Fast and Scalable Mixed Precision Euclidean Distance Calculations Using GPU Tensor Cores

Brian Curless
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, USA
bc2497@nau.edu

Michael Gowanlock
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, USA
Michael.Gowanlock@nau.edu

Abstract

Modern GPUs are equipped with tensor cores (TCs) that are commonly used for matrix multiplication in artificial intelligence workloads. However, because they have high computational throughput, they can lead to significant performance gains in other algorithms if they can be successfully exploited. We examine using TCs to compute Euclidean distance calculations, which are used in many data analytics applications. Prior work has only investigated using 64 bit floating point (FP64) data for computation; however, TCs can operate on lower precision floating point data (i.e., 16 bit matrix multiplication and 32 bit accumulation), which we refer to as FP16-32. FP16-32 TC peak throughput is so high that TCs are easily starved of data. We propose a Fast and Scalable Tensor core Euclidean Distance (FaSTED) algorithm. To achieve high computational throughput, we design FaSTED for significant hierarchical reuse of data and maximize memory utilization at every level (global memory, shared memory, and registers). We apply FaSTED to the application of similarity searches, which typically employ an indexing data structure to eliminate superfluous Euclidean distance calculations. We compare to the state-of-the-art (SOTA) TC Euclidean distance algorithm in the literature that employs FP64, as well as to two single precision (FP32) CUDA core algorithms that both employ an index. We find that across four real-world high-dimensional datasets spanning 128-960 dimensions, the mixed-precision brute force approach achieves a speedup over the SOTA algorithms of 2.5–51×. We also quantify the accuracy loss of our mixed precision algorithm to be <0.06% when compared to the FP64 baseline.

Keywords

CUDA, Euclidean Distance, GPU, Mixed Precision Floating Point, Self-Join, Similarity Search, Tensor Cores

ACM Reference Format:
Brian Curless and Michael Gowanlock. 2025. Fast and Scalable Mixed Precision Euclidean Distance Calculations Using GPU Tensor Cores. In *Proceedings of 54th International Conference on Parallel Processing (ICPP '25).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3754598.3754636

1 Introduction

Euclidean distance calculations are fundamental to numerous application domains (e.g., in the area of data analytics, they are employed for distance similarity searches [9], outlier detection [24], k-nearest neighbor searches [20], and clustering [2]). Given the numerous algorithms that rely on Euclidean distance subroutines, improving the performance of this fundamental operation can have a significant impact on a wide range of algorithms and applications.

In this paper, we present a Fast and Scalable Tensor core Euclidean Distance (FaSTED) algorithm, that is designed for moderate to high-dimensional datasets.[1] We consider comparing all points in an input dataset to each other, which has a quadratic time complexity, and only returning those points that are within a search distance, ϵ, of each other. This is a common application scenario, as it is assumed in many of the data analytics applications described above. FaSTED can be employed as a subroutine in algorithms and workloads requiring the large scale computation of Euclidean distances.

We define peak throughput as the maximum number of Tera Floating Point Operations Per Second (TFLOPS), which is the maximum throughput achievable based on TC hardware constraints. The Nvidia A100 GPU that we use in our evaluation has a peak FP32 throughput on CUDA cores of 19.5 TFLOPS, whereas TCs can achieve a throughput of 312 TFLOPS using FP16-32 [18].[2] Given that the peak throughput of TCs outlined above is a factor of 16 greater than CUDA cores, it is clear that if they can be efficiently exploited to compute Euclidean distances, there is potential for significant performance improvements over methods that employ CUDA cores.

Based on our investigation, we find that there are two major challenges that limit the performance achievable on GPU TCs for Euclidean distance calculations that we describe as follows.
(1) **Memory Bottlenecks** – A major challenge is efficiently transferring data between each level of the GPU's memory hierarchy, from global memory to shared memory and then into registers.
(2) **Insufficient Data Reuse** – Given the enormous disparity between FP16-32 TC throughput and global memory bandwidth, exploiting data reuse is critical.

This paper makes the following contributions:
(1) When copying matrix data into streaming multiprocessors (SMs), we employ inline Parallel Thread Execution (PTX) instructions to design efficient memory access patterns.

[1]The source code is publicly available: https://github.com/bwcurless/fasted.
[2]We use CUDA terminology throughout this paper.

(2) We calculate the data reuse required to maximize performance and design warp- and block-level optimizations to meet these prerequisites.
(3) We design a two-stage pipeline with asynchronous memory copies for loading data from global into shared memory, such that each warp overlaps TC computation with asynchronous loads.
(4) We compare FASTED to both TC and CUDA core state-of-the-art (SOTA) algorithms and demonstrate that it is superior in all experimental scenarios.

The paper is organized as follows: Section 2 provides relevant background information on similarity searches and an exploration of related work; Section 3 describes our proposed algorithm, FASTED; Section 4 presents the experimental evaluation; and, finally, Section 5 reflects on the work's central contributions and future research directions.

2 Background

2.1 Definitions and Problem Statement

Definition of a dataset and data points – Let D be a dataset containing $|D|$ points, where each point is defined as $p_i \in D$, and where i=1, 2, . . . , $|D|$. Each point, p_i, is defined by coordinates in d dimensions. We define the coordinates for point p_i as $p_{i,k}$, where k=1, 2, . . . , d.

