Large Graph on multi-GPUs

Enrico Mastrostefano\textsuperscript{1} Massimo Bernaschi\textsuperscript{2} Massimiliano Fatica\textsuperscript{3}

\textsuperscript{1}Sapienza Università di Roma
\textsuperscript{2}Istituto per le Applicazioni del Calcolo, IAC-CNR, Rome, Italy
\textsuperscript{3}NVIDIA Corporation

May 11, 2012
1. Large Graphs and Supercomputing
2. Graph500 on multi-GPUs
3. Breadth First Search on multi-GPUs
4. Sort-Unique Breadth First Search
5. Results
Large Graphs
Most of graph algorithms have low arithmetic intensity and irregular memory access patterns.
Large Graphs

- Most of graph algorithms have low arithmetic intensity and irregular memory access patterns.

- How do modern architectures perform running such algorithms?
Most of graph algorithms have low arithmetic intensity and irregular memory access patterns.

How do modern architectures perform running such algorithms?

Large-scale benchmark for data-intensive application.
Most of graph algorithms have low arithmetic intensity and irregular memory access patterns.

How do modern architectures perform running such algorithms?

Large-scale benchmark for data-intensive application

“This is the first serious approach to complement the Top 500 with data-intensive applications ...” (from www.graph500.org)
Large Graphs

- Most of graph algorithms have low arithmetic intensity and irregular memory access patterns
- How do modern architectures perform running such algorithms?
- Large-scale benchmark for data-intensive application
- “This is the first serious approach to complement the Top 500 with data-intensive applications ...” (from www.graph500.org)
- The core of the benchmark is a set of graph algorithms
graph500: specifications

**Generator**
- Generate the edge list with **real-world** properties (RMAT generator).
- Minimum size: $2^{28}$ vertices and $16 \times 2^{28}$ edges (268, 435, 456 vertices and 8, 589, 934, 592 edges)
graph500: specifications

**Generator**
- Generate the edge list with real-world properties (RMAT generator).
- Minimum size: $2^{28}$ vertices and $16 \times 2^{28}$ edges
  (268, 435, 456 vertices and 8, 589, 934, 592 edges)

**Kernel 1**
- Build the data structure that represents the graph: timed!
- Data structure cannot be modified by subsequent kernels
graph500: specifications

**Generator**
- Generate the edge list with real-world properties (RMAT generator).
- minimum size: $2^{28}$ vertices and $16 \times 2^{28}$ edges
  (268, 435, 456 vertices and 8,589,934,592 edges)

**Kernel 1**
- Build the data structure that represent the graph: timed!
- Data structure cannot be modified by subsequent kernels

**Kernel 2**
- Perform a Breadth First Search (BFS) visit starting from a random vertex: timed!
- Output the parent array.
Outputs

- Execution time of K1
- Execution time of K2,
- **TEPS**: Traversed Edges Per Second in the BFS (actually the ranking is based only on TEPS)
Outline

1. Large Graphs and Supercomputing
2. Graph500 on multi-GPUs
3. Breadth First Search on multi-GPUs
4. Sort-Unique Breadth First Search
5. Results
Generate the edge list

**Edge list**

- We have to generate: \( |V| = 2^{SCALE}; |M| = 16 \times 2^{SCALE} \)
- Each task generates a subset of the edge list in the form: \((U_0, V_0), (U_1, V_1), \ldots\)
- Edges are assigned to processes via a simple rule: edge \((U_i, V_j) \in P_k\) if \(U_i \mod \#P = k\)
Build the data structure

Data structure
- We transform the edge list in a Compressed Sparse Row (CSR) data structure
- CSR is simple and has minimal memory requirements

Compressed Sparse Row (CSR)

- Local vertices: \( u_1, u_2, u_3, u_4, u_n \)
- Offset array: 0, 3, 3, 7, 7, 7, 7, 18
- Adj. list: 0, 3, 7, l, M
Build the data structure

Data structure
- We transform the edge list in a Compressed Sparse Row (CSR) data structure
- CSR is simple and has minimal memory requirements

