Space-efficient Pointwise Computation of the Distance Transform on GPUs

Numair Khan
Computer Science Department,
Brown University
Providence, Rhode Island.
Email: numair_khan@brown.edu

Mohamed Zahran
Computer Science Department,
New York University
New York, New York.
Email: mzahran@cs.nyu.edu

***Abstract*—To minimize the amount of computation, traditional approaches to calculating the distance transform (DT) on a discrete volume propagate distance values in a local neighborhood. This results in recursive dependencies across the volume, requiring the DT to be calculated for all points in the domain en mass and stored as static values in memory. On the other hand, the ability to calculate the distance transform point-wise not only offers the prospect of efficient memory usage and scalability, but also a high degree of flexibility in accommodating the unique requirements of new application domains. However, among the current DT algorithms, the computationally intensive brute-force algorithm is the only one that allows point-wise computation. We demonstrate that the by decomposing it into a map and a reduction pattern on the massively parallel architecture of a modern Graphics Processing Unit (GPU), the brute-force distance transform algorithm achieves the threefold goals of memory efficiency, flexibility, and performance. We discuss a memory constrained implementation in the CUDA parallel programming model. The flexibility of point-wise computation at runtime is demonstrated by presenting an approximate and an anisotropic variant of the standard distance transform algorithm, and using these variants for the rendering of a CT scan image. Our approach allows the distance transform to be calculated for 1024 query points and up to 16 million feature points in 141.25 milliseconds while allowing direct control over the memory working-set size. These results demonstrate the potential of point-wise computation of the DT at runtime and the need for future algorithms to incorporate this capability.**

***Keywords*-Volume; Rendering; Medical Imaging; Vision; Map; Reduction; Memory; Patterns; Anisotropic; Approximate**

I. Introduction

If d is the distance function on a metric space M, and S a subset of M, then the distance transform is a function $T : M \rightarrow \mathbb{R}$ defined as

$$T(q) = \{d(q,s) \mid d(q,s) \leq d(q,s') \; \forall s' \in S\}$$

Hence, for each $q \in M$, T provides the distance to the nearest element of S. Most often, M is the n-dimensional Euclidean space and d is the Minkowski distance metric $L_p = (\sum_{i=1}^{n} |s_i - q_i|^p)^{1/p}$ between points $s = (s_1, s_2, ..., s_n)$ and $q = (q_1, q_2, ..., q_n)$. L_1, L_2 and L_∞ give the commonly used Manhattan, Euclidean and Chebyshev (chessboard) distances, respectively.

The distance transform, and the closely related operation of Voronoi partitioning, find application in a variety of subject areas, ranging from computer vision, data analysis and computational geometry to artificial intelligence and robotics. In computer graphics, the distance transform is used to speed up the raytracing of volumetric models [1] and Voronoi diagrams form the basis of many nature-like procedural textures [2]. For a more comprehensive list of applications see [3] and [4].

The brute-force approach to calculating the distance transform (DT) for a set of query points $Q \subseteq M$ compares each $q \in Q$ to every single $s \in S$ and runs in $O(|Q||S|)$ time. Current serial and parallel algorithms seek to improve on this by performing a local search in a neighborhood $\mathcal{N}$ around q and propagating known distance values through the domain M [3], [4]. Apart from the fact that the Euclidean distance on a discrete lattice is not a local property and evaluating it as such yields inaccurate results [5], implementations of said algorithms on modern Graphics Processing Units (GPUs) suffer on two additional fronts. Firstly, propagation of distances to neighbors imposes an order on the sequence of operations, requiring at least part of the computation to be serialized and limiting the amount of parallelism that can be exploited. This impairs the scalability of the algorithm to processors with a very large number of cores. Secondly, since T cannot be calculated for a single point q without first calculating $T(q') \; \forall \; q' \in \mathcal{N}$, the algorithms cannot operate at a granularity finer than the whole domain. Hence, in the interest of efficiency, the distance values for all $q \in M$ are calculated en masse and stored in memory, even though a large majority may never be queried ($|Q| \ll |M|$). If $M \subset \mathbb{Z}^3$, and assuming single precision floating point numbers, storing the DT for a $512 \times 512 \times 512$ volume requires an impractical 512 MB of limited global GPU memory. For a $1024 \times 1024 \times 1024$ volume, the requirement goes up to four gigabytes.

While an asymptotic running time of $O(|Q||S|)$ makes the brute-force approach ill-suited to a serial implementation, the algorithm does offer three advantages over existing distance transform algorithms:

1) It offers a very significant degree of task and sub-

This research was conducted while the first author was a graduate student at New York University

task level parallelism: not only can $T(q)$ be calculated independently for each query point q, but the subtask of finding $d(q, s) \forall s \in S$ is also highly parallel in nature. According to Amdahl's Law, this implies very strong parallel scalability.
2) Calculated results are guaranteed to be exact. This is not necessarily true for algorithms which treat the Euclidean distance metric as a local property.
3) Since $T(q)$ is no longer dependent on a neighborhood $\mathcal{N}$ around q, the DT can be calculated at the granularity of a single query point.

The parallelism inherent to the brute-force algorithm offers great potential for exploitation by the large number of processing cores of a modern GPU. Moreover, if sufficient speedup is achieved, the calculation of $T(q)$ can be deferred until runtime, obviating the storage of any precomputed data. Runtime computation also offers flexibility in tuning the distance transform to the specific requirements of each application area: the size of S may shrink or expand to accommodate dynamic systems; time-critical applications may choose to consider only a subset of S depending on operational constraints.

Therefore, the objective of this paper is to demonstrate the viability and utility of calculating the distance transform pointwise at runtime using the brute force algorithm on a modern GPU.

Henceforth, the terms *sites* and *feature points* are used interchangeably to refer to elements of the set S, whereas the elements of Q are called *query points*. While the following discussion assumes both S and Q to be subsets of the three-dimensional Euclidean space, the findings may be extended to any metric-space M.