Definition of the Euclidean Distance – Let $p_i \in D$ and $p_j \in D$ be two points in the dataset, D, where $p_{i,k}$ and $p_{j,k}$ refer to the k^{th} coordinates of points p_i and p_j, respectively. The Euclidean distance between p_i and p_j is typically defined as follows and an expansion yields a form that can be computed on TCs [8]:

$$\begin{aligned} dist(p_i, p_j) &= \sqrt{\sum_{k=1}^{d} (p_{i,k} - p_{j,k})^2} \\ &= \sqrt{\sum_{k=1}^{d} p_{i,k}^2 - 2p_{i,k}p_{j,k} + p_{j,k}^2}. \end{aligned} \tag{1}$$

Definition of queries and the distance similarity self-join – In this paper, we compare all points $p_i \in D$ to each other, which is a high throughput operation that is well-suited to the GPU. In database terminology, this is referred to as a self-join, where conceptually D can be considered a database table that is joined with itself. This is a typical application scenario for the data analytics applications described in Section 1.

2.2 Scenarios for Euclidean Distance Calculations

We describe two common application scenarios for the calculation of Euclidean distances below.

2.2.1 Scenario 1: Brute Force Distance Similarity Searches. The distance is calculated between all permutations of points in a dataset D (containing $|D|$ points), and only the pairs where the distance is $\leq \epsilon$ are returned. This has a time complexity of $O(|D|^2)$.

2.2.2 Scenario 2: Index-Supported Distance Similarity Searches. To improve the performance of brute force similarity searches (described above), algorithms will typically first construct an indexing data structure. The index is used to prune the search between points that are sufficiently far from each other, decreasing the amount of comparisons required.

Table 1: Comparison of matrix sizes for FP16-32.

Size (m-n-k)	WMMA API	PTX `mma`
16×16×16	✓	
32×8×16	✓	
8×32×16	✓	
8×8×4		✓
16×8×8		✓
16×8×16 (Used by FASTED)		✓

2.3 GPU Tensor Cores

In this section, we discuss the operation of TCs and how they are programmed on Nvidia GPUs. TCs can operate on different floating point data types including FP64, FP32, and FP16-32. There are several libraries that directly employ TCs for linear algebra operations, including cuBLAS and CUTLASS.[3] These libraries are suitable when their subroutines can be directly embedded into an algorithm. However, they are unsuitable for our purposes, as they do not offer sufficient granularity or control. There are two other ways to use TCs that provide more flexibility: the WMMA API [17], and PTX instructions.[4]

The WMMA API and PTX instructions both load data from shared memory into registers that are shared across an entire warp to compute small submatrices (called register fragments). However, the WMMA API is limited to larger matrix sizes (see Table 1), does not specify the register layout, and yields less control over memory addressing. In this paper, we use PTX instructions as they provide the most flexibility to achieve the highest throughput.

Listing 1: A 16×16 ldmatrix PTX instruction.

```
asm volatile (
"ldmatrix.sync.aligned.x4.m8n8.shared.b16"
"{%0, %1, %2, %3}, [%4];"
: "=r"(A[0]), "=r"(A[1]), "=r"(A[2]), "=r"(A[3])
: "r"(smem_ptr));
```

Listing 2: A 16×8×16 mma.sync PTX instruction.

```
asm volatile (
"mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32"
"{%0, %1, %2, %3}, "
"{%4, %5, %6, %7}, "
"{%8, %9}, "
"{%10, %11, %12, %13};"
:"=f"(D[0]), "=f"(D[1]), "=f"(D[2]), "=f"(D[3])
:"r"(A[0]), "r"(A[1]), "r"(A[2]), "r"(A[3]),
"r"(B[0]), "r"(B[1]),
"f"(C[0]), "f"(C[1]), "f"(C[2]), "f"(C[3]));
```

There are two main PTX instructions for using TCs. The first is `ldmatrix`, shown in Listing 1, which creates a fragment of a matrix that is stored in registers shared across a warp. The second is `mma.sync`, shown in Listing 2, which computes $\boldsymbol{D} = \boldsymbol{A} \times \boldsymbol{B} + \boldsymbol{C}$, where $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$ are each fragments that were created by prior `ldmatrix` instructions. When using FP16-32 instructions, the data stored in matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ are represented in FP16 as 16×16

[3]cuBLAS: https://developer.nvidia.com/cublas; CUTLASS: https://github.com/NVIDIA/cutlass
[4]https://docs.nvidia.com/cuda/archive/12.0.1/cuda-c-programming-guide/#element-types-and-matrix-sizes

and 16×8 fragments, while the accumulator matrices, ***C*** and ***D***, are represented in FP32 as 16×8 fragments.

2.4 Literature on Expanding TC Applicability

There are few papers in the literature that extend the applicability of GPU TCs. Most of the papers below are not directly related to this paper, as they examine differing use cases for TCs; however, we summarize them for context.

A pioneering paper by Dakkak et al. [5] used TCs to perform scan and reduction operations. They show that their reduction algorithm on an Nvidia V100 GPU achieves a 100× speedup over using CUDA cores. Ji and Wang [13] used TCs to accelerate the DBSCAN clustering algorithm. In their paper, they use cosine distance matrices to determine the ϵ-neighborhood of points when clustering. They achieve a 2.61× speedup using TCs over CUDA cores. Li et al. [14] use TCs to improve the performance of half precision Fast Fourier Transforms (FFTs) and achieve a speedup of 3.03× over the cuFFT library.[5] Wan et al. [21] used TCs to accelerate the Number Theoretic Transform (NTT) operation that is a bottleneck in the post-quantum cryptography algorithm, CRYSTALS-Kyber [1]. They demonstrate that their TC NTT algorithm yields a speedup of 6.47× over the CUDA core algorithm.

Cui [4] proposes using TCs for finite element methods (FEM), which are computationally expensive and thus a good target for GPU TCs. Similarly to Cui [4], we use PTX instructions to eliminate bank conflicts when copying data between registers and shared memory. Unlike Cui [4], we optimize copying data from global memory into shared memory in a bank-conflict-free manner.

