A GPGPU Algorithm for c-Approximate r-Nearest Neighbor Search in High Dimensions

Lee A. Carraher, Philip A. Wilsey, and Fred S. Annexstein

School of Electronic and Computing Systems
University of Cincinnati

May 24, 2013
Parallel Nearest Neighbor Search

- An approximate Nearest Neighbor algorithm in parallel
- GPGPU implementation of the Leech Lattice decoder for LSH NN
- Other non-sequential parts of LSH NN in parallel (projection, sorting)
- Optimized for the CUDA architecture
Motivations

- NN is an intuitive, powerful basis for machine learning algorithms
- Approximate answers are good enough for big data
- GPU provides many benefits and challenges
  - High ALU to Control allocation ratio
  - Remap for data intensive processing
  - Memory transfer bottlenecks (no MMU or cache)
  - Getting it to fit in shared memory
NN, KNN, $c$-approx NN, $cr$-approx NN

- NN problem: given a query vector, return the nearest vector in a set of vectors by minimizing some distance function.
- KNN is NN but the $K$ nearest vectors are returned (NN is KNN where $K=1$)
- $c$–approx NN returns a NN within a constant $\epsilon$-distances of the optimal NN
- $cr$–approx. NN extends the $c$-approx NN by returning all the vectors within a radius $r$ of the query vector
Problems With Other Parallel Search

- **GPU Linear Search**
  - Fast, simple implementation
  - Linear search complexity and speedup
  - All vectors must be in MM

- **GPU Kd-Tree Search**
  - Vectors do not have to be in MM
  - Work well for $d < 20$
  - Parallel DFS has issues scaling
Locality Sensitive Hashing for NN

- Nearest Neighbor search algorithm based on LSH functions
  - Collision probability corresponds to ‘closeness’
- Collision Probability is a pseudo-distance function
- Only consider colliding vectors
- Exhaustively search colliding vectors
Algorithm 3 Query: c-approx k-nearest neighbor

Require: \( \hat{x} \in \mathbb{R}^n, G, H, U(), D \)

\[
L = \[
\text{for all } g_j(x) \in G \text{ do}
L \leftarrow U(g_j(\hat{x}))
\text{end for}
K = \[
\text{for all } l \in L \text{ do}
d \leftarrow dist(\hat{x}, D[l])
K \leftarrow \{d, D[l]\}
\text{end for}
\text{sort}(K)
\text{return } K[0 : k]
\]
An Example Hash Family

Figure: Random Projection of $\mathbb{R}^2 \rightarrow \mathbb{R}^1$
The goal of ECCs are the same as a ’good’ LSH function

- Send data over a noisy channel and correct the errors
- Only a subset of data is valid (codes/hashes)
- Codes correct within an error correcting radius
- Shannon’s Limit (optimal codes exist, but are complex)
- Fast implementation (not as important as cpu clock >> channel capacity)

Need to find a compromise between decoding complexity and the native dimensionality of the ECC.
Comparison of ECC Decoders

- Leech Lattice partitions $R^{24}$, in 519 operations
GPU tweak: Avoid Branching

- Warps (sets of 16 threads) run as SIMD.
- Branches cause a warp to be rescheduled and run again.
  - 99.998% chance for uniform boolean test, \((1 - 2^{-16})\)
- Conversion provides a 68% Speedup for our algorithm

<table>
<thead>
<tr>
<th>Conditional Code</th>
<th>Sequential Access Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (\text{distA} &lt; \text{distB}):</td>
<td></td>
</tr>
<tr>
<td>(d_{ij}[4i]=\text{distA})</td>
<td>(d = \text{distA} &lt; \text{distB})</td>
</tr>
<tr>
<td>(d_{ijk}[4i]=\text{distB})</td>
<td>(d_{ij}[4i]=\text{distA}<em>d+\text{distB}</em>(-d))</td>
</tr>
<tr>
<td>(\text{kparities}[4i]=0)</td>
<td>(d_{ijk}[4i]=\text{distB}<em>d+\text{distA}</em>(-d))</td>
</tr>
</tbody>
</table>

ELSE:

| \(d_{ij}[4i]=\text{distB}\) | \(\text{kparities}[4i] = (\neg d)\) |
| \(d_{ijk}[4i]=\text{distA}\) | |
| \(\text{kparities}[4i]=1\) | |
Results For SIFT Vectors

Comparison of Time for LSH and Linear Search

- Linear Search Times
- LSH Search Times
Result Extrapolation For Larger DBs

Extrapolated Comparison of Time for LSH and Linear Search (Semilog)
Query Database:
Parallel/Total Compute Time for SIFT Samples

Ratio Par/Seq

SIFT sample data samples
avg=0.9340, R2=2.83203799077e-05
Scaled Speedup Results

Query: Total Speedup as a function of Parallel Speedup

- **Parallelism**
  - Our Theoretical Speedup (93.4%)
  - 90% Serial Code
  - 95% Serial Code
  - 97% Serial Code
  - Theoretical Speedup

Parallel Speedup

Total Speedup
Conclusions

- GPU parallel implementations of lattice hashing accelerates the NN algorithm
- Required sequential parts of the LSH algorithm will always cause bottlenecks
- Within the processor count range of some medium embedded applications (UAVs, Cellphones, Surveillance Systems) speedup scales.
- Tweaking parts of the LSH algorithm for the GPU
  - Random Projection Calculation (bulk or per query)
  - Hash list sorting and intersection (per-query)
  - Linear searching of candidate NNs (for exact NN)