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

Lee A. Carraher, Philip A. Wilsey, Fred S. Annexstein
School of Electronic and Computing Systems
University of Cincinnati
Cincinnati, USA
carrahl@email.uc.edu, philip.wilsey@uc.edu, fred.annexstein@uc.edu

Abstract—Nearest Neighbor search is often considered one of the simplest and most intuitive ideas in data mining. Nearest Neighbor search is often found to be the workhorse of a variety of data mining, machine learning, and computer vision algorithms. For very high dimensional data, the naive linear search tends to be optimal. This is due to the so-called curse of dimensionality in database search. To accelerate the search time, many researchers have turned to the power and efficiency of GPGPU computing; mainly for computing the data intensive distance metrics. Though promising speedups are achieved through this method, it fails to capitalize on many of the recent advances in algorithms for bounded error approximate nearest neighbor searches. Approximate nearest neighbor methods give satisfactory results, especially in very large and possibly redundant datasets, while boasting sub-linear search complexities.

In this paper we present a c-approximate r-nearest neighbor search algorithm for CUDA using Locality Sensitive Hash with Nearest Neighbor search (LSH-NN). We implement this system in CUDA and test it on real world image SIFT vector data. Our tests show that we are able to achieve significant speedup over the serial version, with good approximations on scalability, while achieving the near-optimal search complexity of LSH-NN.

Keywords: KNN, GPGPU, LSH, Nearest Neighbor

I. INTRODUCTION

Nearest neighbor (NN) is a very well-known problem throughout data analysis and is often utilized in a variety of different algorithms. NN is an active area of research in computer vision, genetic sequencing and string matching, similarity search in unstructured databases, data clustering and analysis, and a variety of other artificial intelligence applications. Due to its versatility, advances in the speed and throughput of NN search often result in the acceleration of other related data intensive algorithms utilizing NN. Data algorithms such as mean-shift clustering, point-cloud modeling for image analysis, and content based image retrieval are often accelerated by a fast NN search that limits their model update and search space to only relevant, or adjacent data.

In addition to the use of NN to accelerate data intensive applications, GPGPU computing has also become prominent as an algorithm accelerator. GPGPU computing has become an active area of development, but its application to general purpose computing is blocked by algorithms that do not easily lend themselves to efficient parallelization. One reason for this is that the ratio of control logic and cache to ALU transistor allocation on the GPU makes them efficient for some problems, and not suitable for others. For this reason, many algorithms need to adopt different methods of processing data in order to effectively utilize the GPU’s architecture.

One such algorithm that benefits nicely from GPGPU acceleration is nearest neighbor linear search [1]. The distribution and segmentation of computing vector distances is very similar in structure to many 3d graphics algorithms, for which GPUs were designed. For example, Garcia and Debreuve [1] achieve a speedup of nearly 10x over the highly optimize ANN library [2] (which is a serial implementation of approximate nearest neighbor) using exact nearest neighbor search on GPGPUs. Although promising, further analysis of their timing results shows that the speedup ratio decreases as the number of dimensions exceeds 32 and the number of vectors exceeds 19,200. This result comes from the inherent differences in the complexity of the two algorithms. While linear search complexity increases linearly, ANN search complexity is sub-linear where complexity is dependent upon the entropy of the data [3].

In this paper we investigate approximate nearest neighbor methods for a GPGPU. Approximate methods may seem like a compromise to the exact solution, however, in this paper we show that due to factors such as noise, data redundancy, and the Curse of Dimensionality (COD) approximate methods attain similar accuracy. For approximate nearest neighbor two widely used packages exist: FLANN [4] and the previously mentioned ANN [2]. Both of these methods employ tree like structures to perform the nearest neighbor search. Zhou et al [5] implements a K-d Tree search method for GPGPUs to perform various computer vision tasks and obtain a 7x to 11x speedup over an optimized sequential CPU algorithm. However, in Zhou the dataset was again limited in dimension and size of the search database. In general, K-d Tree search decomposes into linear search complexity as the number of dimensions increases [6]. This is due to the nature of the K-d Tree construction in
a string or list
\( \hat{x} \)
a permutation of \( x \)
\( \otimes \)
the concatenation operation
\( \oplus \)
the list append operation
\( H(x) \)
returns a hash digest
\( N(\mu, \sigma) \)
returns a normal variate
\( \Lambda_{23}(x) \)
the Leech lattice decoder
\( \text{map}(f(x), \text{GPUs}) \)
maps \( f(x) \) to the available SPEs