2.5 Related Work on using Tensor Cores for Euclidean Distance Calculations

To our knowledge, there is one paper in the literature on employing TCs to compute Euclidean distances. The paper proposes "Tensor Euclidean Distance Join" (TED-Join) [8], which is used to compute the Euclidean distances between all points in a dataset for similarity searches (the two scenarios outlined in Sections 2.2.1–2.2.2).

TED-Join [8] is designed for FP64 calculations and uses the WMMA API. It was found to perform best when the data dimensionality (d) is low rather than high. Given that TED-Join is the only other TC algorithm in the literature, we compare FaSTED to TED-Join in our evaluation.

2.6 Literature on Distance Similarity Searches

In this section, we discuss the literature on the distance similarity self-join as defined in Section 2.2.2. GDS-Join [9, 10] is a CUDA core GPU algorithm that uses a grid-based index for the distance similarity self-join. The algorithm uses several optimizations, including assigning points to threads within warps such that they have low intra-warp load imbalance, processing warps from largest to smallest workload to prevent inter-warp load imbalance towards the end of a kernel execution, and reordering the coordinates in the dataset to increase the probability that when computing the distance between points, the distance calculation will abort early (we refer to this as short circuiting).

[5]https://developer.nvidia.com/cufft

MiSTIC [6] is another CUDA core distance similarity search algorithm that is similar to GDS-Join. A major difference between GDS-Join and MiSTIC is that the latter uses a combination of metric- and coordinate-based partitioning to prune the search for nearby points of each query point. Additionally, MiSTIC uses incremental index construction to select between candidate partitions. MiSTIC outperforms GDS-Join across all experiments (on datasets up to d=90). Interestingly, the authors find that one reason MiSTIC outperforms GDS-Join is that it has better load balancing properties across warps and blocks executing on SMs. The algorithm we propose in this paper, FaSTED, has perfect load balancing, which indicates that it could have a performance advantage over these CUDA core algorithms.

In Section 4, we compare FaSTED to GDS-Join and MiSTIC, as they are SOTA CUDA core algorithms. FaSTED does not use an index, so it is disadvantaged compared to these approaches. We do not compare to CPU algorithms in this paper, as the GPU algorithms are significantly faster [8–10], particularly when processing high-dimensional datasets.

3 FaSTED

We present our Fast and Scalable Tensor core Euclidean Distance (FaSTED) algorithm, which exploits (FP16-32) TCs on GPUs to compute Euclidean distances. We begin by describing how TCs can be used for Euclidean distance calculations and then describe the optimizations required to effectively utilize FP16-32 TCs using the A100 as our platform (the method is generalizable to other TC-equipped GPU models as well).

3.1 Overview of Euclidean Distance Calculations on Tensor Cores

Figure 1 shows an illustrative example of calculating Euclidean distances, where the distances between points $p_1, \ldots, p_{16}$ and $p_1, \ldots, p_8$ are computed. The steps below describe how TCs can be used to calculate the resulting set of 128 Euclidean distances:

Step 1: On CUDA cores, we precompute the sum of the squared dimensions, s_i, for each point $p_i \in D$, $s_i = \sum_k p_{i,k}^2$ and store them in global memory. All summations round towards zero to match TC rounding [7].

Step 2: Tensor cores operate on small matrices that are stored in registers; we refer to these matrices as "register fragments," or "rf". There are three register fragments required to compute distances:

(1) $\boldsymbol{P}_{\mathrm{rf}}$: 16 dimensions from 16 points in dataset D. We refer to these simply as the "points".

(2) $\boldsymbol{Q}_{\mathrm{rf}}$: 16 dimensions from 8 points in dataset D in a transposed layout. We refer to these as the "query points".

(3) $\boldsymbol{A}_{\mathrm{rf}}$: A 16×8 fragment initialized with zeros.

We use TCs to multiply and accumulate these three fragments to obtain: $\boldsymbol{A}_{\mathrm{rf}} = \boldsymbol{P}_{\mathrm{rf}} \times \boldsymbol{Q}_{\mathrm{rf}} + \boldsymbol{A}_{\mathrm{rf}}$. As TCs can only compute inner products of 16 dimensions per operation (see Table 1), we must iteratively load new point data into $\boldsymbol{P}_{\mathrm{rf}}$ and $\boldsymbol{Q}_{\mathrm{rf}}$ and accumulate into $\boldsymbol{A}_{\mathrm{rf}}$ $\frac{d}{16}$ times.

Step 3: On CUDA cores, we combine the elements of $\boldsymbol{A}_{\mathrm{rf}}$ ($a_{i,j}$), along with s_i and s_j (step 1) to compute the square of the final distance (from Equation 1) $dist_{i,j}^2 = -2 \times a_{i,j} + s_i + s_j$.

3.2 Overview of the Data Path from Global Memory to Registers

While computing Euclidean distances using TCs is conceptually straightforward, computing step 2 in Section 3.1 with high throughput is challenging. Previous GPU work that computed distance similarity searches focused on algorithmic optimizations using indexed search methods to reduce the search space, as well as short circuiting techniques to reduce computation (Section 2.5–2.6). In contrast, maximizing performance on TCs requires a balanced workload that these index-supported algorithms cannot provide (e.g., they have non-deterministic execution pathways that cause warp divergence). We propose several optimizations to balance the workload and maximize TC throughput.

Computing a large matrix of Euclidean distances with FASTED is shown in Figures 1-4 and we describe them as follows.

The foundation is a single PTX instruction (shown in Listing 2 and Figure 1) that computes a 16×8×16 "k-slice" of distances. A k-slice refers to a contiguous subset of a point's total dimensions (e.g. four d=16 k-slices together represent a d=64 point).