Original Contribution: The novel contribution of our work is as follows:

1) The brute-force algorithm is decomposed into a map and a reduction pattern to enable space-efficient pointwise computation of the distance transform at run-time on a GPU. An implementation of the algorithm in the CUDA parallel programming model is presented.
2) An approximate variant of the algorithm that allows accuracy of results to be traded for speed of computation is presented.
3) The flexibility provided by pointwise computation is demonstrated by extending the original kernel to calculate an anisotropic distance transform.

Organization: Section II presents a brief survey of the existing work on distance transforms. The CUDA implementation of our algorithm, along with its approximate and anisotropic variants, is discussed in Section III. Sections V and VI respectively describe the setup and the results of our experiments as well as the application of the abovementioned variants. A concluding section sums up our findings.

II. Related Work

The work of Rosenfield and Pfaltz [6] is among the earliest to propose the restriction of the global distance minimization operation to a local neighborhood. Their algorithm is the first example of a *raster-scanning* DT algorithm and uses a Von Neumann neighborhood to calculate the Manhattan distance for each pixel in two passes over a digital image. Other neighborhoods may be used and each neighbor's contribution weighed to better approximate the Euclidean distance metric [7]–[9]. The choice and weight of neighbors is commonly represented as a mask centered on the pixel under consideration. Since each pass propagates only an approximation of the Euclidean metric, errors tend to accumulate with distance and over several passes. In addition, the value of each pixel may be updated more than once [3] requiring multiple reads and writes. The former shortcoming is remedied by vector propagation schemes [10], [11] which propagate a vector to the nearest site rather than the distance value. However, the increased accuracy comes at the cost of a larger memory footprint as a single distance value is replaced by the n coordinates of an n-dimensional vector. Moreover, the dependency between consecutive passes, present in all raster-scanning algorithms, limits parallelism. Techniques that scan rows and columns independently seek to overcome this deficiency. Ragnemalm [11] presents a modified, three dimensional version of the original vector masks of Danielsson that allows the x, y and z axes to be scanned in parallel during each pass, as do Schneider et al. [12] who also provide a GPU implementation. The algorithm of Cao et al. [13] uses the geometric properties of Voronoi sites to combine the results of scans along a single axis on a GPU. It should be noted that operations along the columns of matrices (images) stored in row-major order leads to a large number of uncoalesced memory accesses, thereby, significantly degrading performance.

Ordered propagation distance transform algorithms transmit distances along a wavefront originating at the feature elements [5]. These algorithms are similar in nature to Djikstra's shortest path algorithm for graphs and the Fast Marching Method for solving the Eikonal equation [14], [15]. Accordingly, Lotufo et al. [16] treat the calculation of the distance transform as a shortest path forest problem and present two algorithms based on Djikstra's. Jones et al. [4] provide a survey of Fast Marching Methods applied to the distance transform problem. Because the size of the wavefront along which distances are calculated is usually small compared to the size of the entire domain, ordered propagation techniques have good best-case running times on serial machines. In addition, Piper and Granum [17] show that propagation can be used to calculate the distance transform in non-convex domains. However, the ordered nature of the algorithms and the concentration of computation on a narrow wavefront limits the potential for parallelism on manycore systems. Moreover, in a discrete grid, an irregular contour shape may cause threads in the same warp to access memory in an uncoalesced manner. Rong and Tan [18] address the former limitation by propagating values to points at logarithmically varying distances rather than immediate neighbors.

Due to the propagatory and data dependent nature of both categories of algorithms discussed above, the distance value

for a solitary point cannot be determined without first calculating the transform for the entire domain. As noted earlier, this shortcoming is addressed by adapting the memory-intensive approach of calculating the DT for all points en masse and storing it in memory. While hierarchical data structures may provide improvements over regular grids by grouping large homogeneous regions [19], [20], their memory requirements are dependent on the contents of the domain and degenerate to the worst case when a large number of high frequency features are present. Since the brute-force algorithm referred to in the introduction allows pointwise independent calculation of distance values, it is useful to survey other problems with a similar construction and analyze the data structures and approximations used to overcome the high asymptotic running time.

If the feature points are considered as entries in a database, finding the Voronoi site for an element q in the domain is a *nearest neighbor problem* [21] and the distance between q and its Voronoi neighbor is easily calculated to yield $T(q)$. In d-dimensional domains, the $O(dn)$ search time of the nearest neighbor, where n is the size of the database, is commonly accelerated using either spatial data structures [22]–[24] or, in high dimensional spaces, Locality Sensitive Hashing [25]. However, the work of Garcia et al. [26] supports our hypothesis that the brute-force implementation of the nearest neighbor search on a GPU is many times faster than a CPU algorithm that uses kd-trees. In turn, a GPU kd-tree algorithm to handle a very large number of queries is proposed by Gieseke et al. [27]. Their algorithm seeks to amortize the cost of communication by waiting for enough spatially proximate queries to accumulate in a buffer to allow memory access in a coalesced manner. While it outperforms a brute-force search when the number of queries is very large, the delay in waiting for the buffer to fill and the conditional branches involved in tree traversal make it ill-suited for cases where a small number of queries are to be served in a time-constrained manner. The computation of approximate nearest neighbors has been studied by many authors [28], [29], but their work primarily deals with higher dimensions where the performance of spatial data structures degrades significantly.

The $O(n^2)$ pairwise interactions between elements of a set, where n is the size of the set, makes the distance transform an *n-body simulation* problem. As in the nearest neighbor problem, a hierarchical subdivision of space with a tree is used to accelerate the calculation [30]. While techniques for approximating the contribution of distant tree nodes, such as the Fast Multipole Method [31], are commonly used, they are not relevant to our study of the distance transform which, by definition, only considers the nearest contributing element. The parallelization of the brute-force n-body simulation algorithm on GPUs is studied by Nyland et al. [32].