Compressed Sparse Row (CSR)

The core of the algorithm is a sort
K1 results

Weak scaling plot, Kernel 1

- Reference K1
- Sort K1

Time (sec) vs. number of processors
Outline

1 Large Graphs and Supercomputing
2 Graph500 on multi-GPUs
3 Breadth First Search on multi-GPUs
4 Sort-Unique Breadth First Search
5 Results
Queue based BFS

- Each vertex $U_i$ of $Q_{BFS}$ is assigned to one thread $t_i$
- Each thread $t_i$ visits all the neighboring $V_j$ of its vertex
- If $V_j$ is local: visit it
- If $V_j$ is not local: send to its owner
- receive vertices $V_k$ from other processes
Straightforward BFS

Queue based BFS

- Each vertex $U_i$ of $Q_{BFS}$ is assigned to one thread $t_i$
- Each thread $t_i$ visits all the neighboring $V_j$ of its vertex
- If $V_j$ is local: visit it
- If $V_j$ is not local: send to its owner
- receive vertices $V_k$ from other processes
Straightforward BFS

Queue based BFS

- Each vertex $U_i$ of $Q_{BFS}$ is assigned to one thread $t_i$
- Each thread $t_i$ visits all the neighboring $V_j$ of its vertex
- If $V_j$ is local: visit it
- If $V_j$ is not local: send to its owner
- Receive vertices $V_k$ from other processes
Queue based BFS

- Each vertex $U_i$ of $Q_{BFS}$ is assigned to one thread $t_i$
- Each thread $t_i$ visits all the neighboring $V_j$ of its vertex
- If $V_j$ is local: visit it
  - update the predecessor array
  - enqueue $V_j$
- if $V_j$ is not local: send to its owner
- receive vertices $V_k$ from other processes
Queue based BFS

Each vertex $U_i$ of $Q_{BFS}$ is assigned to one thread $t_i$

Each thread $t_i$ visits all the neighboring $V_j$ of its vertex

If $V_j$ is local: visit it
  ▶ update the predecessor array
  ▶ enqueue $V_j$

if $V_j$ is not local: send to its owner

receive vertices $V_k$ from other processes
Straightforward BFS

Queue based BFS

- Each vertex $U_i$ of $Q_{BFS}$ is assigned to one thread $t_i$
- Each thread $t_i$ visits all the neighboring $V_j$ of its vertex
- If $V_j$ is local: visit it
  - update the predecessor array
  - enqueue $V_j$
- If $V_j$ is not local: send to its owner
- receive vertices $V_k$ from other processes
  - update the predecessor array
  - enqueue $V_k$
Straightforward BFS

Queue based BFS

- Each vertex $U_i$ of $Q_{BFS}$ is assigned to one thread $t_i$
- Each thread $t_i$ visits all the neighboring $V_j$ of its vertex
- If $V_j$ is local: visit it
  - update the predecessor array
  - enqueue $V_j$
- if $V_j$ is not local: send to its owner
- receive vertices $V_k$ from other processes
  - update the predecessor array
  - enqueue $V_k$
- Output parent array and TEPS

Adj list

Enrico Mastrostefano
Large Graph on multi-GPUs
May 11, 2012 12 / 32
Straightforward BFS: Results

Weak Scaling plot:

- Ideal scaling
- Reference BFS
- Straight BFS

Single GPU TEPS:

- Reference BFS
- Straight BFS

Enrico Mastrostefano
Large Graph on multi-GPUs
May 11, 2012 13 / 32
Straightforward BFS: Issues

Gpu-related issues

- Threads workloads are unbalanced when threads visit different adjacency lists.
Straightforward BFS: Issues

Gpu-related issues

- Threads workloads are unbalanced when threads visit different adjacency lists
- Memory access patterns can be irregular and threads that belong to the same warp may access non-contiguous regions of memory
Straightforward BFS: Issues

Gpu-related issues

- Threads workloads are unbalanced when threads visit different adjacency lists.
- Memory access patterns can be irregular and threads that belong to the same warp may access non-contiguous regions of memory.