Transferring data from global memory into registers for the MMA PTX instruction is a bottleneck. Box #1 estimates the amount of data that needs to be reused when reading from global memory and reading from shared memory in order to reach peak throughput. As will be described below, this estimation derives tile size prerequisites that ensure conditions (1) & (2) from Box #1 are met.

Box #1: Selecting Tile Sizes Based on Data Reuse

In a matrix multiply-accumulate product, there are two floating-point operations for every two elements processed. Peak throughput (312 TFLOPS) requires reading the following number of elements per second:

$$\frac{312\ \text{TFLOP}}{\text{second}} \times \frac{2\ \text{elements}}{2\ \text{FLOP}} = \frac{312 \times 10^{12}\ \text{elements}}{\text{second}}.$$

(1) Global memory bandwidth is 1.5 TB/s. With a 100% hit rate, the L2 cache increases this to 6.4 TB/s. Therefore each value read from global memory must be reused:

$$\frac{312 \times 10^{12}\ \text{elements}}{\text{second}} \times \frac{2\ \text{B}}{1\ \text{element}} \times \frac{\text{second}}{6.4\ \text{TB}} = \mathbf{98\ times}.$$

(2) Shared memory bandwidth is 17.9 TB/s. Therefore each value read from shared memory must be reused:

$$\frac{312 \times 10^{12}\ \text{elements}}{\text{second}} \times \frac{2\ \text{B}}{1\ \text{element}} \times \frac{\text{second}}{17.9\ \text{TB}} = \mathbf{35\ times}.$$

We create a "warp tile" (Figure 2) that contains multiple P_{rf} and Q_{rf} fragments. Each P_{rf} is multiplied by every Q_{rf}, and the results are accumulated into the corresponding A_{rf}. Only one d=16 k-slice can fit into registers at a time, so fragments are loaded and accumulated into A_{rf} in four separate iterations. This sequence computes a 64×64×64 warp tile.

There are four schedulers and four TCs per SM, so we elect to run four warp tiles simultaneously on each SM. These four warps form a "block tile" (Figure 3) that computes a 128×128 tile of distances $dist(p_i, p_j)^2$. A block tile transfers two different "block fragments," which are d=64 k-slices of 128 points (P_{bf} and Q_{bf}), from global memory to shared memory that each warp tile reads from during its four iterations. This repeats until all dimensions of the points have been accumulated by the warp tiles and the final distances have been computed. When a block completes a tile, a work queue provides the next tile to compute and this process repeats (Figure 4).

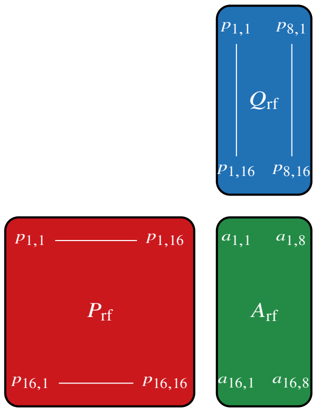
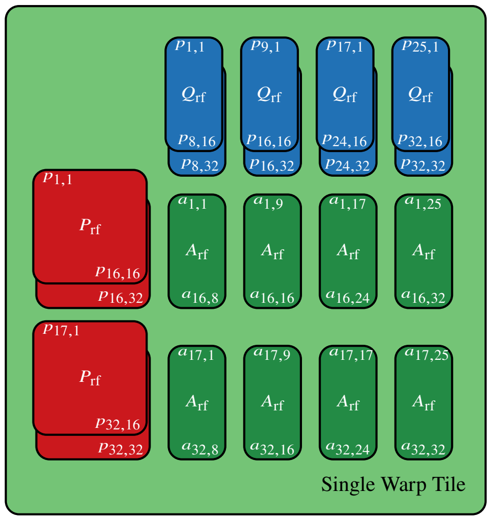


Figure 1: A single 16×8×16 TC computation used to accumulate the distances between two points p_i and p_j into $a_{i,j}$. Only the corner elements in each matrix are shown for illustrative purposes.



$p_{1,1}$ Q_{rf} $p_{8,16}$ $p_{8,32}$
$p_{9,1}$ Q_{rf} $p_{16,16}$ $p_{16,32}$
$p_{17,1}$ Q_{rf} $p_{24,16}$ $p_{24,32}$
$p_{25,1}$ Q_{rf} $p_{32,16}$ $p_{32,32}$
$p_{1,1}$ P_{rf} $p_{16,16}$ $p_{16,32}$
$p_{17,1}$ P_{rf} $p_{32,16}$ $p_{32,32}$
$a_{1,1}$ A_{rf} $a_{16,8}$
$a_{1,9}$ A_{rf} $a_{16,16}$
$a_{1,17}$ A_{rf} $a_{16,24}$
$a_{1,25}$ A_{rf} $a_{16,32}$
$a_{17,1}$ A_{rf} $a_{32,8}$
$a_{17,9}$ A_{rf} $a_{32,16}$
$a_{17,17}$ A_{rf} $a_{32,24}$
$a_{17,25}$ A_{rf} $a_{32,32}$
Single Warp Tile

Figure 2: Two iterations of a 32×32×16 warp tile are shown. It is made up of many register fragments from Figure 1. Only the upper left and lower right coordinates of each fragment are specified for illustrative purposes.