III. CUDA Implementation

GPUs are massively parallel computation devices, designed to provide high throughput by running many lightweight threads in parallel on a large number of low-frequency

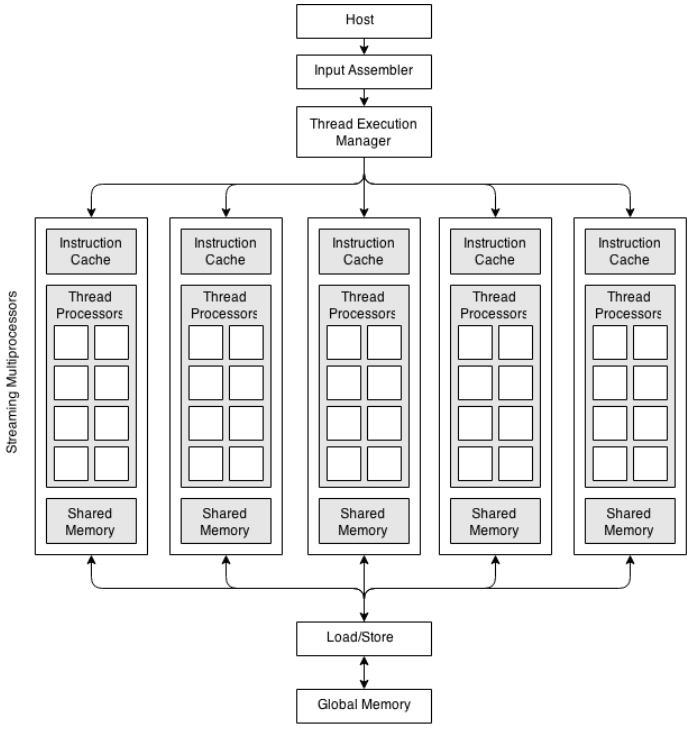


Fig. 1: CUDA GPU architecture

streaming multiprocessors (SMs). NVIDIA's CUDA parallel programming model [33] groups threads into *blocks* which act as units of organization and work distribution. Threads in a block execute on the same streaming multiprocessor and may synchronize via barriers and communicate through fast, per-block shared memory. The shared memory is interleaved across 16 or 32 memory banks depending on the implementation. A *warp* refers to a subgroup of threads within a block that runs in SIMD (Single Instruction, Multiple Data) manner on the execution units of an SM. While the warp size is specific to each implementation, the block size and the total number of blocks is set by the user and usually determined by the amount of data-parallelism on offer and the memory access patterns of the CUDA kernel.

Calculating the brute force distance transform for a single query point q is an example of an *pleasingly parallel* task that may be decomposed into a map and reduction pattern [34]. A map applies a function to each element of an input collection to generate an output collection whereas a reduction combines all inputs into a single output by applying an associative binary operator. In the present case, the mapping function f calculates the Euclidean distance between a single feature point and q whereas reduction is done by applying the binary minimum operator to the collection output by map (Figure 2).

A. Map

The serial construct for evaluating a map pattern is a for loop over all input elements and while the individual iterations, being independent, are easily parallelizable, invoking a separate thread for each iteration is not work-efficient if the

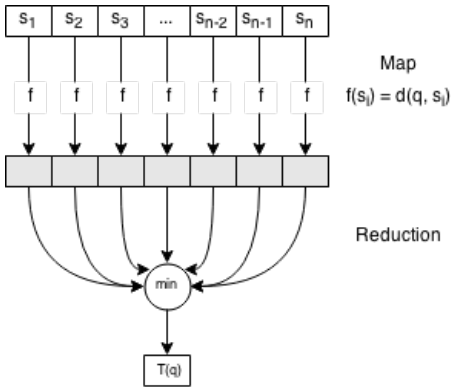


Fig. 2: The map pattern applies the function $f(s_i) = d(q, s_i)$ to each feature element $s_i \in S$. The reduction pattern applies the minimum operator to all elements in the intermediate list output by map to find $T(q)$.

mapping function does only a small amount of computation. Instead, a common strategy is to divide the input collection into *tiles* and invoke one thread per tile [32], [34], [35]:

$tileId \leftarrow threadId$
$i \leftarrow 0$
for $i < tileSize$ **do**
  $idx \leftarrow tileId * tileSize + i$
  $output[idx] \leftarrow f(input[idx])$
end for

Knowing that a map is followed by a reduction, the two patterns may be combined in the implementation of a single thread to yield the results of a map and a partial, tile-wide reduction:

$tileId \leftarrow threadId$
$D \leftarrow maximum\ datatype\ value$
$i \leftarrow 0$
for $i < tileSize$ **do**
  $idx \leftarrow tileId * tileSize + i$
  $D \leftarrow min(f(input[idx]), D)$
end for
$result[tileId] \leftarrow D$

Since in most practical case $|Q|$ would be a small number larger than one, multiple threads may operate on one tile to calculate results for multiple query points $q_i \in Q$ in parallel. With this consideration in mind, our CUDA kernel is structured so that a one-to-one ratio exists between tiles and thread blocks. Then, the ith threads of all blocks collectively implement a map and a partial reduction for query point q_i. If the number of query points is greater than the number of threads per block n, then the same group of ith threads is also responsible for $q_{i+n}, q_{i+2n}, q_{i+3n}$, and so on. By imposing a one-to-one mapping between tiles and thread blocks, we are assuming that the number of tiles will always be less than, or equal to, the maximum number of thread blocks supported by the GPU. This is not an unreasonable assumption for most applications: CUDA devices with compute capability greater than 2.0 can have 65535^3 blocks per grid, and 1024 threads per block, which allows more than 2×10^{17} feature points.

An alternative arrangement, adopted by Nyland et al. [32] in their n-body force calculation kernel, is to divide the collection of query points rather than the collection of feature points into tiles. The benefit of this approach is that the entire distance transform computation for a single point is serialized in one thread and no separate reduction kernel is needed. However, as we expect the number of query points to be much smaller than the number of feature points, a division along the former limits the maximum number of tiles that may be created which in turn limits the scalability of our algorithm.