Algorithms rely on the use of **Atomic Add**
Straightforward BFS: Issues

MPI-related issues

- We don’t process non-local vertices so the array to send contains multiple copies of the same vertices.
- Multiple copies of the same vertex are sent to the owner.
Straightforward BFS: Issues

MPI-related issues

- We don’t process non-local vertices so the array to send contains multiple copies of the same vertices.
- Multiple copies of the same vertex are sent to the owner.

![Diagram showing Graph traversal with MPI-related issues]
Outline

1. Large Graphs and Supercomputing
2. Graph500 on multi-GPUs
3. Breadth First Search on multi-GPUs
4. Sort-Unique Breadth First Search
5. Results
Beyond the straightforward BFS

We use \( k \) threads to visit neighbors. We want to use as many threads as the number of neighbors. Neighbors of \( U \) are not-contiguous in the Adjacency list array. We want a contiguous array of neighbors. We send/recv multiple copies. We want to prune the array that we send.
Beyond the straightforward BFS

- We use $k$ threads to visit neighbors
- We want to use as many threads as the number of neighbors

Neighbors of $U$ are not-contiguous in the Adjacency list array

We want a contiguous array of neighbors.

We send/recv multiple copies

We want to prune the array that we send
Beyond the straightforward BFS

- We use $k$ threads to visit neighbors
- We want to use as many threads as the number of neighbors
- Neighbors of U are not-contiguous in the Adjacency list array
- We want a contiguous array of neighbors.
Beyond the straightforward BFS

- We use $k$ threads to visit neighbors
- We want to use as many threads as the number of neighbors
- Neighbors of U are not-contiguous in the Adjacency list array
- We want a contiguous array of neighbors.
- We send/recv multiple copies
- We want to prune the array that we send
Beyond the straightforward BFS: Sort-Unique BFS overview

What we will do is:

1. Compute the total number of neighbors, say $m$
2. Start $m$ threads, read the Adjacency list and build a contiguous array of neighbors (We call this array **Next Level Frontier**)
3. With $m$ threads prune the **Next Level Frontier**
4. Exchange vertices with other processes and update the parent array
Sort-Uniq BFS
Recipe #1: compute the total number of neighbors

- **Start** \( k \) **threads**, each element of \( Q_{\text{BFS}} \) is assigned to one thread

- Build \( Q_{\text{deg}} \), substituting each vertex with its degree

- Perform a **prefix-sum** operation on \( Q_{\text{deg}} \) to build the **New Offset** array
Sort-Uniq BFS
Recipe #1: compute the total number of neighbors

- **Start** $k$ **threads**, each element of $Q_{BFS}$ is assigned to one thread

- **Build** $Q_{deg}$, substituting each vertex with its degree

- **Perform a** prefix-sum **operation** on $Q_{deg}$ to build the **New Offset** array
Sort-Uniq BFS

Recipe #1: compute the total number of neighbors

- **Start** $k$ threads, each element of $Q_{BFS}$ is assigned to one thread

- **Build** $Q_{deg}$, substituting each vertex with its degree

- **Perform a** prefix-sum **operation on** $Q_{deg}$ **to build the** New Offset **array**

```
QBFS U3 U7 U21 U16 Ui Uj
1 2 3 4 k-1 k
QBFS U3 U7 U21 U16 Ui Uj
Qdeg d3 d7 d21 d16 di dj
prefix-sum
New Off off3 off7 off21 off16
prefix-sum
New Off off3 off7 off21 off16 offi offj
```

The last element of New Offset is:

$$m = \sum_{i \in Q_{BFS}} d_i$$
Sort-Uniq BFS

Recipe #1: compute the total number of neighbors

- **Start** $k$ threads, each element of $Q_{BFS}$ is assigned to one thread

- **Build** $Q_{deg}$, substituting each vertex with its degree

- Perform a **prefix-sum** operation on $Q_{deg}$ to build the **New Offset** array