<table>
<thead>
<tr>
<th>Table I</th>
<th>ADDITIONAL NOTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2]</td>
<td></td>
</tr>
<tr>
<td>[2,1]</td>
<td></td>
</tr>
<tr>
<td>[2,2]</td>
<td></td>
</tr>
<tr>
<td>[3,5]</td>
<td></td>
</tr>
<tr>
<td>[4,6]</td>
<td></td>
</tr>
<tr>
<td>[5,5]</td>
<td></td>
</tr>
<tr>
<td>[2,2]</td>
<td></td>
</tr>
</tbody>
</table>

regards to the decreasing probability that a selected vector’s
chosen dimension will provide a balanced splitting plane. As
the number of dimensions increase K-d Tree construction
form deep chains of length \( n \). In FLANN and ANN, the
unbalanced splitting planes are addressed by creating bins
of vectors and using the median of the bins as the splitting
planes. Despite the more balanced tree in these approximate
methods, they still require a depth first search (DFS) tree
traversal, which is inherently sequential and does not provide
a parallel algorithm. Furthermore, in regards to ANN, the
DFS tree must also be recursed at times in order to backtrack
after a vector exceeds its given error tolerance. Backtracking,
in general must either employ recursion or a stack, both of
which are inefficient to implement on a GPGPU.

Using the above reasoning, in light of the limitations
of the GPGPU hardware, realizable parallel algorithms,
and the desire for a sub-linear nearest neighbors search
complexity, we suggest the use of the locality sensitive hash
nearest neighbor (LSH-NN) search algorithm of Indyk [8].
In this paper we will present a parallel c-approximate r-
nearest neighbor search algorithm for the CUDA GPGPU framework.
In addition we show that our algorithm is able to achieve
reasonable speedup relative to the number of sequential
processing elements (SPEs) and the same complexity bound
of Indyk’s LSH-NN algorithm.

The remainder of this paper is organized as follows.
Section II presents some background information. Section
III presents the serial LSH algorithm and discusses some of
its time complexity. Section IV presents the parallel LSH
algorithm and some of the optimizations we made for the
CUDA execution hardware. Throughout the algorithms pre-
sented in Sections III and IV we use several key operators
that might be non-intuitive; definitions for these operators
are provided in Table I. Section V reviews the performance
results from our experiments. Finally, Section VI contains
some concluding remarks.

II. BACKGROUND

A. Linear/Parallel NN

We begin by first describing the linear nearest neighbor
search algorithm.

Definition II.1. Exact NN [9] Given a set of points \( P \)
in \( \mathbb{R}^d \) and query point \( q \) return a point \( p \in P \) such that
\( p = \text{Argmin} \{ \text{dist}(p', q) \} \), where dist is some metric function.

The above definition suggest a natural and straightforward
algorithm, that we omit here, for finding their nearest
neighbor of a query vector in a dataset. A similarly apparent
parallel version can also be realized through simple loop
parallelism either by distributing the elements of a vector
across GPGPUs and summing as in parallel dot product
algorithms, or by computing multiple candidate data vectors
over the GPGPUs in parallel. This approach is shown in
Algorithm I. This algorithm (or its transpose) attains
linear speed up corresponding to the number of available
processor units. However, as stated in the introduction, this
bound remains linear for a fixed number of processors.

Algorithm 1. Brute Force Parallel Nearest Neighbor Search

Require: \( q \in \mathbb{R}^{n}, X = \{x_1, \ldots, x_m\}, x_k \in \mathbb{R}^{n}, \text{GPUs} \)
\( \text{min} = \emptyset \)

for all \( i \in \text{Range}(m) \) do
\( S \leftarrow \text{map}(\text{dist}(x_{i+\text{GPU.id}}, q)) \)
\( i = i + \text{len}(\text{GPUs}) \)
\( \text{min} = \argmin(S) \)
end for

return \( \text{min} \)

B. K-d Trees

K-d trees partition d-dimensional space by creating sepa-
rating hyperplanes based on the value of a data vector
at different dimensions. As the tree is generated, each
node corresponds to a separating hyperplane in the data
space in the \( i^{th} (\text{mod } d) \) dimension corresponding to the
\( i^{th} \) level of the tree. By using this data structure, a nearest
neighbor search is able to avoid calculating distance metrics
exhaustively over all data vectors. Figure 1 shows a tree
for the dataset \( (3, 5), (2, 2), (2, 1), (1, 2), (5, 5), (4, 6) \) of 6,
2-dimensional vectors.