As in Nyland et al. [32], effective data reuse is achieved by synchronizing a block of threads to load a tile from global memory into the high-speed shared memory before proceeding with map. By storing the list of feature points as a *structure of arrays* in global memory with the x-components of all points followed by the y-components followed by the z-components, we ensure threads in a warp access the individual components in a coalesced manner. The tile size is chosen so that each coalesced access is aligned to a 128-byte boundary. Finally, when a tile of feature points has been loaded into shared memory each thread proceeds with calculating $d(q_i, s)$ for each s in the loaded tile. Conflicts on shared memory banks are avoided by having all threads in a warp simultaneously access the same 32-bit component of the feature point s, thereby, allowing the CUDA implementation to invoke an efficient inbuilt broadcast mechanism. One limitation of our approach is that if the number of query points is smaller than the warp size, a number of threads in the block will be idle after the initial load into shared memory. While this problem can not be completely avoided, its effects can be mitigated by choosing a tile size equal to the warp size.

B. Reduction

The design of the reduction pattern is relatively more complicated due to the distributed manner in which a single input point is processed and the lack of any efficient synchronization mechanism across blocks. Launching a separate reduction kernel following the map operation offers the greatest parallelism, but the large memory requirements of the intermediate data that would be generated by map violates the premise of our work: a $1,342,178$ element collection of feature points (a number that results from a 512^3 discrete volume having a modest site density of 1%) divided into tiles of 128 elements each, requires 40MB to store the raw output of map for a small input of only 1024 query points. More specifically, if m is the number of query points and N the number of tiles, the intermediate data requires $O(Nm)$ space. As both N and m are expected to be relatively larger numbers in real volume data sets, it is imperative that the space complexity of the reduction pattern be optimized.

One solution is to use the `atomicMin()` function offered by CUDA implementations with compute capability 1.1 and higher to commit or discard the output of a thread as soon as it is generated reducing the memory requirement to $O(m)$. The disadvantage of using atomic operations, however, is that by

locking the data under consideration they effectively serialize all accessing threads.

A more refined solution is to output map data in small chunks that are immediately consumed by a reduction kernel, never allowing the storage requirements to exceed a user-specified limit. This is achieved by decomposing the monolithic map and reduction operation into K kernels each, and interleaving their execution across J CUDA streams (Figure 3). After all J streams have finished executing, one final reduction combines their individual results. The values of J and K together determine the space requirements of the operation. Theoretically, since each kernel now operates on N/K tiles, a value of $J = 2$ suffices to reduce the intermediate storage requirement to $O(J\frac{N}{K}m) = O(\frac{N}{K}m)$ while allowing all kernels to execute in the same total time as the monolithic version. In practice, however, performance may not scale linearly with data size and larger values of J may be required to achieve the target execution time at the cost of storage space (Figure 4). In the extreme case when $J = K$ and all K map kernels run in parallel, the storage requirement of the intermediate data becomes $O(J\frac{N}{K}m) = O(mN)$ again.

The reduction kernel is designed so that a thread block operates along one column of the intermediate data to produce results for a single point. Since the data is stored in row-major format to allow its decomposition among the K map kernels, this direction of reduction necessarily involves uncoalesced global memory access. The resulting performance penalty is minimized by loading tiles of data into shared memory.

IV. Distance Transform Variants

The flexibility afforded by pointwise calculation at run-time allows the distance transform to be adapted to the unique requirements of each application. This is demonstrated by two variants of the standard algorithm.

A. Approximate Distance Transform

Many application domains are tolerant of a degree of error in results allowing accuracy to be traded for speed of computation. As profiling results indicate reduction accounts for a only a small percentage of the total running time of our DT calculation, we chose to concentrate our approximation efforts on the map kernel.

Samadi et al. [36] approximate map by using a pre-defined value from a lookup table for the mapping function f. However, the use of a necessarily large lookup table harkens back to the method of storing the distance values for the entire volume in memory and violates the memory-efficient philosophy of our work. Moreover, the simplicity of the distance mapping function does not warrant the relatively long latency of a memory access. The pointwise calculation of the DT allows us to use an alternative approximation technique: *perforation*. Perforation provides performance gains by restricting a loop to a subset of critical iterations [37], [38]. In the present case, this implies performing distance calculations for only a subset of feature points, effectively reducing the input size. While it is not possible to provide a bound on the maximum absolute error, a probabilistic analysis of loop perforation by Misailovic et al. [39] confirms that, under certain assumptions, the mean absolute percentage error remains low for computational patterns very similar to map.

It should be noted that perforation is not the same as filtering the input using a spatial data structure such as a kd-tree, which optimization always provides an accurate result, albeit, at the cost of greater processing requirements. Even in situations where a spatial data structure is used, loop perforation may improve performance by approximating results for feature points falling in the same leaf node.

B. Anisotropic Distance Transform

We define an anisotropic distance transform as one which only considers feature points lying in a certain direction from an input query point. The pointwise algorithm is conveniently modified to calculate the anisotropic DT by incorporating an additional dot product calculation in the map kernel to discard all feature points lying outside a particular angle range.

TABLE I: Relevant properties of the GPUs used for our experiments

	GTX TITAN Black	GTX 750 Ti
Multiprocessor count	15	5
Cores/multiprocessor	192	128
Clock rate	0.98 GHz	1.267 GHz
Shared memory size	48 KB	48 KB
Global memory size	5 GB	1 GB
Global memory bandwidth	313 GB/s	80 GB/s

V. Experimental Setup

Our CUDA kernels were executed on a multi-GPU system with an NVIDIA GeForce GTX TITAN Black and a GeForce GTX 750 Ti. The CUDA properties of each device are presented in Table I. For kernels executing jointly on the two devices, the workload was not split evenly as the TITAN Black provided more than twice the performance as the GTX 750. Instead, a ratio of 4:1 was determined analytically from the observed speedup differential. The baseline for our speedup calculations was a sequential C implementation of the brute-force distance transform run on a 16-core, 2.1 GHz AMD Opteron 6272 with 256GB of memory, a 16KB and 64KB L1 data and instruction cache respectively, a 2048KB L2 cache and a 6144KB L3 cache.

A CT scan of the Stanford Bunny [40] provided the volume data for the anisotropic and approximate DT kernels. The $512 \times 360 \times 512$ dataset stores the electron density of the subject at discrete locations as Hounsfield units in the range $0 - 4096$. A threshold value of 1843 (0.45 on the grey scale) was used to filter out the solid portion of the volume and yield the feature points for our distance transform calculations.