The last element of **New Offset** is: $m = \sum_{i \in Q_{BFS}} d_i$
Recipe #2: build a contiguous array of neighbors

- Start \( m \) threads

- Each thread performs a **binary search** on **New Offset** and finds its index

- Each thread reads from the Adj list the element corresponding to the index

- and write it in the **Next Level Frontier**.
Recipe #2: build a contiguous array of neighbors

- Start $m$ threads

- Each thread performs a **binary search** on New Offset and finds its index

- Each thread reads from the Adj list the element corresponding to the index

- and write it in the **Next Level Frontier**.
Recipe #2: build a contiguous array of neighbors

- Start $m$ threads
- Each thread performs a **binary search** on **New Offset** and finds its index
- Each thread reads from the Adj list the element corresponding to the index
- and write it in the **Next Level Frontier**.
Recipe #2: build a contiguos array of neighbors

- Start $m$ threads
- Each thread performs a **binary search** on New Offset and finds its index
- Each thread reads from the Adj list the element corresponding to the index
- and write it in the **Next Level Frontier**.
Sort-Uniq BFS
Recipe #3: prune the Next Level Frontier

- Start $m$ threads

- Perform a **sort-uniq** operation on the **Next Level Frontier** (by using thrust library)

- and compact it to $n$ unique elements
Recipe #3: prune the Next Level Frontier

- Start $m$ threads
- Perform a **sort-uniq** operation on the **Next Level Frontier** (by using thrust library)
- and compact it to $n$ unique elements
Sort-Uniq BFS

Recipe #3: prune the Next Level Frontier

- Start $m$ threads

- Perform a sort-uniq operation on the Next Level Frontier (by using thrust library)

- and compact it to $n$ unique elements

![Diagram showing the process of sorting and compacting the Next Level Frontier](image-url)
Sort-Uniq BFS
Recipe #3: prune the Next Level Frontier

- Start \( m \) threads
- Perform a sort-uniq operation on the Next Level Frontier (by using thrust library)
- and compact it to \( n \) unique elements

Unique ratio \( \frac{m}{n} \sim 20 \)
Sort-Uniq BFS: communication and enqueue
Recipe #4: Exchange vertices and update the parent array

- Start \( n \) threads
- Substitute vertices with tasks
- Sort by tasks (by using thrust library)
- Exchange non-local edges
- Update the array of predecessors and Enqueue

\[ \begin{array}{c}
\text{Compact array} \\
V_i \quad V_k \quad \ldots \quad \ldots n
\end{array} \]

If \( Q_{BFS} = 0 \) quit.
Sort-Uniq BFS: communication and enqueue

Recipe #4: Exchange vertices and update the parent array

- Start $n$ threads
- Substitute vertices with tasks
- Sort by tasks (by using thrust library)
- Exchange non-local edges
- Update the array of predecessors and Enqueue
Sort-Uniq BFS: communication and enqueue
Recipe #4: Exchange vertices and update the parent array

- Start $n$ threads

- Substitute vertices with tasks

- Sort by tasks (by using thrust library)

- Exchange non-local edges

- Update the array of predecessors and Enqueue

Enrico Mastrostefano
Large Graph on multi-GPUs
May 11, 2012 22 / 32
Sort-Uniq BFS: communication and enqueue

Recipe #4: Exchange vertices and update the parent array

- Start $n$ threads
- Substitute vertices with tasks
- Sort by tasks (by using thrust library)
- Exchange non-local edges
- Update the array of predecessors and Enqueue

$$\text{MPI - Exchange}$$
Sort-Uniq BFS: communication and enqueue
Recipe #4: Exchange vertices and update the parent array

- Start \( n \) threads
- Substitute vertices with tasks
- Sort by tasks (by using thrust library)
- Exchange non-local edges
- Update the array of predecessors and Enqueue