Figure 1. K-d Tree Search
Since the K-d Tree search space grows in the number of dimensions, the probability of a generated hyperplane resulting in a balanced K-d Tree becomes increasingly small. In high dimensions, K-d Trees become chains, and have little advantage over linear search [10]. This result is often referred to as the Curse of Dimensionality (COD) for nearest neighbor search.

A more mathematical interpretation of the Curse of Dimensionality demonstrates our suggestion that approximate and exact algorithms can give equivalent qualitative results. Consider the distance between any two points \( x \) and \( y \) in \( \mathbb{R}^d \).

The general concept of LSH is similar to the intuition of K-d Tree search, suggesting that we can limit the number of candidate vectors in our dataset by only looking at those within some bounded distance of our query vector. The difference however in LSH, is that instead of traversing a tree, LSH selects partitions that are more likely contain the query vector and only search vectors within the selected region. The function that performs the region decoding is the locality sensitive hash function. Stated more formally we have:

**Definition II.2** (Locality Sensitive Hash Function). let \( \mathbb{H} = \{ h : S \rightarrow U \} \) is \( (r_1, r_2, p_1, p_2) \)-sensitive if for any \( u, v \in S \)

1) if \( d(u, v) \leq r_1 \) then \( P_{\mathbb{H}}[h(u) = h(v)] \geq p_1 \)
2) if \( d(u, v) > r_2 \) then \( P_{\mathbb{H}}[h(u) = h(v)] \leq p_2 \)

Though we choose a lattice based hash function from Andoni and Indyk [12], the original LSH search can utilize any function that that adhere to the LSH Function definition.

**Algorithm 2 Preprocessing**

**Require:** \( X = \{x_1, \ldots, x_m \}, x_k \in \mathbb{R}^n \)

1. \( U(x) \) is a standard hash function
2. \( h_k(x) \in \mathbb{H} \) is \( (r, cr, p_1, p_2) \)-sensitive hash function
3. choose \( l \) s.t. \( l = \log(n^\rho) \)
4. \( G \leftarrow \{i \in \mathbb{Z}^l \text{ from } [0, n)\} \)
5. \( g(x) = \{h_1(x), \ldots h_j(x)\} \)
6. \( D \leftarrow [] \)
7. for all \( x_k \in X \) do
8. \( D \leftarrow U(g(x_k)) \)
9. end for
10. return \( G, D \)

To increase the specificity of hash functions, multiple hash families can be concatenated into a single hash. Below we show the algorithms for the database construction (Algorithm 2) and the database query (Algorithm 3), in which we concatenate LSH hash families and then apply a standard digest hash \( U(\cdot) \) corresponding to our maximum database size [8].

This algorithm solves the c-approx Nearest Neighbors problem with log overhead [13] using \( O(dn + n^{1+\rho}) \)-space and \( O(n^\rho + \log(n)) \)-evaluations for the query [8].

**III. SERIAL IMPLEMENTATION**

We augmented the basic LSH algorithm with the lattice additions by Andoni [14] and the linear complexity database addition of Panagry [15]. We also apply a concept from Jegou [16] for query adaptive list pruning of the \( E_8 \) lattice, and apply it to the Leech Lattice decoder.

**A. \( \Lambda_24 \) Lattice Decoder As An LSH Function**

The complexity of LSH Nearest Neighbor search is given by Andoni and Indyk as \( \Theta(n^\rho) \) [17] where \( \rho \) is dependent on the selectivity of the locality sensitive hash function. Because the overall complexity is determined by the hash function, it stands to reason that the majority of our search algorithm should focus on optimizing the hash function. Intuitively we want a function that is able to divide the search space into even partitions, such as the partitions of the Voronoi diagram of a dataset. However, as the number of dimensions increase, the search complexity and generation of such diagrams becomes prohibitive. We consider a similar method, but instead of splitting regions separating points, we will consider hyper-spheres. This is defined formally as:

