Accelerating CUDA Graph Algorithms at Maximum Warp

Pervasive Parallelism Laboratory
Stanford University

Sungpack Hong, Sang Kyun Kim,
Tayo Oguntebi and Kunle Olukotun
Graph Analysis

- Fundamental data structure; random relationship between entities
- Wide usage of graph analysis
  - Social Networks, Computational Biology, ...
- Abundant data-level parallelism

Still, is computationally challenging
- Growing data size
- Expensive algorithms
  - e.g. betweenness centrality: $O(NM)$
- Random memory access
- Hard to partition for cluster execution
  (large surface to volume ratio)
Machines for Graph Analysis

- **Supercomputers** (e.g. Cray XMT)
  - Large, single shared address space
    - Uniform memory access time (cache-less)
  - Many processors, heavily multithreaded
    (parallelism, latency hiding)
  - Large memory bandwidth
  - But, rare and expensive

- **GPU architecture ~ supercomputers**

- **Difference**
  - GPU has limited memory capacity (a few GB; no VM)

Let’s use GPU as long as the problem size fits.
Example Algorithm: BFS

- Breadth First Search (BFS)
  - Starting from a node, visit all nodes in breadth-first order
  - Node visit at each level is parallel.
  - A building block for many other algorithm
- Assigns BFS level to each node
  - e.g. Kevin-Bacon Number
GPU Implementation [Harish et al, HiPC 2007]
- Frontier-expansion method
- Good for CUDA; no atomic operation required

**Pseudo-Code**

```plaintext
Foreach (v: G.Nodes)
    if (v.level == curr)
        Foreach (w: v.Nbrs)
            if (w.level == INF)
                w.level = curr + 1;

Root.level = curr = 0;
Repeat
    BFS_kernel(curr);
    curr++
Until not changed
```

**CUDA Code**

```plaintext
int v = THREAD_ID;
if (levels[v] == curr) {
    // iterate over neighbors
    int num_nbr = nodes[v+1]-nodes[v];
    int* nbrs = & edges[nodes[v]];
    for(int i = 0; i < num_nbr; i++) {
        int w = nbrs[i];
        if (levels[w] == INF) {
            levels[w] = curr + 1;
        }
    }
}
```

Each thread process a node
Previous Result

- Order of magnitude faster than CPU execution
- ... depending on the shape of input graph
  - 14x for Random Graph (Erdős–Renyi)
  - 1.3x for RMAT Graph (Kronecker)
  - with same # nodes (4M) and edges (48M)

Random: uniform degree distribution
RMAT: skewed degree distribution
... it means we’re in trouble

- Real-world graphs ➔ RMAT-like
- Nature of real-world graphs
  - Degree distribution follows power-law curve (skewed, long tail)
  [Barabasi et al, Science 1999]

There are nodes that have very high-degree!
Most nodes have low degrees
Remainder of This Talk

- Why GPUs don’t perform well
- Techniques for improving GPU performance
- Performance results
Overview: GPU Architecture

- **Thread-Block**
  - Mapped to a physical computation unit, Streaming Multiprocessor (SM)
  - True Multi-Processing

- **Warp (1TB = N warps)**
  - A SM is time-shared by N warps
  - Hardware Multi-Threading

- **Threads (1 Warp = 32 Threads)**
  - Single instruction on multiple data
  - In fact, they are vector lanes
  - SIMD
Overview: CUDA programming model

- CUDA provides little notion of warp, but assumes independent threads
- Hardware provides such illusion via
  - Thread divergence
  - Random (scattered) memory access
GPU HW: Divergence

- Threads (=lanes) in a warp are allowed to diverge and execute different instructions.
- However, divergent threads are *serialized*.

```c
......
Switch(THREAD_ID % 4) {
  case 0 : ... break;
  case 1 : ... break;
  case 2 : ... break;
  case 3 : ... break;
}
......
```

Threads in a warp
GPU HW: Random Memory Access

- Threads (=lanes) can do random memory access.
- Consecutive addresses ➔ Coalesced
- Scattered (non-consecutive) addresses ➔ Serialized

(possibly wasting memory BW)
Review: previous work

- Divergence + Random memory access
  - Gives an illusion of independent threads
  - But with a performance penalty
  - Degree skew exacerbates such penalty

\[
\text{Foreach } (v: G.\text{Nodes}) \\
\text{if } (v.\text{level} == \text{curr}) \\
\quad \text{Foreach } (w: v.\text{Nbrs}) \\
\quad \text{if } (w.\text{level} == \text{INF}) \\
\quad \quad w.\text{level} = \text{curr} + 1;
\]

Divergence + Load Imbalance = Big performance loss!
Our Techniques

1) Utilize warps (in a systematical way)
2) Virtualize warp-size
3) Other techniques – dynamic task-allocation (deferring outliers)
Technique #1: Utilizing Warps

- **Idea**
  - Use *warps*, instead of *threads* (to prevent divergence)
  - In a systematic way

- **Our Systematic Method**
  - Divide kernel into two phases
  - SISD phase (unit: warp)
    - Each *warp* processes one task.
  - SIMD phase (unit: thread)
    - Each *thread* processes one sub-task.
    - Initiated by explicit function call
  - Resembles classic SIMD programming
    - But easier (thread divergence and scattering during SIMD)
Applying Warp-centric Method

- No divergence or scattering
- Each Warp processes a node
- Each Thread processes its neighbors
- More parallelism in neighbor iteration
- No big workload imbalance (Neighbors of the same node)
- Short and balanced divergence ➔ Okay

```c
Foreach (v: G.Nodes)
  if (v.level == curr)
    Foreach (w: v.Nbrs)
      if (w.level == INF)
        w.level = curr + 1;
```

Implementation Issue

How to implement SISD Phase in CUDA?
- Without changing CUDA compiler or GPU HW

Redundant execution
- Every thread executes the *same* instruction on the *same* data.
- Okay because there is no race!
  - Instruction executions are synchronized.
  - Memory accesses are merged.