Diagram:
- Compact array: \( V_i, V_k, \ldots, V_n \)
- Procs array: \( P_0, P_1, P_0, P_2, P_1, P_2, \ldots, P_s \)
- Procs sorted: \( P_0, P_0, P_1, P_1, P_1, P_2, P_2, \ldots, P_s \)
- MPI - Exchange
- Enqueue

If \( Q_{BFS} = 0 \) quit.
Sort-Uniq BFS: communication and enqueue

Recipe #4: Exchange vertices and update the parent array

- Start \( n \) threads
- Substitute vertices with tasks
- Sort by tasks (by using thrust library)
- Exchange non-local edges
- Update the array of predecessors and Enqueue

If \( Q_{BFS} == 0 \) quit.
Outline

1  Large Graphs and Supercomputing
2  Graph500 on multi-GPUs
3  Breadth First Search on multi-GPUs
4  Sort-Unique Breadth First Search
5  Results
Sort-Uniq BFS: Results

Weak Scaling plot

- Ideal scaling
- Reference BFS
- Straight BFS
- S-U BFS

Number of processors vs. TEPS (Billion)
K2: balancing

Computations and communications among processes are well balanced
K2: cuda kernels times

Sum of running time over bfs levels, proc 0 of 64

Cuda Kernels
- binary search
- sort-unique
- mpi allreduce
- enqueue-received
- cuda-cpy recvbuff

Communications
- mpi allgather
- mpi send/recv
- mpi allreduce

CudaCpy
- cuda-cpy sendbuff
- cuda-cpy recvbuff
K2: cuda kernels times

Sum of running time over bfs levels, proc 0 of 64

Cuda Kernels
- binary search
- sort-unique
- mpi allgather
- enqueue-local
- mpi send/recv
- cuda-cpy sendbuff
- mpi allreduce
- enqueue-received
- cuda-cpy recvbuff

Communications
- mpi allgather
- mpi send/recv
- mpi allreduce

CudaCpy
- cuda-cpy sendbuff
- cuda-cpy recvbuff
K2: cuda kernels times

Sum of running time over bfs levels, proc 0 of 64

Cuda Kernels
- binary search
- sort-unique
- mpi allgather
- enqueue-local
- mpi send/recv
- cuda-cpy sendbuff
- mpi allreduce
- enqueue-received
- cuda-cpy recvbuff

Communications
- mpi allgather
- mpi send/recv
- mpi allreduce

CudaCpy
- cuda-cpy sendbuff
- cuda-cpy recvbuff
# The Graph 500 List

## Complete Results - November 2011

<table>
<thead>
<tr>
<th>Rank</th>
<th>Machine</th>
<th>Owner</th>
<th>Problem Size</th>
<th>TEPS</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NNSA/SC Blue Gene/Q Prototype II (4096 nodes / 65,536 cores)</td>
<td>NNSA and IBM Research, T.J. Watson</td>
<td>32</td>
<td>254,349,000,000</td>
<td>Custom</td>
</tr>
<tr>
<td>2</td>
<td>Hopper (1800 nodes / 43,200 cores)</td>
<td>LBL</td>
<td>37</td>
<td>113,368,000,000</td>
<td>Custom</td>
</tr>
<tr>
<td>2</td>
<td>Lomonosov (4096 nodes / 32,768 cores)</td>
<td>Moscow State University</td>
<td>37</td>
<td>103,251,000,000</td>
<td>Custom</td>
</tr>
<tr>
<td>3</td>
<td>TSUBAME (2732 processors / 1366 nodes / 16,392 CPU cores)</td>
<td>GSIC Center, Tokyo Institute of Technology</td>
<td>36</td>
<td>100,366,000,000</td>
<td>Custom</td>
</tr>
<tr>
<td>4</td>
<td>Jugene (65,536 nodes)</td>
<td>Forschungszentrum Jülich</td>
<td>37</td>
<td>92,876,900,000</td>
<td>Custom</td>
</tr>
<tr>
<td>18</td>
<td>Blacklight (512 processors)</td>
<td>PSC</td>
<td>32 (Small)</td>
<td>4,452,270,000</td>
<td>Custom</td>
</tr>
<tr>
<td>19</td>
<td>Todi (176 AMD)</td>
<td></td>
<td>3,059,970,000</td>
<td>Custom GPU Result</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Interlagos, 176 NVIDIA Tesla X2090)</td>
<td>CSCS</td>
<td>28</td>
<td>3,059,970,000</td>
<td>Custom GPU Result</td>
</tr>
<tr>
<td>20</td>
<td>Dingus (Convey HC-1ex - 1 node / 4 cores, 4 FPGAs)</td>
<td>SNL</td>
<td>28</td>
<td>1,758,682,718</td>
<td>Convey Custom</td>
</tr>
</tbody>
</table>
The Graph 500 List