**Definition III.1** (Lattice LSH). given a lattice \( \Lambda_d \), generate all shifts \( v_i \in \mathbb{R}^d \) of \( \Lambda_d \), \( \Lambda_d + v_i \) such that the entire space \( \mathbb{R}^d \) is covered by a ball of radius \( r \) around the lattice points. Label the shifts by the vector \( v_i \). A hash function based on the uniform lattice \( \Lambda_n \) maps all \( x_n \rightarrow U \) by storing all shifts \( v_i \) with the integer referenced uniform lattice center in \( \Lambda_n \), i.e., \( x_1 = \{\{v_0, \Lambda_d(T)\}, \ldots, \{v_i, \Lambda_d(T)\}\} \).
The hash function is tasked with finding the nearest ball center to a given query point. For an arbitrary lattice in $\mathbb{R}^d$ the process of finding the sphere center requires checking all sphere centers and finding which is the closest to the query point and offers little advantage over linear search. A practical variant of sphere covering lattice LSH is considered in Andoni [14]. Well known lattices such as Gosset’s and the Leech Lattice not only have sub-linear nearest sphere center in Andoni [14]. Be’ery and Vardy’s [19], describe a decoding algorithm for the Leech Lattice that takes at most 519 floating point operations. This is quite remarkable as the lattice itself contains 196,560 vectors subjected to random projections in $\mathbb{R}^d$ ; $\Theta(\frac{\log(n)}{n})$. The density of the Leech Lattice is $\approx 0.00192957$, so despite the optimistic complexity on the number of dimensions needed to preserve relative distances between vectors, to assure all points map to their minimum distance lattice centers we still must store multiple random projections of the database vectors. The repeated projections increase the order of LSH Nearest Neighbor storage complexity, beyond the desirable linear bound maintained by linear and K-d Tree Search.

Panigrahy [15] suggests a time-space trade-off between storage and query complexity during the query stage that reduces the required storage complexity and maintains the same query time complexity. Panigrahy’s suggestion is to generate the random projections dynamically during the query stage. By choosing a set of random projections for a query vector to the Leech Lattice’s native space that are relatively near its location in the original space, the query vector is able to magnify the collision probability with the one set of stored database decodings. Thus both filling the holes, and adapting higher dimensional data to the Leech Lattice’s native space. The query algorithm now requires additional decodings, but the addition does not effect the overall complexity bound for the query algorithm [14].

The Leech Lattice is used as the locality sensitive hash function in this c-approximate k-nearest neighbors search. The sphere packing density of the Leech Lattice is optimal among regular lattices in $\mathbb{R}^{24}$ [18]. Be’ery and Vardy’s [19], describe a decoding algorithm for the Leech Lattice that takes at most 519 floating point operations. This is quite remarkable as the lattice itself contains 196,560 vectors in $\mathbb{R}^{24}$ in [III-A] we give a high level construction of the decomposition of the decoding process into various sub-code decodings.

**B. Random Projection**

To address the issue of the Leech lattice’s fixed dimensionality, Panigrahy [15]. and Andoni [14] propose using repeated random projections to map the data vectors to the Leech Lattice’s native space. The Johnson-Lindenstrauss lemma gives a tight bound for discretion of $n$ vectors subjected to random projections in $\mathbb{R}^d$ ; $\Theta(\frac{\log(n)}{n})$. The density of the Leech Lattice is $\approx 0.00192957$, so despite the optimistic complexity on the number of dimensions needed to preserve relative distances between vectors, to assure all points map to their minimum distance lattice centers we still must store multiple random projections of the database vectors. The repeated projections increase the order of LSH Nearest Neighbor storage complexity, beyond the desirable linear bound maintained by linear and K-d Tree Search.

Panigrahy [15] suggests a time-space trade-off between storage and query complexity during the query stage that reduces the required storage complexity and maintains the same query time complexity. Panigrahy’s suggestion is to generate the random projections dynamically during the query stage. By choosing a set of random projections for a query vector to the Leech Lattice’s native space that are relatively near its location in the original space, the query vector is able to magnify the collision probability with the one set of stored database decodings. Thus both filling the holes, and adapting higher dimensional data to the Leech Lattice’s native space. The query algorithm now requires additional decodings, but the addition does not effect the overall complexity bound for the query algorithm [14].

**C. Serial LSH with Parallel Projection and Decoding**

Below we give an overview of our parallel algorithm from the standpoint of the serial code distributing jobs to the available GPGPU processing units. The algorithm is shown in two parts, namely: (i) a first part (Algorithm 4) that builds the database, and (ii) the second part (Algorithm 5) that performs the nearest neighbor search of the database. This search is described more fully below.