When calculating the running time of each kernel, the memory transfer latency was included only for the list of query points since the feature points, assumed stationary, remain resident in the GPU memory after one initial copy. The final running-time is the average of three runs of a kernel, obtained

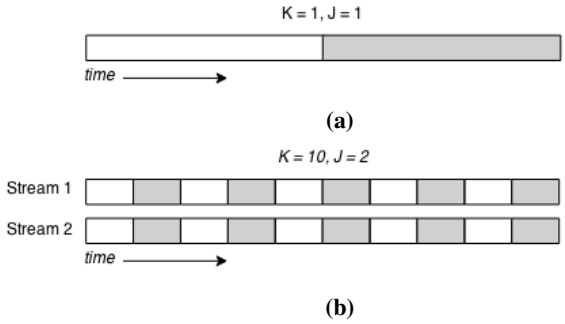


Fig. 3: (a) represents the execution of monolithic map and reduction kernels. (b) shows how the memory requirement is reduced by dividing both patterns into $K = 10$ smaller kernels. Each kernel now operates on $N/10$ tiles reducing the storage requirement by $1/5$th. This figure assumes a proportional reduction in kernel execution time with data size.

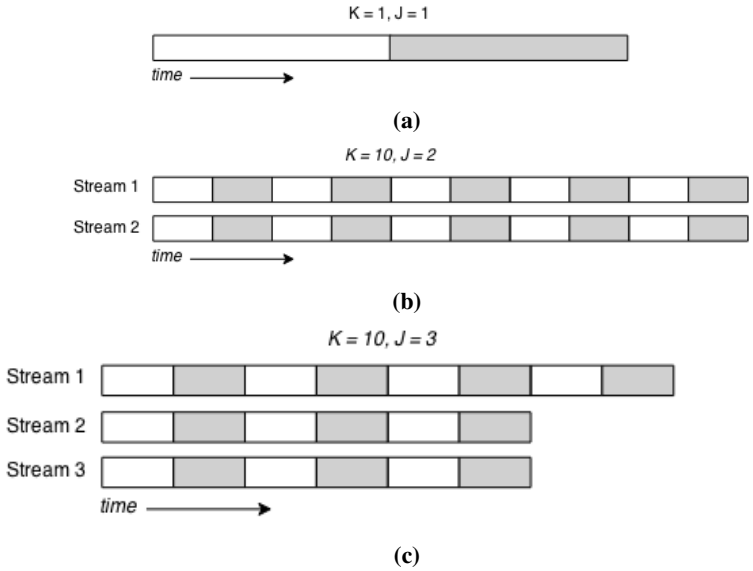


Fig. 4: (a) represents the execution of monolithic map and reduction kernels. (b) depicts a situation in which the execution time of a kernel does not reduce proportionally with data size. (c) shows how adding an extra stream ensures timely execution but increases the memory requirement to $3/10$th of the original.

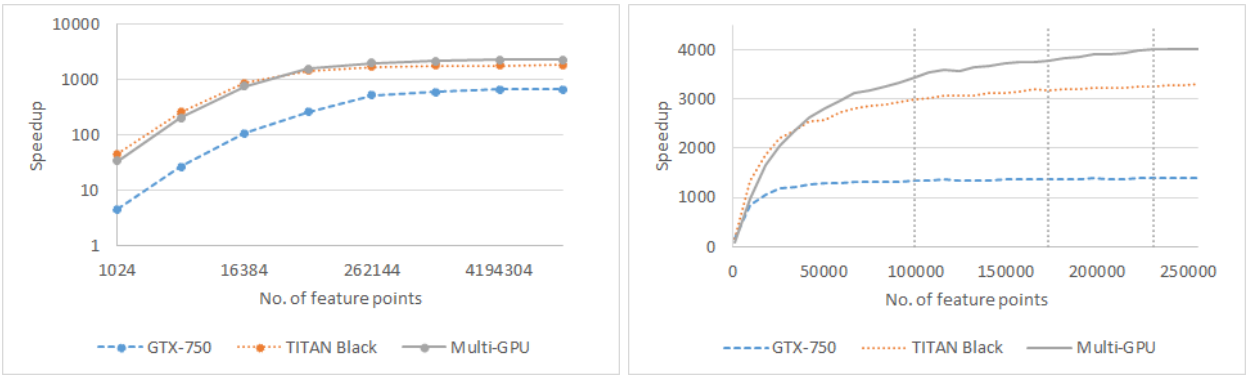
after discarding the highest and lowest value from a sample of five runs.

By restricting all points to $\mathbb{Z}^3$ and using the square of the Euclidean distance for comparisons, we were able to avoid floating point calculations in all but the anisotropic DT kernel, which requires normalization of vectors for angle calculation. The requirement that all points be in $\mathbb{Z}^3$ is not unreasonable as many existing DT algorithms [5], [6], [8], [10], [12], [13], [18] operate on a two or three-dimensional raster with integer positions. A 32-bit unsigned integer can represent the maximum distance between two points in a volume as large as 32768^3 and, hence, we need not be concerned about bit overflow during our computations on much smaller volumes.

VI. Experimental Results

Figure 5a shows the speedup over the serial version achieved by the distance transform kernel on three different hardware configurations: the GTX 750 and the GTX TITAN as stand-alone devices, and a combination of the two in a multi-GPU system. With three initial exceptions, it is observed that for a given number of feature points, the speedup increases as more processing cores become available. The exceptional cases, where the GTX TITAN outperforms the multi-GPU, may be accounted for by the unamortized overhead of executing a kernel on multiple devices. For each configuration, the speedup increases linearly with data size before settling on a constant value as the parallelism on offer is gradually saturated by an increasing data size. The point of inflection on each curve represents the data size at which saturation begins and, as expected, it is further to the right for the configuration with greater cores (Figure 5b) suggesting the data-parallel nature of the algorithm.

