High Performance with High Level Languages

Abstraction without regret?

- Blazegraph GPU Plug-in
- Blazegraph DASL

Bryan Thompson, Founder, Blazegraph
James Lewis, CUDA Researcher, Blazegraph

April 2016

http://blazegraph.com/
We enable Fortune 500 companies, startups, governments and research organizations achieve business objectives with graph data.
Blazegraph™ Open Source Platform: Proven and in Production

500+ weekly downloads
Thousands of active deployments

Powers Business with Billion Edge Datasets

Life Sciences
Precision Medicine

Cyber Defense
Intelligence

Financial Services
Fraud Detection

Information Management
Retrieval

Industrial IoT

http://blazegraph.com/

SYSTAP LLC DBA Blazegraph © 2006-2016 All Rights Reserved
Exploding Data Volumes Requires New Approach for Relationship Insight

Graph databases designed to analyze diverse entities and relationships. Today's datasets have billions of edges and nodes.

A (very small) Knowledge Graph

http://blazegraph.com/
Scalability Challenge is the Blazegraph Opportunity

The Blazegraph Platform

- Embedded and Single Server Deployments
- Enterprise High Availability
- Enterprise Scale-out Architecture

Data Scale (Edges)

Speed & Scale

- Fast
- Fastest

Single GPU (500+M)
- Multi-GPU Clusters (100+B)
- Embedded Single Server (50B)
- High Availability (50B)
- Scale Out (1T+)

Speed

http://blazegraph.com/
**GPUs – A Game Changer for Graph Analytics**

- **Graphs are a hard problem**
  - Non-locality
  - Data dependent parallelism
  - Memory bus and communication bottlenecks

- **GPUs deliver effective parallelism**
  - 10x+ memory bandwidth
  - Dynamic Granularity (Merrill)
  - Load balancing (Baxter)
  - Bi-directional (Beamer, Liu)

- **GPUs are hard to program…**

Graphic: Merrill, Garland, and Grimshaw, "GPU Sparse Graph Traversal", GPU Technology Conference, 2012
The Billion-Edge Graph Challenge: Scaling Up Requires the Right Paradigm and Hardware

CPU Cache Access Latencies in Clock Cycles

- Main memory: 167 clock cycles
- L3 Cache in Full Random access: 38 clock cycles
- L3 Cache in Page Random access: 18 clock cycles
- L3 Cache sequential access: 14 clock cycles
- L2 Cache in Full Random access: 11 clock cycles
- L2 Cache in Page Random access: 11 clock cycles
- L2 Cache sequential access: 11 clock cycles
- L1 Cache in Full Random access: 4 clock cycles
- L1 Cache in Page Random access: 3 clock cycles
- L1 Cache sequential access: 4.3 clock cycles

Graph Cache Thrash
The CPU just waits for graph data from main memory...

Access Latency Per Clock Cycle

https://datatake.files.wordpress.com/2015/09/latency.png
http://blazegraph.com/
Blazegraph Multi-GPU: Extreme Scale Traverse 1B Edges/Sec (GTEP) 40x more affordably!

Large Hadoop Cluster
$\sim$18M per GTEP

Cray XMT-2
$\sim$180K per GTEP

Blazegraph with GPU Clusters
$16K$ per GTEP (K40)
$4K$ per GTEP (Pascal)

1 GTEP = 1 Billion Traversed Edges Per Second
Easy to use. Native CUDA performance

• Native CUDA
  – Powerful, flexible
  – Very high barrier for use

• Vertex Centric API
  – Simple extensible abstraction
  – Templated kernels create adoption barrier

• SPARQL / Gremlin
  – Graph pattern matching languages (300x)
  – 20x over best in class multi-core CPU column-wise database.