In order to maintain the exact c,r query complexity...
bound, it is important the linear search list does not grow unbounded. In the original LSH Nearest Neighbor algorithm this was handled by truncating the list at 2L. A useful heuristic to improve the selectivity is suggested in Jegou LSH algorithm [16] which uses the E8 lattice to partition the search space, was to use the decoder center distance from the query to rank the returned list. Our decoder also takes advantage of this heuristic but for the Leech Lattice.

IV. PARALLEL IMPLEMENTATION

Our parallel implementation consists of a serial algorithm similar to the standard LSH preprocessing and query algorithms, with the LSH computation and random projection portions performed in parallel on the GPGPU.

A. Subspace Projection

In order to develop our parallel implementation we need to first consider the projection of vectors (when d > 24) to the subspace of our lattice in R24. Depending on the sparseness and length of our data vectors, we have a two options for when to perform the projection step. The first option is to transfer the data vector in its entirety to the GPGPU, and perform the projection within the main decoding step. Each thread corresponding to a vector computes its random projection prior to its decoding step and then returns the hash label result to the CPU. The benefit of the in-decoder projection is that it avoids a transfer between the GPGPU and CPU. By using a pseudo-random normal variate generator, that is seeded corresponding to the GPGPU’s thread id, the random projection matrix can be made consistent across all decoding threads. The advantage of this method assumes that we are dealing with dense vectors with relatively low (∼1000) dimensionality. If the vectors are sparse and high dimensional, we run the risk of not running as many decoding threads as possible.

The alternative is to compute all vector subspace projections in one bulk step. In the bulk projection method we are able to use already available, and efficient algorithms for sparse vector multiplication [20]. This method utilizes optimized asynchronous memory transfer in order to hide the data transfer latency of transferring to the GPGPU and back to the CPU. Furthermore, we are not limited by available memory size restrictions on the GPGPU. As with the in decoder projection method, we will generate the normal variates in the GPU using the thread ids such that subsequent threads will generate the same numbers for all the same dimensions. Although regenerating normal variates is redundant, the cost of transferring data from global memory is generally longer (∼1000 cycles, see Figure III) [21]. Because our proposed algorithm is for high dimension, we use the bulk projection method, though some problems may nonetheless benefit from in-decoder projections.

B. Parallel Lattice Decoding

The parallel decoder contains ≈ 519 floating point operations and has data requirements [2]. All operations are performed as fixed point operations to avoid the 8x overhead incurred on NVIDIA GPGPUs when using double datatypes. Furthermore this reduces the memory overhead of a single decoding thread.

1) Using Available Shared Memory: Due to the Nvidia GPGPU architecture having a restrictively partitioned memory access and processing structure, it is important to share data among small groups of threads(< 16 threads). In particular, shared memory can be accessed at speeds on par with register memory. This is in comparison to global memory, which comes at a penalty of around 100-200 cycles according to Nvidia’s Specifications (Figure III) [21].
2) Overcoming Shared Memory Limitations: The first attempt at a CUDA optimized implementation of the Leech Lattice Decoder simply tries to use shared memory instead of global memory. Intuitively the more we rely on faster memory, the faster the overall algorithm should operate. Following this idea, all global memory allocations were replaced with shared memory allocations. Immediately an issue arises as the maximum shared memory size is only 16kB per block. Given our data requirements, a single decoder needs approximately 778 bytes. The data footprint of an \( H\Lambda_{24} \) decoder is shown in Table II (the other half uses the same memory and follows sequentially).

Given the shared memory limit and the required data footprint of the decoder, a block can contain no more than 20 simultaneous lattice decoder threads. However according to NVIDIA’s documentation on CUDA [21] the number threads per block should exceed 32, allowing for two fully occupied half-warsps to be run. Thus leaving SPEs underutilized by the lattice decoder threads. To remedy this we decided to trade computation for memory occupancy. In this case it comes in the form of recomputing the \( d_{ijk} \) metrics from the QAM portion of the lattice decoder. By doing this along with reusing the k-parity array as the PrefReps array, and the \( d_{ijk}s \) array as the muE and muO arrays, we are able to compress the memory footprint down to 444 bytes. The memory requirements are shown in Table IV.

Using this reformatting with our new compressed memory footprint, we are able to theoretically run 36 threads per block. In general blocks should be multiples of 32, so we set our thread per block count to 32.

3) Partitioning the Leech Decoder over the \( Q\Lambda_{24} \) Subdecoders: Although the above reorganization allows us to meet NVIDIA’s minimum requirements, they suggest having far more threads per block in order to allow the CUDA thread scheduler to efficiently schedule threads to optimally occupy the GPGPU’s processing cores. We explore this possibility below.