The space benefits of the algorithm are shown in Figure 6b. Whereas the memory requirement of a single kernel grows linearly with the number of feature points (N), having K smaller kernels launched in two CUDA streams, and assuming



(a) **(b)**

Fig. 5: The (a) speedup for three different GPU systems, and (b) a close-up of the speedup on a linear scale with the approximated inflection point for each curve marked by a vertical line. The inflection point is further to the right for a curve representing a device configuration with a higher number of processing cores. The number of query points is fixed at 1024.

the number of query points is fixed, implies the memory required at any given instant is $\Omega(N/K)$. Hence, increasing K in proportion to N allows the size of the working set to be kept constant.

While the primary purpose of using multiple small kernels in place of monolithic map and reduction kernels is to reduce the memory footprint of the distance transform calculation, it was observed that the former approach yielded significant gains in performance as well (Figure 6a). A profile of the CUDA code indicates the primary source of the gain as the reduction operation. This is explained by the tree-based structure of parallel reduction [41] in which a portion of threads in each block remain idle after an initial computation. By reducing the block size and, hence, the number of idle threads, the smaller reduction kernels utilize the resources of the GPU more efficiently than a single, large kernel. The resulting speedup more than offsets the overhead of multiple launch calls. It should be noted that performing reduction implicitly through the atomic minimum operation offered by CUDA provides the greatest speedup on a single device (Table II) indicating that the latency of serialized memory access can be tolerated through a high level of thread-level parallelism. Nonetheless, as atomic memory operations are not defined across devices in a multi-GPU system, a final reduction to combine the results from all streams is required in any case. Hence, multiple map kernels, each using atomic operations to implicitly perform reduction on their share of data, offer the best result in terms of performance, memory usage, and scalability to multiple cores and devices.

A. Approximate Kernel

The approximate DT kernel sacrifices accuracy for performance by considering only a subset of feature points for the map operation. The resulting speedup is directly proportional to the factor by which the size of the set is reduced. The mean absolute percentage error was used as the metric for measuring the accuracy of the approximate kernel:

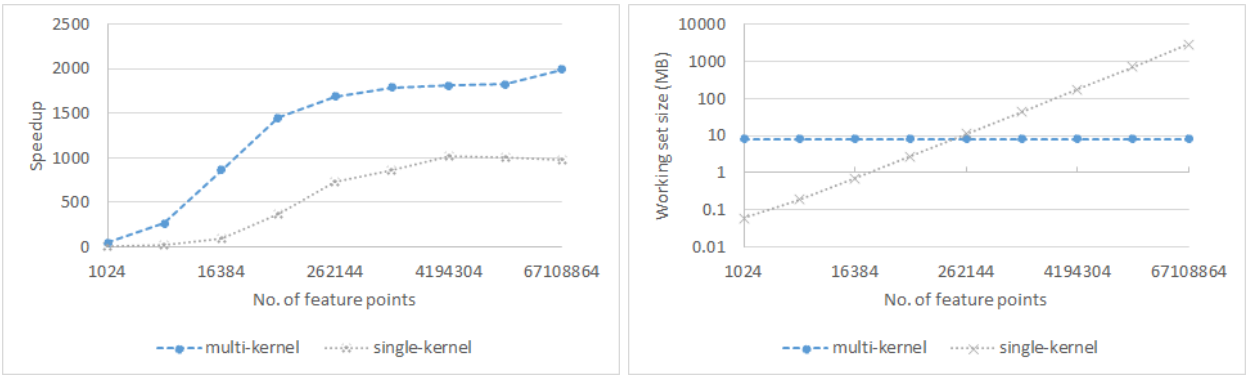
TABLE II: The actual running times in milliseconds for three variants of the distance transform algorithm on different feature set sizes: a single monolithic map and reduction kernel, multiple small kernels, and a single map kernel that performs reduction implicitly through the use of atomic operations. The number of query points is fixed at 1024.

	single kernel	multi-kernel	atomic ops.
16384	2.85	0.31	0.24
65536	2.99	0.75	0.65
262144	6.03	2.62	2.30
16777216	291.06	159.55	141.25

$$E = \frac{1}{n}\sum_{i=1}^{n}\frac{(d_i - d_i')}{d_i} \times 100$$

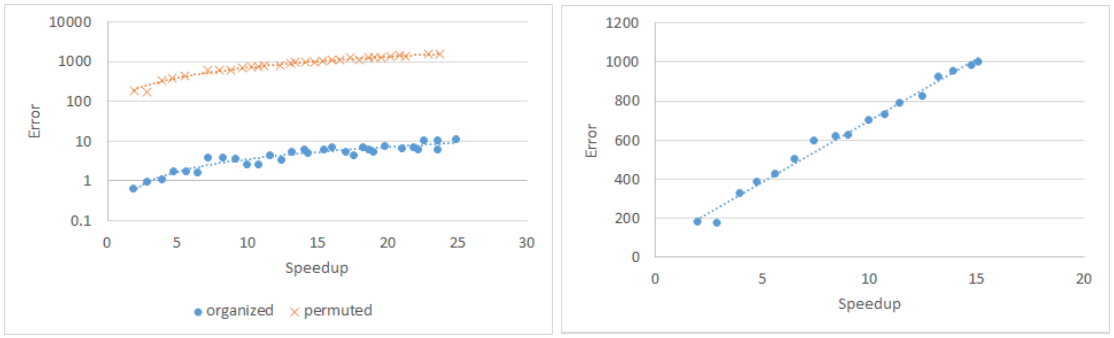
where d_i is the actual distance value and d_i' the approximate value for the ith query point in a set of size n.

It was observed that the results of the approximate DT kernel are heavily dependent on the memory organization and spatial proximity of the set of feature points. In the case of the bunny model, where the spatial proximity of the feature points in at least one dimension is mirrored by their organization in memory, a speedup fractionally over 23 was achieved while the error remained well below 10% (Figure 7a). However, if the feature points are permuted randomly, even a small speedup of two, obtained by discarding every other point, comes at the cost of a very high error. Similarly high error values are observed in the case of a uniformly distributed random set of feature points (Figure 7b). Therefore, while the approximate kernel allows dynamic control over the accuracy and speedup of the distance transform calculation, its benefits are realized when the feature points are in close proximity and this organization is reflected in the memory layout of the points. We have not yet sought to provide quantitative bounds on the spatial proximity in terms of approximation error.



