
<th>runtime (ms)</th>
<th>GPU (Titan V)</th>
<th>CPU-Serial</th>
<th>CPU-Parallel (6 core, 12 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuSpatial</td>
<td>0.92</td>
<td>334.0</td>
<td>72.2</td>
</tr>
</tbody>
</table>

Point-in-Polygon Test
GPU MEMORY AND ANALYTICS: WRAP UP
NVIDIA® DGX-2™ SERVER AND NVSWITCH™

16 Tesla™ V100 32 GB GPUs
FP64: 125 TFLOPS
FP32: 250 TFLOPS
Tensor: 2000 TFLOPS
512 GB of GPU HBM2

Single-Server Chassis
10U/19-Inch Rack Mount
10 kW Peak TDP
Dual 24-core Xeon CPUs
1.5 TB DDR4 DRAM
30 TB NVMe Storage

12 NVSwitch Network
Full-Bandwidth Fat-Tree Topology
2.4 TBps Bisection Bandwidth
Global Shared Memory
Repeater-less

New NVSwitch Chip
18 2nd Generation NVLink™ Ports
25 GBps per Port
900 GBps Total Bidirectional Bandwidth
450 GBps Total Throughput
SINGLE GPU

- CPU
- PCIe Bus
- PCIe I/O
- L2
- HBM
- NVLink2
TWO GPUs
Can read and write each other’s memory over PCIe
TWO GPUS USING NVLINK

6 Bidirectional Channels Directly Connecting 2 GPUs
MULTIPLE GPUs

Directly connected using NVLink2

- Requires dedicated connections between GPUs
- Decreases bandwidth between GPUs as more are added
- Not scalable
ADDING A SWITCH

NVSwitch
FULL DGX-2 INTERCONNECT
Baseboard to baseboard

2.4TB/s bisection bandwidth
PARTITIONED HASH JOIN ON DGX-2

REPLICATED VS PARTITIONED HASH JOIN

DGX-2 TPC-H Q4 SF1000

Query time (s)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Query Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-only 2x Intel Xeon Platinum 8180</td>
<td>3.2</td>
</tr>
<tr>
<td>DGX-2 GPU HT, CPU input w/o compression</td>
<td>1.8</td>
</tr>
<tr>
<td>DGX-2 GPU HT, CPU input with compression</td>
<td>1.0</td>
</tr>
<tr>
<td>DGX-2 GPU HT, GPU input</td>
<td>0.06</td>
</tr>
</tbody>
</table>

lower is better

GPU OPTIMIZED SYSTEMS
With GPUDirect Storage
EXTREME FILE IO BANDWIDTH
SCALLED APPLICATION PERFORMANCE
Up to 20x Faster with GPU Accelerated TPC-H Query 4 Scaling on EXT4

https://devblogs.nvidia.com/gpudirect-storage/
GPU programming is easy
Explore TensorCores, RTCores

GPU memory is fast
Use it for OLAP

Scale up with NVLINK and NVSwitch
Use GPUDirect to avoid staging