At this point further optimization will not be achieved by compressing memory footprints any further, and instead a reorganization of our algorithm is needed. Fortunately the Bounded Distance Hexacode Based Decoder of Amrani et al [22] has a fairly straightforward method of parallelization. The partition simply breaks the decoder along the lines of the 4 \( Q\Lambda_{24} \) subdecoders. In addition to parallel computation, it may also be useful to utilize the shared nature of CUDA’s shared memory to our advantage. We do this by looking at the memory overlaps of the various subdecoders. The Venn Diagram IV-B3 shows where the 4 quad decoders overlap in their memory requirements, where each circle represents a \( Q\Lambda_{24} \) decoder. Using the \( Q\Lambda_{24} \) partitioning along with the conservative memory organization in the Figure IV-B3 we get a new outline for our decoder.

The below reorganization of the Lattice Decoder [4] results in the parallel execution of the 4 \( Q\Lambda_{24} \) lattice decoders of the full Leech Lattice decoder. This decoder requires 1344 bytes of shared memory (Figure V), but exists as 4 simultaneous threads. Using the same calculation as before to see how many simultaneous threads we can run, we get 16kB/1344B = 12 decoders. However, since each decoder consists of 4 threads, we can actually get 48 simultaneous threads per block, and our decoder algorithm is roughly 4x faster due to the improved occupancy.

4) CUDA Improvements: We will begin with the CUDA improvements to decoding the Leech Lattice vectors. We will not cover every improvement, as many are trivial, and follow a similar pattern to how other improvements were

<table>
<thead>
<tr>
<th>Memory</th>
<th>Per</th>
<th>Access</th>
<th>Latency</th>
<th>Caching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>thread</td>
<td>R/W</td>
<td>1</td>
<td>NA</td>
</tr>
<tr>
<td>Local</td>
<td>thread</td>
<td>R/W</td>
<td>1000</td>
<td>not cached</td>
</tr>
<tr>
<td>Shared</td>
<td>Block</td>
<td>R/W</td>
<td>2</td>
<td>NA</td>
</tr>
<tr>
<td>Global</td>
<td>Grid</td>
<td>R/W</td>
<td>1000</td>
<td>not cached</td>
</tr>
<tr>
<td>Constant</td>
<td>Grid</td>
<td>R/W</td>
<td>1000</td>
<td>cached</td>
</tr>
<tr>
<td>Texture</td>
<td>Grid</td>
<td>R/W</td>
<td>1000</td>
<td>cached</td>
</tr>
</tbody>
</table>

Table III
GPGPU Memory Organization

<table>
<thead>
<tr>
<th>Var</th>
<th>datatype</th>
<th>Size</th>
<th>Variable Alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_{ijk} )</td>
<td>float</td>
<td>( 12 \times 4 )</td>
<td>always needed</td>
</tr>
<tr>
<td>block</td>
<td>float</td>
<td>48</td>
<td>( d_{ijk} \rightarrow {QAM,kparity} )</td>
</tr>
<tr>
<td>( \muE, \muO \rightarrow )</td>
<td></td>
<td></td>
<td>{BlockConf, H6}</td>
</tr>
<tr>
<td>( bblock )</td>
<td>byte</td>
<td>48</td>
<td>kparities ( \rightarrow {QAM,kparity} )</td>
</tr>
<tr>
<td>( \text{prefRepE}, \text{prefRepO} \rightarrow )</td>
<td></td>
<td></td>
<td>{BlockConf, h-parity}</td>
</tr>
<tr>
<td>( yE )</td>
<td>float</td>
<td>6</td>
<td>always needed</td>
</tr>
<tr>
<td>( yO )</td>
<td>float</td>
<td>6</td>
<td>always needed</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td></td>
<td>444 bytes</td>
</tr>
</tbody>
</table>

Table IV
Compressed Data Structures Requirements for Lattice Decoding

![Figure 3. Better Occupancy By Overlapping Decoders](image-url)
The first important aspect of the CUDA architecture is that the SPE cores running in a half-warp are executed functionally in lock-step. This is a direct consequence of the cores being SIMD partitioned per half-warp. Due to this limitation, multiple branch path code will always operate the cores being SIMD partitioned per half-warp. Due to this functionally in lock-step. This is a direct consequence of that the SPE cores running in a half-warp are executed implemented.