(see the paper for special care for atomic ops)
Sketch: New Code

BFS_KERNEL (...) {
    int v = WARP_ID; // THREAD_ID/WARP_SZ
    ...
    if (levels[v] == curr) {
        int num_nbr = nodes[v+1] - nodes[v];
        int* nbrs = & edges[nodes[v]];
        SIMD_BFS_Iter (THREAD_ID % WARP_SZ, ...);
    }
}

SIMD_BFS_Iter (int LANE_ID, ...) {
    for (i=LANE_ID; i<num_nbrs; i+=WARP_SZ) {
        int w = nbrs[i];
        // if not visited yet
        if (levels[w] == INF) {
            levels[w] = curr + 1;
        }
    }
    __threadfence_block();
}

Begins with SISD phase
Work based on Warp-ID
Explicit entrance to SIMD phase
SIMD phase; work based on Lane ID
Ensure visibility across the warp before back to SISD

(See the paper for C-Macro based simpler description)
Technique #2: Virtualize warps

- **Drawback of previous method:** under-utilization
  1. Amdahl’s law: SISD vs. SIMD ratio
  2. Data width: sub-task data-width < warp-size

- **Our solution:** virtualize warps
  - Logically partition a warp into $K$ virtual warps
  - Assign a task per *virtual warp*
  - Virtual warp-size = $1/K \times$ physical warp-size(=32)
  - May introduce divergence again

---

1. Amdahl’s Law
2. Sub-task data width

Trade-off divergence and underutilization

$K=2$
Implementing Virtual-Warps

- Use the same code as warp-centric method.
- Simply let warp-size as a template variable.
  - Execution is still correct.
  - Can explore trade-offs with this single variable.

```cpp
template <int WARP_SZ>
SIMD_BFS_Iter (…) {
    for(i=LAIN_ID;i<num_nbrs;i+=WARP_SZ){
        …..}
}

template <int WARP_SZ>
BFS_KERNEL (…) {
    int v = WARP_ID; // THREAD_ID/WARP_SZ
    …}
```
Technique #3: Dynamic load balance

- **Inter-warp load imbalance**
  - GPU HW thread-block scheduler:
    - SM is time-shared by multiple warps in a thread block.
    - SM is finished when all warps are finished.
    - One long-running warp prevents SM to finish.

- **Solution: Dynamic task allocation**
  - Each warp grabs a chunk of work from the work-queue.
  - (+) dynamic load balancing
  - (-) work queue overhead (atomic instruction)
BFS Results

- **Speed-up**
  - 1x: Single CPU execution
  - GPU: Nvidia GTX 275 (1.2 Ghz)
  - CPU: Intel Xeon E5345 (2.3Ghz, 8MB LLC)

**Dynamic load balance:**
overhead <-> benefit

<table>
<thead>
<tr>
<th>Name</th>
<th>Node</th>
<th>Edge</th>
<th>Skew</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMAT</td>
<td>4M</td>
<td>48M</td>
<td>High</td>
</tr>
<tr>
<td>Random</td>
<td>4M</td>
<td>48M</td>
<td>Low</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>~ 4.3M</td>
<td>~ 69M</td>
<td>High</td>
</tr>
<tr>
<td>Patents</td>
<td>~ 1.7M</td>
<td>~ 10M</td>
<td>Low</td>
</tr>
</tbody>
</table>

Dynamic load balance results for different datasets and configurations:

- **RMAT Random LiveJournal Patents**
  - Warp32 versus Warp32 + Dynamic

Warp method solves workload imbalance issue
BFS Results

- Virtual warp-size
  - Trade-off: under-utilization vs. load imbalance
  - Best warp-size depends on the graph instance.

No single best virtual warp-size

Under-utilization issue solved.

[Harish 2007]
Data-Size Scalability

- Scale size of the graph (RMAT instance)

- Little parallelism
- Stable speedup
- Heavy imbalance → Dynamic load balance
- under-utilization → Smaller virtual warps
Dynamic Workload Distribution

- Parameter: Chunk-size
- Overhead vs. Degree of imbalance

Graph showing execution time (ms) vs. chunk size. The graph indicates that overhead is relatively insensitive to chunk size, with the x-axis being log-scaled.
Other applications

- Selective applications from GPU Benchmarks
  - Applications having work-imbalance or scattering issues.
  - Baseline(1x) is previous GPU implementation.

Similarly benefited two other graph applications in [Harish2007]

Also benefited other four non-graph applications

Application without heavy work imbalance issue→ overheads
Summary

- **Graph Algorithm on GPU**
  - Large memory bandwidth + Parallelism
  - Workload imbalance issue (due to skewed degree distribution)

- **Virtual warp-centric method**
  - A systematic way of using warps in CUDA
  - Enables trade-off:
    - under-utilization vs. workload imbalance
  - Provides up to ~9x speedup to the previous *GPU* implementation
  - Works for other applications too