3.3 Optimizations: Mitigating the Memory Bottleneck

3.3.1 L2 Cache: Maximizing Data Reuse by Ordering Block Tiles. The L2 cache has a bandwidth of 6.4 TB/s which is ~4× greater than global memory. Our algorithm is able to utilize most of this bandwidth by having blocks retrieve work from a queue such that

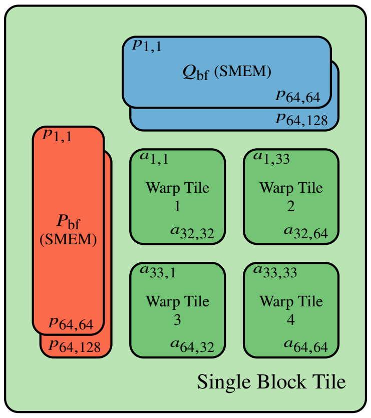

$p_{1,1}$
Q_{bf} (SMEM)
$p_{64,64}$
$p_{64,128}$
$p_{1,1}$
P_{bf} (SMEM)
$p_{64,64}$
$p_{64,128}$
$a_{1,1}$ Warp Tile 1 $a_{32,32}$
$a_{1,33}$ Warp Tile 2 $a_{32,64}$
$a_{33,1}$ Warp Tile 3 $a_{64,32}$
$a_{33,33}$ Warp Tile 4 $a_{64,64}$
Single Block Tile

Figure 3: Two iterations of a 64×64×64 block tile are shown. Two d=64 block fragments, P_{bf} and Q_{bf}, are paged from global memory to shared memory (SMEM). Four warp tiles (Figure 2) each page a portion of the block fragments into their own register fragments and accumulate the distances. As d=128, this sequence is repeated for two iterations.

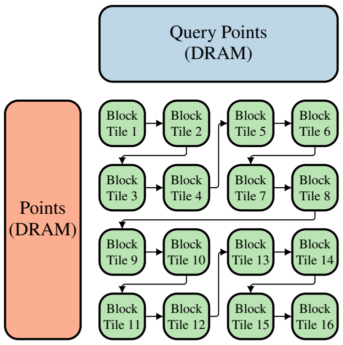


Figure 4: An illustration of how a work queue orders block tile computations into small squares to improve the L2 cache hit rate when reading from global memory (DRAM).

concurrently executing block tiles read similar data elements (point coordinates) to maximize spatial locality across SMs (Figure 4).

3.3.2 Single Block Tile Shared Memory Buffering. Every block tile is responsible for computing a 128×128 tile of the distance matrix. A d=64 k-slice of the input data is stored in shared memory and used by all four warps. This block tile size yields 128× data reuse which is more than that needed to reach peak TC throughput (Box #1).

3.3.3 L1 Cache elimination. Each SM has 192 KB of memory that can be used as an L1 cache or directly manipulated as shared memory. In order to maximize throughput, most of the L1 cache has been configured as shared memory to allow us to use a pipelined execution that hides memory access latency, which will be described in the following sections.

3.3.4 Asynchronous Global to Shared Memory Transfers. Normal synchronous methods of transferring data from global to shared memory use synchronous copies where the data path is as follows: global memory → L2 Cache → L1 Cache → registers → shared memory. The A100 GPU supports asynchronous memory copies [18] using `cuda::memcpy_async` that skips the L1 cache and registers, storing directly into shared memory. This reduces latency and register pressure.

3.3.5 Global to Shared Memory Transfer Pipelining. There is sufficient shared memory to create a multi-stage pipeline of memory transfers using `cuda::pipeline` with `cuda::memcpy_async` as described in Section 3.3.4. Our algorithm issues asynchronous memory copies in a two-stage pipeline and overlaps computation with data transfer to hide latency.

3.3.6 Multiple Blocks Per Streaming Multiprocessor. We use multiple blocks that are executed in parallel on each SM to hide high latency instructions like MMA and memory transfers. We elect to maximize the following: block tile size, k-slice width, and pipeline depth (two levels) while leaving sufficient shared memory and registers to allow two blocks to run simultaneously on each SM.

3.3.7 Single Warp Tile. We create a 64×64×16 warp tile (Figure 2) that reuses each $\boldsymbol{P}_{\text{rf}}$ and $\boldsymbol{Q}_{\text{rf}}$ fragment, 4 and 8 times respectively, to achieve the required reuse calculated in Box #1. To reduce register pressure, only a single d=16 k-slice is stored in registers at a time.

3.3.8 Memory Address XOR Swizzling. When paging data into and out of shared memory, we ensure that all global reads are fully coalesced, while simultaneously eliminating shared memory bank conflicts. To achieve this, the shared memory addresses are swizzled as values are stored (using XOR), and unswizzled as they are read into registers. Figures 5–7 illustrate the global memory, shared memory, and register layouts, respectively, showing the path that data takes when being read from global memory and stored into the registers corresponding to a single fragment.

The FP16 point data is stored in global memory in row-major order with each point having 128 B alignment. The first d=64 of the first 8 points are shown in Figure 5. Groups of 8 threads work together to load a single d=64 k-slice of a point from global memory and store it in shared memory.

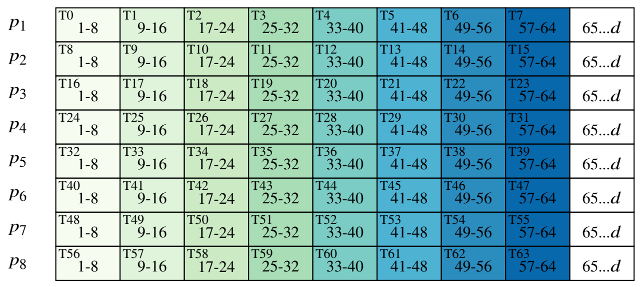
Shared memory contains 32 discrete 4 B banks. Figure 6 shows which shared memory bank each d=8 slice of point data was stored in when loading from global memory (Figure 5). The destination address is “swizzled,” meaning that for a given d=8 slice s of a point p_i, the destination shared memory address is:

$$A_{\text{dest}} = 8 \cdot (i-1) + s \oplus ((i-1) \bmod 8). \tag{2}$$

Figure 6 shows how each d=8 slice in a group of 8 threads will be stored in a unique bank and is thus free of bank conflicts.