The accuracy of our algorithm was compared on a subset of the 35K image database, as brute force searching the image database. This image database contains 35K images of the 35K image database, as brute force searching the database size of roughly 17.5M vectors. And on average each image generates 500 SIFT vectors, giving us a database size of roughly 17.5M vectors.

In the event of the conditional code, both conditions of the if statement must be traversed on average for the 8 sequential cores. This results in 2 accesses for the 3 distance arrays, which in the CUDA architecture results in a major slowdown. In the concurrent code, the conditional array accesses occur simultaneously, which allows the scheduler to swap in an non-blocking IO execution block, while the current block awaits data. Furthermore, the additional computation is negligible, as the cores allow for some pipelining of arithmetic operations, that would otherwise not be utilized. Unfortunately the overall execution paths are difficult to measure or visualize as the CUDA framework is somewhat hidden from view. Instead results can only be obtained from full system simulation with random point data. This data is then used to measure basic speedup. In the case of this alteration for most conditional branches in the code, the result to determine the memory locations (Figure VI). The way that conditionals are dealt with throughout the the code, is by pre-computing the results of conditional operations and allowing the result to determine the memory locations (Figure VI).

In the event of the conditional code, both conditions of the if statement must be traversed on average for the 8 sequential cores. This results in 2 accesses for the 3 distance arrays, which in the CUDA architecture results in a major slowdown. In the concurrent code, the conditional array accesses occur simultaneously, which allows the scheduler to swap in an non-blocking IO execution block, while the current block awaits data. Furthermore, the additional computation is negligible, as the cores allow for some pipelining of arithmetic operations, that would otherwise not be utilized. Unfortunately the overall execution paths are difficult to measure or visualize as the CUDA framework is somewhat hidden from view. Instead results can only be obtained from full system simulation with random point data. This data is then used to measure basic speedup. In the case of this alteration for most conditional branches in the code, the results were significant (~3:5 speed increase).

V. EXPERIMENTAL RESULTS

A. Application

For testing our nearest neighbor algorithms we wanted to focus on dense, medium-high dimension vectors, against a very large database. For this we will be using the Scale Invariant Feature Transform of [23] on the Caltech256 [24] image database. This image database contains 35K images and on average each image generates 500 SIFT vectors, giving us a database size of roughly 17.5M vectors.

B. Accuracy

The accuracy of our algorithm was compared on a subset of the 35K image database, as brute force searching the
database was computationally infeasible. The highly tuned
best-bin search method of Lowe incurs a recall accuracy loss
of $\approx 5\%$ over K-d Tree search [23]. Our algorithm performs
slightly less accurately and has recall accuracy $\approx 8.8\%$ less
than linear search. However, this is over the vanilla LSH
algorithm, with no specific tuning for SIFT (Figure VII).

**Table VII**

| L=1 | 0.7848605577689243 |
| L=10 | 0.8725099601593626 |
| L=10 | 0.900398406374502 |
| Linear Search | 0.9881889763779528 |

**C. LSH-NN and Exhaustive Search of SIFT Vectors**

In this section we will use our image search application
utilizing LSH-NN to search for near neighbor vectors of im-
ages generated by the scale invariant feature transform [23].
Our goal in this paper is not to show any gain or superiority
over other search methods, but instead to simply establish an
equivalence to other similar methods. Our main result was
generated from a random selection of 294 images from the
Caltech 256 Image Database [24]. The reason for selecting
such a small subset of images was the nature of the linear
search technique employed for comparison to our algorithm.
We will present 3 metrics here in regards to the performance
of our LSH algorithm. As stated before, the LSH portion
resides mainly in GPGPU processing while the LSH-NN
framework is implemented in python. For this reason the
times are considerably slower than an optimized LSH-NN
would yield, however the overall complexity as a function
of $n$ should be consistent with our expectations of the LSH-
NN algorithm of Andoni [14]. After establishing equivalence
between our algorithm and previous similar LSH algorithms,
we will show a comparison by way of Selectivity Recall
comparing our new query-adaptive Leech Lattice search
algorithm, and that of the previous unsorted algorithm. Due
to the heuristic nature of the nearest lattice point ranking, the
overall complexity bound will remain unchanged. However
the adaptive algorithm shows significant advancement in real
world applications for exact c-approximate nearest neighbor
search.