Complete Results - November 2011

<table>
<thead>
<tr>
<th>Rank</th>
<th>Machine</th>
<th>Owner</th>
<th>Problem Size</th>
<th>TEPS</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NNSA/SC Blue Gene/Q Prototype II (4096 nodes / 65,536 cores)</td>
<td>NNSA and IBM Research, T.J. Watson</td>
<td>32</td>
<td>254,349,000,000</td>
<td>Custom</td>
</tr>
<tr>
<td>2</td>
<td>Hopper (1800 nodes / 43,200 cores)</td>
<td>LBL</td>
<td>37</td>
<td>113,368,000,000</td>
<td>Custom</td>
</tr>
<tr>
<td>2</td>
<td>Lomonosov (4096 nodes / 32,768 cores)</td>
<td>Moscow State University</td>
<td>37</td>
<td>103,251,000,000</td>
<td>Custom</td>
</tr>
<tr>
<td>3</td>
<td>TSUBAME (2732 processors / 1366 nodes / 16,392 CPU cores)</td>
<td>GSIC Center, Tokyo Institute of Technology</td>
<td>36</td>
<td>100,366,000,000</td>
<td>Custom</td>
</tr>
<tr>
<td>4</td>
<td>Eugene (65,536 nodes)</td>
<td>Forschungszentrum Jülich</td>
<td>37</td>
<td>92,876,900,000</td>
<td>Custom</td>
</tr>
</tbody>
</table>

18  Blacklight (512 processors) | PSC | 32 (Small) | 4,452,270,000 | Custom |
19  Todi (176 AMD Interlagos, 176 NVIDIA Tesla X2090) | CSCS | 28 | 3,059,970,000 | Custom GPU Result |
20  Dingus (Convey HC-1ex - 1 node / 4 cores, 4 FPGAs) | SNL | 28 | 1,758,682,718 | Convey Custom |
Conclusions and Outlook

- To visit large graphs we need a distributed algorithm
- We are slower on a single GPU
- We relay on sorting to achieve better scaling
- If we can speed-up the sorting then we will speed up the BFS
D. Chakrabarti, D. Chakrabarti, Y. Zhan, and C. Faloutsos.
R-mat: A recursive model for graph mining.
*IN SDM*, 2004.

A. G. Duane Merrill, Michael Garland.
High performance and scalable gpu graph traversal.

P. Harish and P. J. Narayanan.
Accelerating large graph algorithms on the gpu using cuda, 2007.

J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and
Z. Ghahramani.
Kronecker graphs: An approach to modeling networks.

R. Murphy and G. Chukkapalli.
Graph 500 and data-intensive hpc.
Technical report, Oracle.
P. of the 16th ACM symposium on Principles and practice of parallel programming, editors.  
*Accelerating CUDA Graph Algorithms at Maximum Warp*, 2011.
[5] [1, 4] [3, 6, 2]
Random Graph
Unique ratio example

<table>
<thead>
<tr>
<th></th>
<th>0.28</th>
<th>0.47</th>
<th>0.18</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.16</td>
<td>0.15</td>
<td>0.21</td>
</tr>
<tr>
<td>4</td>
<td>0.77</td>
<td>0.50</td>
<td>0.97</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table: Unique ratio, proc 0 of 64, 3 run of BFS