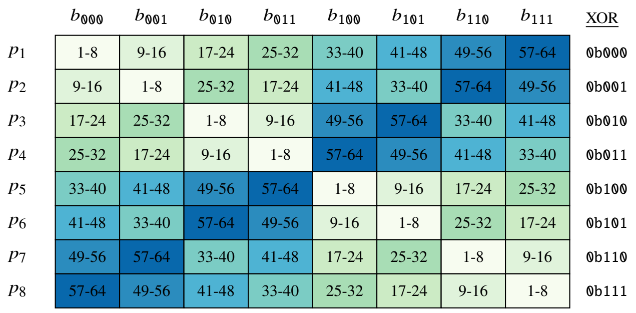
While swizzling the data layout is not required to have coalesced and bank-conflict-free shared memory stores, it is imperative for loading the point data from shared memory and storing it into registers. To load a single fragment $\boldsymbol{P}_{\text{rf}}$, there are four phases, or memory transactions, in a `ldmatrix` instruction (see Listing 1 and Figure 7). Each phase generates a single 128 B memory transaction that must be bank-conflict-free to achieve maximum throughput.

To see that banks are accessed in a conflict-free manner, observe that each d=8 slice of data being accessed in a single phase in Figure 7 is located in a unique bank in Figure 6. In contrast, a



p_1	T0 1-8	T1 9-16	T2 17-24	T3 25-32	T4 33-40	T5 41-48	T6 49-56	T7 57-64	65...d
p_2	T8 1-8	T9 9-16	T10 17-24	T11 25-32	T12 33-40	T13 41-48	T14 49-56	T15 57-64	65...d
p_3	T16 1-8	T17 9-16	T18 17-24	T19 25-32	T20 33-40	T21 41-48	T22 49-56	T23 57-64	65...d
p_4	T24 1-8	T25 9-16	T26 17-24	T27 25-32	T28 33-40	T29 41-48	T30 49-56	T31 57-64	65...d
p_5	T32 1-8	T33 9-16	T34 17-24	T35 25-32	T36 33-40	T37 41-48	T38 49-56	T39 57-64	65...d
p_6	T40 1-8	T41 9-16	T42 17-24	T43 25-32	T44 33-40	T45 41-48	T46 49-56	T47 57-64	65...d
p_7	T48 1-8	T49 9-16	T50 17-24	T51 25-32	T52 33-40	T53 41-48	T54 49-56	T55 57-64	65...d
p_8	T56 1-8	T57 9-16	T58 17-24	T59 25-32	T60 33-40	T61 41-48	T62 49-56	T63 57-64	65...d

Figure 5: The row-major global memory layout for the first 8 points is shown here. The threads that are responsible for reading each 8 dimension chunk of data from global memory and storing it in shared memory are labeled in the top left corner of each rectangle (T0–T63).



	b_{000}	b_{001}	b_{010}	b_{011}	b_{100}	b_{101}	b_{110}	b_{111}	XOR
p_1	1-8	9-16	17-24	25-32	33-40	41-48	49-56	57-64	`0b000`
p_2	9-16	1-8	25-32	17-24	41-48	33-40	57-64	49-56	`0b001`
p_3	17-24	25-32	1-8	9-16	49-56	57-64	33-40	41-48	`0b010`
p_4	25-32	17-24	9-16	1-8	57-64	49-56	41-48	33-40	`0b011`
p_5	33-40	41-48	49-56	57-64	1-8	9-16	17-24	25-32	`0b100`
p_6	41-48	33-40	57-64	49-56	9-16	1-8	25-32	17-24	`0b101`
p_7	49-56	57-64	33-40	41-48	17-24	25-32	1-8	9-16	`0b110`
p_8	57-64	49-56	41-48	33-40	25-32	17-24	9-16	1-8	`0b111`

Figure 6: The point data from Figure 5 is copied into the shared memory layout shown here one row at a time. The destination bank, b, of each 8 dimension chunk is swizzled by computing the XOR of the original bank b with the row number (shown on the right). Each column in the table represents a group of 4 of the 32 shared memory banks. Reordering the data makes the `ldmatrix` instruction, shown in Figure 7, access unique banks in each phase of its operation.

simple row-major copy from global memory (Figure 5) into shared memory would create 8-way bank conflicts during each phase.

3.3.9 Shared Memory Alignment. Shared memory allocations do not have alignment guarantees. Alignment specifiers, `__align__(128)`, were added to shared memory definitions to ensure 128 B alignment. This enables swizzling to place the data in the appropriate bank, thus mitigating bank conflicts.

4 Experimental Evaluation

4.1 Experimental Methodology

4.1.1 Platform. We conduct our experiments on a platform with 2× AMD EPYC 7542 CPUs (64 physical cores total) clocked at 2.9 GHz with 512 GiB of main memory containing an Nvidia A100 GPU with 40 GiB of global memory (PCIe model). The host code uses C/C++ and the FᴀSTED GPU code is parallelized using CUDA v.12.6.3. All host code is compiled with the O3 optimization flag.