1) **Comparison of Time for Searching:** Figures 5 and 6
shows a random subset of times to perform exact linear
searches as well as times to perform LSH searches for a
given size database. A few issues prevent us from showing
the true advantages of LSH searching. In particular
our experimental analysis is strongly limited due to linear
searching quickly becoming intractable for large databases.
Initially a subset was sampled from the database and used
to generate an approximate average time. However, even for
relatively small image databases ($n > 1000$ images, $\approx 300$
SIFT vectors each), results from linear searching began to
be affected by memory thrashing. Furthermore, the variance,
as depicted by the error bars for linear search, increases as
the database size increases. This is due to the images with
$>> 300$ SIFT vectors having a higher probability of being in
the search sample. As search complexity increases in regards
to $n$, the average search time exhibits an overall increase in
variance.

To avoid these issues, and still provide a reasonable
comparison for more realistically sized datasets, we will
simply refer to the complexity bounds as defined in Andoni’s
dissertation [17]. We will extend the exact times from data
in Figure 6 and use this formula along with the linear search
complexity, to extrapolate speedup. Figure 7 shows the
previous chart extrapolated using the corresponding big-oh
functions for LSH and Linear search, in a semi-logarithmic
chart, for an $n$ size search database. The functions used to
fit the previous data are $\Theta(n^\rho)$ and $\Theta(n)$, where $\rho = .3671$ and least squares constant values are used to establish a curve fit via the somewhat overkill, but standard Levenberg-Marquardt variable optimization algorithm. Figure 5 and Figure 6 show the fit quality of the corresponding equations. An important note, is that the LSH algorithm tends to jump, instead of grow gradually with $n$. This is due to the random projection matrix generation having a discrete number of dimension, which has direct implications on the search time. The variable is still bounded above by $\Theta(n^\rho)$ according to Panigrahy [15] as $n$ grows large.

The LSH search time experimental results are fit to $\Theta(dn^\rho)$ where $\rho$ is attained experimentally from the $\log(p_1)/\log(p_2)$ hash intersection probabilities, as given in the search complexity bound in Andoni [17] for LSH Search. The Linear search time results are fit to $\Theta(nd)$. The variability in the search comes from the variation in the number of sift vectors in our real world image set. The Linear search is slightly more consistent, while the LSH search, projecting to a randomly generated sub-space, causes more greater variability, but is consistent over a large database.

D. LSH-NN Speedup

In this section we will draw conclusions on the theoretical speedup of our parallel LSH-NN algorithm using Amdahl’s Law for parallel speedup. Although we get good speedup from the GPGPU LSH function, the algorithm will ultimately be constrained by Amdahl’s Law in regards to sequential to parallel computing time ratios. This constraint is rather bleak, as even with very efficient parallel implementations of the LSH-NN algorithm, we will always be bounded by some portion of sequential code. The equation for Ahmdal’s law regarding theoretical speedup as a function of the achievable parallel speedup ($S$), with a constant ratio of sequential to parallel code $P$, is stated below.

$$\frac{1}{(1 - P) + \frac{P}{S}}$$

E. Parallel vs Sequential Computation Time

Our calculated ratio of sequential to parallel time for our parallel LSH-NN algorithm run on a sample of random SIFT vectors can be seen in Figure 8. We believe this ratio is a lower bound for the speedup attainable by our presented algorithm. In Figure 9 we show calculated curves of our parallel to sequential ratio as well as other speedup ratios that we believe are attainable under Amdahl’s Law. The light green rectangular area depicts the feasible speedup region of our parallel Leech decoder on a single GPGPU. An ideal goal of parallelizing LSH-NN would be to have a speedup ratio of 1, however, as this is unlikely, we will settle for a speedup curve that has good performance...
in the region corresponding to the number of SPEs on a typical GPU. From the theoretical curves, 97% parallel results in a desirable speedup region for common GPU hardware. A solid black horizontal line shows the predicted parallel speedup using our speedup ratio and the parallel lattice decoding speedup (41x) attained by our decoder. We conclude that 11x is the speedup attained by our presented algorithm.

VI. CONCLUSION

In this paper we demonstrate an algorithm for c-approximate r-nearest neighbors search using GPGPUs that attains an approximately 11x speedup over the sequential algorithm. Using Nvidia’s CUDA framework, we were able to accelerate the approximate nearest neighbors search while maintain the storage and computational complexity of the original LSH nearest neighbor search algorithm. We also show that under Amdahl’s Law speedup estimates, our algorithm performs well for the number of processing units commonly found on modern GPUs.

REFERENCES