• DASL (Spark)
  – Embedded DSL for Graph analytics (1000x)
200-300x Speed-up. NO changes in end user code

LUBM Query #9 U1000 (167,697,263 Edges w/Inference)

Blazegraph

```sparql
SELECT ?x ?y ?z
WHERE {
  ?x a ub:Student .
  ?y a ub:Faculty .
  ?z a ub:Course .
  ?x ub:advisor ?y .
  ?x ub:takesCourse ?z .
}
```

172,632 results
53,960ms on Blazegraph CPU

Blazegraph +

```sparql
SELECT ?x ?y ?z
WHERE {
  ?x a ub:Student .
  ?y a ub:Faculty .
  ?z a ub:Course .
  ?x ub:advisor ?y .
  ?x ub:takesCourse ?z .
}
```

290X

172,632 results
187ms on Blazegraph GPU

http://blazegraph.com/
## Blazegraph Architecture

<table>
<thead>
<tr>
<th>JNI</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manage Objects</td>
<td>Runtime Objects</td>
</tr>
<tr>
<td></td>
<td>(Vectors, Matrices, Relations, Indices)</td>
</tr>
</tbody>
</table>

### Query Optimization

- W3C SPARQL

### DASL Translator

- DASL
- DASL Translator

### Manage Objects

- JNI

### Execute Task Graphs

- C++

### Task Graph Scheduler

- NCCL / MPI
- NVIDIA CUDA
- NVIDIA NVLink

### Extensible Runtime Operators

- Task Graph Scheduler

### Other Technologies

- JNI
- C++
- NVIDIA CUDA
- NVIDIA NVLink
Design Philosophy

• Task graphs
  – Decompose work into
    • Operations
    • Data dependencies
  – Expose potential parallelism
  – Transparent scaling

• High level languages
  – No changes in user code
  – Native CUDA performance
  – Abstract away device and machine barriers

Operations:
SPMM, SPMV, Apply, Reduce, Construct Index, etc.

Data:
Matrix, Vector, Relation, Index, etc.

Task Graph:
DAG of Operations and Data Dependencies
Blazegraph DASL: GPU Acceleration for Data Analytics

Native CUDA performance with the ease of use of Spark and Scala.

Graph
- SSSP, Page Rank, BFS, CC, Louvain, Jaccard, …

Collaborative Filtering (Recommendation Systems)
- ALS, SVD, etc.

Supervised Neural Network Techniques
- Hidden Markov, DNN

Clustering
- Canopy, k-Means, Spectral Clustering, …

Dimensionality Reduction
- SVD, Lanczos, QR NNMF, …

http://blazegraph.com/
GPUs 700x-1800x Faster for Graphs Compared to Apache Spark on 700M and 1.98B edge graph
Anatomy of a DASL algorithm

``` scala
abstract class SSSPWorker(wmat: Matrix[Float], srcId: Int) extends DASLAlgorithm[Vector[Float]] {
  /***************************************/
  /* Return true if the algorithm assumes zero-weight self-edges */
  def assumesDiagonalsZero() : Boolean
  /*******************************/
  /************
  /* Initialize the current-distance vector such that it uses the Semiring
  * (Float, min, plus) as default Semiring for the following computation
  * populate the current-distance vector as follows: shortest path to the
  * source vertex is 0, all other elements will be set to the
  * identity of the Semiring, that is, Float.PositiveInfinity.
  * identity of the Semiring, that is, Float.PositiveInfinity.
  ************
  def init(daslContext : DASLContext) : Vector[Float] = {
    daslContext.vectorGenerator.createVector(
      size = wmat.numOfRows,
      dfltSemiring = wmat.dfltSemiring, // MinPlusSemiring
      dfltValue = wmat.dfltSemiring.addIdentity, // PositiveInfinity
      data = { srcId, 0F })
  }
  /************
  /* Run the SSSP algorithm, returning the shortest paths. */
  def run(daslContext : DASLContext) : Vector[Float] = {
    return dd
  }
}
```
2D Partitioning (aka Vertex Cuts)

- p x p compute grid
  - Edges in rows/cols
  - Minimize messages
    • log(p) (versus p^2)
    - One partition per GPU
- Batch parallel operation
  - Grid row: out-edges
  - Grid column: in-edges
- Representative frontiers

- Parallelism – work must be distributed and balanced.
- Memory bandwidth – memory, not disk, is the bottleneck
Stay in Touch

Bryan Thompson, Founder
bryan@blazegraph.com

Brad Bebee, CEO
beebs@blazegraph.com

http://blazegraph.com
@blazegraph