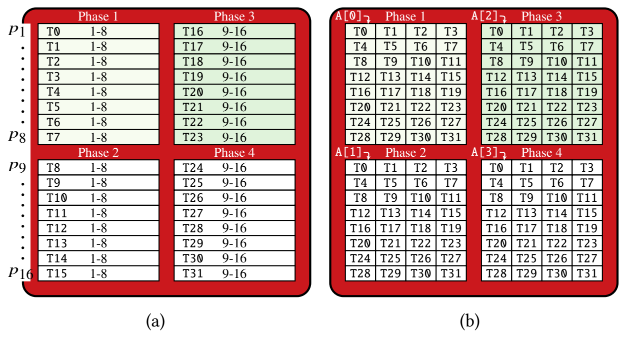


Figure 7: (a) Single fragment P_{rf} shared memory access layout. (b) Single fragment P_{rf} register layout. The result of a single 16×16 `ldmatrix` instruction reading from shared memory and storing into a fragment made up of registers is shown above. Each 8 dimension chunk is labeled with the thread responsible for reading it from shared memory. After each thread reads its 16 B chunk from shared memory, that data is shuffled into the registers of four different threads (b). For example, T0 reads p_1 dimensions k=1–8, but it is stored into four registers, owned by threads 0-3.

Table 2: Summary of optimized parameters.

Parameter	Optimized Value
Block tile dispatch shape	8×8 blocks
Block tile iteration size	128×128×64
Number of blocks in grid	2× # of SMs (216 total)
Warp tile iteration size	64×64×16
Warps per block	4
Pipeline depth	2

Reported response times are averaged over three trials. In all experiments using FᴀSTED or the reference implementations, the variation between time trials is negligible, such that the error bars are smaller than the data points on the plots, so we omit plotting these statistics. However, for completeness, we include this information in the associated figures for FᴀSTED.

We examine two main metrics when measuring performance. *Response time:* We measure the response time of the algorithms. *Derived TFLOPS:* We report the TFLOPS yielded by the TCs by taking the total number of operations and dividing it by the measured response time. We refer to this when we discuss TFLOPS.

To make a fair comparison between all implementations, we include all overheads, such as allocating memory, transferring data to and from the GPU, and storing the results in main memory. Regarding methods that employ indexing, we include all index construction overheads.

4.1.2 Implementations. We compare FᴀSTED to several GPU implementations. In what follows, we outline the reference implementations, their configurations, and the default configuration for FᴀSTED. We summarize the implementations in Table 3.

FᴀSTED – We configure FᴀSTED using the parameters outlined in Table 2, which were motivated in Section 3.

Table 3: Comparison of implementation properties.

Implementation	GPU Core	Precision	Scenario 1: Brute Force (Sec. 2.2.1)	Scenario 2: Index-Supported (Sec. 2.2.2)
FaSTED	Tensor	FP16-32	✓	
TED-Join-Brute	Tensor	FP64	✓	
TED-Join-Index	Tensor	FP64		✓
GDS-Join	CUDA	FP32		✓
MiSTIC	CUDA	FP32		✓

TED-Join – As described in Section 2.5, TED-Join [8] is the only other algorithm in the literature that computes Euclidean distances on TCs, and it uses double precision floating point (FP64) values.

TED-Join exceeds the shared memory capacity and fails to compile when a dataset contains points with $d > 128$. We elected to modify TED-Join to allocate part of the L1 cache as shared memory to allow us to employ datasets with $d \leq 384$. TED-Join can operate in two modes, the first using brute force searches, and the second using an index for index-supported range queries. We refer to these variants as **TED-Join-Brute** and **TED-Join-Index**, respectively.
GDS-Join – As described in Section 2.6 this algorithm performs index-supported range queries using CUDA cores. We use the same optimizations and configuration as described in the GDS-Join papers [9, 10], so we only outline the differences here. We configure GDS-Join to use FP32 data, which provides a baseline of comparison to FaSTED that does not use FP64 like TED-Join. Lastly, we use a batch size of $b_s = 2 \times 10^9$ for batching the result set from the GPU to the host. We also employ GDS-Join as executed using FP64 to compute the accuracy of FaSTED's mixed-precision results. The code is publicly available.[6]
MiSTIC – This algorithm is similar to GDS-Join above as it uses CUDA cores (Section 2.6). We use the same configurations and optimizations as outlined in the MiSTIC paper [6] and use a block size of 256 with 1024 blocks per kernel invocation (multiple invocations are used to batch the result set from device to host). The algorithm uses 6 levels and selects between 38 candidate layers at each level when incrementally constructing the index. MiSTIC is configured to use FP32 instead of FP64 (the latter was used in the evaluation in the MiSTIC paper [6]). The code is publicly available.[7]
Short circuiting of distance calculations – The CUDA core algorithms (GDS-Join and MiSTIC) abort computing the distance between each pair of points when the running total distance exceeds ϵ. When points in a dataset are very spread out, this saves significant computation time. The TC algorithms (TED-Join and FaSTED) are less flexible because they compute the distances between all permutations of two sets of points simultaneously. FaSTED compares an entire 128×128 block of points at a time. As FaSTED is computing 16,384 distances at a time, and every single point would need to short circuit for the optimization to be viable, we forego this optimization. TED-Join compares an 8×8 matrix of points at a time in FP64 mode, which is small enough that it does employ short circuiting. TED-Join-Brute does not abort early.