(a) **(b)**

Fig. 6: (a) The speedup for the multiple-kernel and the single-kernel version of the distance transform algorithm, and (b) the corresponding size of the working set in megabytes.



(a) **(b)**

Fig. 7: The approximation error plotted against the speedup achieved by means of loop perforation for (a) the Stanford bunny model (b) a uniformly distributed random set of feature points.

B. Anisotropic Kernel

In computer graphics, *raymarching* is a commonly used image-order technique for rendering volumetric models: for every pixel in the image, a ray is shot into the volume and sampled at discrete positions along its length in front-to-back order. The process of sampling terminates when an intersection with the volumetric geometry is found or the bounds of the volume are exceeded. For a cubic volume with sides of length n, sampling each ray at unit-sized intervals yields a worst case running time of $O(n^3)$. For this reason, it is desirable to skip regions of empty space which are guaranteed not to produce any ray-voxel intersections. As the distance transform provides the maximum radius of empty space around a query point, it is often used to determine the best interval between samples along a ray. Figure 8b shows the effects of using the standard distance transform to vary the ray traversal stride: while the number of samples for most rays is close to the minimum value of 1, as indicated by the large swathes of white in the image, the result remains sub-optimal for rays that pass very close to the surface without ever intersecting it. These rays are represented by the thin dark band along the profile of the model. Additionally, as the distance transform operates uniformly in all directions, the stride length of rays continues to be constrained by geometry they have gone past. The anisotropic distance transform addresses these shortcomings by restricting calculations to feature points lying within a given angle θ of the ray's direction. Figure 8c and 8d show the number of samples when using the ADT for $\theta = \pi/5$ and $\theta = \epsilon$, where a very small ϵ is used to account for the fact that, due to floating-point roundoff errors, the dot product between vectors pointing in the same direction may not always be calculated as an exact 1.

Since calculating the angle by means of a dot product requires the vectors involved be normalized, one drawback of using the anisotropic distance transform is that it involves relatively expensive floating-point instructions, including a division and square-root operation. However, as long as the relative performance degradation of the kernel is less than the proportional reduction in the number of samples, the anisotropic distance transform achieves a net gain in raymarching performance (Figure 9).

VII. Conclusion

We have presented a CUDA implementation of the distance transform algorithm that allows pointwise computation of results at run-time, thereby, providing the twin advantages of space-efficiency and flexibility. Evidence of the latter property

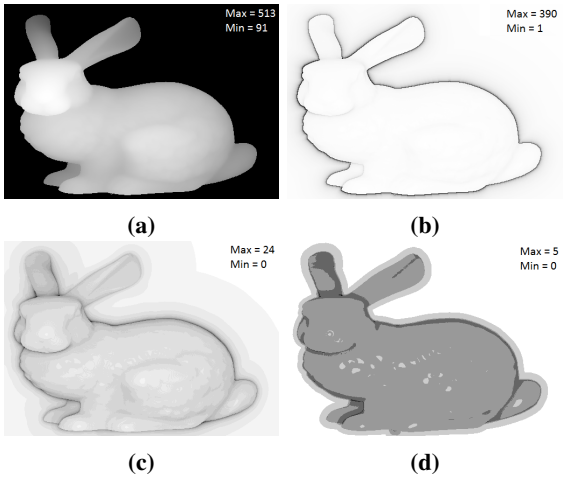


Fig. 8: The number of samples when raymarching (a) with a unit-sized stride, (b) using the standard distance transform, (c) and (d) using an anisotropic distance transform with $\theta = \pi/5$ and $\theta = \epsilon$, respectively. The contrast of each image is adjusted independently so that black represents the maximum and white the minimum number of samples per ray.

was presented by adapting the generic algorithm to calculate an approximate and an anisotropic distance transform. By decomposing the brute-force DT algorithm into a map and reduction pattern, our implementation was able to leverage the inherent parallelism and scalability of the brute-force approach while providing familiar targets for further optimization on graphics processing units.

REFERENCES

[1] M. Sramek and A. Kaufman, "Fast ray-tracing of rectilinear volume data using distance transforms," *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, no. 3, pp. 236 – 252, 2000.

[2] S. Worley, *Texturing and Modeling: A procedural approach*, 3rd ed., ser. The Morgan Kauffman Series in Computer Graphics. Morgan Kauffman, 2002, ch. Cellular Texturing.

[3] R. Fabbri, L. D. F. Costa, J. C. Torelli, and O. M. Bruno, "2d euclidean distance transform algorithms: A comparative survey," *ACM Computing Surveys*, vol. 40, no. 1, Feb 2008.

[4] M. Jones, J. Baerentzen, and M. Smarek, "3d distance fields: a survey of techniques and applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 4, Jul 2006.

[5] O. Cuisenaire and B. Macq, "Fast euclidean distance transformation by propagation using multiple neighborhoods," *Computer Vision and Image Understanding*, vol. 76, no. 2, pp. 163–172, 1999.

[6] A. Rosenfield and J. L. Pfaltz, "Sequential operations in digital picture processing," *Journal of the ACM (JACM)*, vol. 13, pp. 471–494, Oct 1966.

[7] G. Borgefors, "Distance transformations in arbitrary dimensions," *Computer vision, graphics, and image processing*, vol. 27, no. 3, pp. 321–345, 1984.

[8] ——, "Distance transformations in digital images," *Computer vision, graphics, and image processing*, vol. 34, no. 3, pp. 344–371, 1986.

[9] ——, "On digital distance transforms in three dimensions," *Computer Vision and Image Understanding*, vol. 64, no. 3, pp. 368–376, Nov 1996.

[10] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 227–248, Nov 1980.

[11] I. Ragnemalm, "The euclidean distance transform in arbitrary dimensions," in *International Conference on Image Processing and its Applications*, Apr 1992, pp. 290–293.

[12] J. Schneider, M. Kraus, and R. Westermann, "Gpu-based real-time discrete euclidean distance transforms with precise error bounds," in *In International Conference on Computer Vision Theory and Applications (VISAPP)*, 2009, pp. 435–442.

[13] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan, "Parallel banding algorithm to compute exact distance transform with the gpu," in *I3D '10 Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. ACM, 2010, pp. 83–90.

[14] J. A. Sethian, "Fast marching methods," *SIAM Review*, vol. 41, pp. 199–235, 1998.

[15] J. N. Tsitsiklis, "Efficient algorithms for globally optimal trajectories," *IEEE Transactions on Automatic Control*, vol. 40, no. 9, pp. 1528–1538, Sep 1995.

[16] R. A. Lotufo, A. A. Falcao, and F. A. Zampirolli, "Fast euclidean distance transform using a graph search algorithm," in *Proceedings XIII Brazilian Symposium on Computer Graphics and Image Processing*, 2000, pp. 269–275.

[17] J. Piper and E. Granum, "Computing distance transformation in convex and non-convex domains," *Pattern Recognition*, vol. 20, no. 6, pp. 599–615, 1987.

[18] G. Rong and T.-S. Tan, "Jump flooding in gpu with application to voronoi diagram and distance transform," pp. 109–116, 2006.

[19] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones, "Adaptively sampled distance fields: a general representation of shape for computer graphics," in *SIGGRAPH '00 Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, 2000, pp. 249–254.

[20] H. Samet, "Distance transform for images represented by quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-4, no. 3, pp. 298–303, May 1982.

[21] G. Shakhnarovich, T. Darrell, and P. Indyk, *Nearest Neighbor Methods in Learning and Vision: Theory and Practice*, 1st ed., ser. Neural Information Processing Series. The MIT Press, 2006, ch. Introduction.

[22] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software*, vol. 3, no. 3, pp. 209–226, 1977.

[23] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *SIGMOD '95 Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. ACM, 1995, pp. 71–79.

[24] D. Qiu, S. May, and A. Nuchter, "Gpu-accelerated nearest neighbor search for 3d registration," in *ICVS '09 Proceedings of the 7th International Conference on Computer Vision Systems: Computer Vision Systems*, 2009, pp. 194–203.

[25] A. Andoni and P. Indyk, "Near optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Communications of the ACM*, vol. 51, no. 1, pp. 117–122, January 2008.

[26] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2008, pp. 1–6.

[27] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer k-d trees: Processing massive nearest neighbor queries on gpus," in *Proceedings of the 31st International Conference on Machine Learning*, ser. JMLR Workshop and Conference Proceedings, 2014, vol. 32, pp. 172–180.

[28] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 891–923, Nov 1998.

[29] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *In VISAPP International Conference on Computer Vision Theory and Applications*, 2009, pp. 331–340.

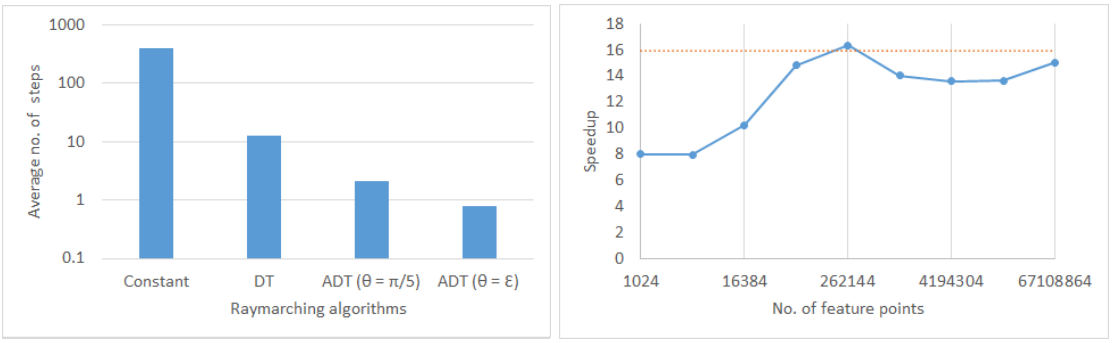
[30] J. Barnes and P. Hut, "A hierarchical o(n logn) force calculation algorithm," *Nature*, vol. 324, pp. 446–449, Dec 1986.

[31] R. Beatson and L. Greengard, "Wavelets, multilevel methods and elliptic pdes," ser. Numerical Methods and Scientific Computation. Oxford University Press, 1997, pp. 1–37.

[32] L. Nyland, M. Harris, and J. Prins, *GPU Gems 3*. Addison-Wesley Professional, 2007, ch. Fast N-Body Simulation with CUDA.

[33] NVIDIA, *CUDA C Programming Guide*, aug 2014.

[34] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Program-*



(a) **(b)**

Fig. 9: (a) The average number of samples per ray decreases exponentially when using the Anisotropic Distance Transform (ADT). For $\theta = \epsilon$, the ADT based raymarching algorithm requires 15.89 times fewer samples than the DT based one. (b) The speedup of the integer-only DT kernel over the ADT kernel involving floating point operations. As long as the speedup is less than 15.89, using the ADT provides a net gain in raymarching performance.

ming: Patterns for Efficient Computation, 1st ed. Morgan Kaufmann, jun 2008.

[35] D. B. Kirk and W. W. Hwu, *Programming Massively Parrallel Processors: A Hands-on Approach*, 2nd ed. Morgan Kauffman, 2012, ch. Data Parallel Execution Model.

[36] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: pattern-based approximation for data parallel applications," in *ASPLOS '14 Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 35–50.

[37] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *ESEC/FSE '11 Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 124–134.

[38] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *ICS '06 Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 324–334.

[39] S. Misailovic, D. M. Roy, and M. C. Rinard, "Probabilistically accurate program transformations," in *SAS'11 Proceedings of the 18th international conference on Static analysis*. Springer-Verlag, 2011, pp. 316–333.

[40] M. Levoy, "Stanford volume data archive," http://graphics.stanford.edu/data/voldata/, Aug 2014, [Online; accessed 12-Nov-2014].

[41] M. Harris, "Optimizing parallel reduction in cuda," http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf, 2007, [Online; accessed 27-Dec-2014].