4.1.3 Datasets & Selectivity Values. **Real-World Datasets** – We use real-world datasets to compare the performance of FaSTED to reference implementations that employ indexing methods because their performance is impacted by the data distribution and the search radius, ϵ. These datasets are standard benchmarks for evaluating search algorithms on high-dimensional datasets [11, 16, 19, 22, 23], are publicly available[8], and are summarized in Table 4.
Synthetic Datasets – Table 4 outlines the dataset size and dimensionality of the synthetic dataset that we use in the evaluation. We use this class of datasets to compute the throughput of the brute force approaches, which require comparing all points to each other; therefore, the data distribution does not impact performance.
Selectivity of the Range Queries – The search radius, ϵ, used in a similarity search will impact the degree of pruning afforded by an indexing data structure. A large search radius that returns a large number of neighbors per point in a dataset will reduce the degree of index pruning, and a small value of ϵ will allow many points to be ignored when performing distance calculations. To standardize experiments across datasets, we employ the selectivity, which refers to the mean number of points found by each point searched in the dataset. The selectivity is defined as $S=(|R|-|D|)/|D|$, where $|R|$ is the total result set size (the number of pairs of points found within ϵ of each other). Common selectivity values are used to ensure that searches have a proportional amount of work to each other; otherwise, comparing performance between datasets is less meaningful. In our evaluation, we select values of ϵ for each dataset that yield three levels of selectivity (small, medium, and large), and these are defined as S_s=64, S_m=128, and S_l=256. The ϵ values corresponding to these selectivity values are given for the real-world datasets in Table 4.

Table 4: *Top:* Real-world datasets that we use in our evaluation. The dataset size, $|D|$, and data dimensionality, d, are shown, in addition to the ϵ values that correspond to three target selectivity values that we employ (S_s, S_m, and S_l). *Bottom:* Synthetic datasets that we use in our evaluation.

Real-World Datasets					
Dataset	$\|D\|$	d	$\epsilon(S_s)$	$\epsilon(S_m)$	$\epsilon(S_l)$
Sift10M	10,000,000	128	122.5	136.5	152.5
Tiny5M	5,000,000	384	0.1831	0.2045	0.2275
Cifar60K	60,000	512	0.6289	0.6591	0.6914
Gist1M	1,000,000	960	0.4736	0.5292	0.5937
Synthetic Datasets					
Dataset	$\|D\|$	d			
Synth	$10^{3+n/3}$ where $n \in [0, 1, \ldots, 9]$	$d \in 2^n$ where $n \in [6, 7, \ldots, 12]$			

4.2 FaSTED: Performance as a Function of Dataset Size and Dimensionality

With all of our optimizations described in Section 3.3 enabled, we examine how the performance of FaSTED varies as a function of dataset size ($|D|$) and dimensionality (d). Some of the optimizations are sensitive to these parameters so we vary them by using the *Synth* dataset and plot the throughput as derived TFLOPS. We only include the kernel execution times to derive TFLOPS as we are interested in understanding FP16-32 performance without the additional overheads included in the end-to-end response time (e.g., data transfers between host and GPU).

[6]https://github.com/mgowanlock/gpu_self_join
[7]https://github.com/bwd29/self-join-MiSTIC
[8]https://www.cse.cuhk.edu.hk/systems/hash/gqr/datasets.html

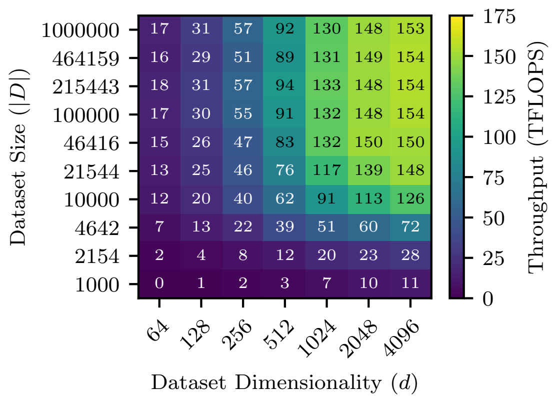


Figure 8: The number of TFLOPS achieved using FaSTED as a function of dataset size ($|D|$) and dimensionality (d) on the *Synth* class of datasets. The maximum throughput was 154 TFLOPS with a standard deviation of 0.02.

Figure 8 shows the results of this experiment. Our maximum throughput is roughly 150 TFLOPS, which can be achieved with a minimum dataset size of $|D|$=46416 and dimensionality of d=2048. In practice, this indicates that only a small to moderate dataset size is needed to reach peak performance, assuming that the dataset contains a sufficient number of dimensions.

Recall that because FaSTED is brute force, its performance is not impacted by the data distribution, so these results apply to any real-world dataset of the same size and dimensionality. To estimate FaSTED's performance on a dataset with a dimensionality that is not a multiple of 64, one must use the throughput measurements from the next multiple of 64. In other words, for low dimensionality like d=65, 63 dimensions of zeros are added, and the throughput is almost halved. This effect on performance decreases as dimensionality increases.

4.3 FaSTED: Impact of Optimizations

In Section 3.3, we described numerous optimizations and implementation details that were primarily used to mitigate the memory bottleneck when using FP16-32 TCs. Here, we present the relative impact of those optimizations by performing a leave-one-out evaluation that disables an optimization while leaving the remainder of the optimizations enabled. All optimizations were individually disabled, except for the `cuda::memcpy_async` optimization which required disabling the multi-stage pipeline as well.[9] We select the *Synth* dataset with $|D|$=10^5 and d=4096 because as we showed in Section 4.2, this dataset size and dimensionality are sufficient to reach our maximum throughput with FaSTED.

From Table 5, we find that there are several optimizations that have an exceptional impact on performance, and we outline them as follows: (i) We find that warp tile optimization has the greatest impact because reading data from shared memory into registers and reusing the k-slice data numerous times is needed to ensure that the TCs are not underutilized. (ii) The asynchronous memory copies from global to shared memory are critical to reducing memory latency. (iii) Using a block tile with four warps reusing data from shared memory significantly improves performance.

[9]Synchronous memory copies cannot be pipelined using the libcudacxx API: https://nvidia.github.io/cccl/libcudacxx/extended_api/synchronization_primitives/pipeline.html

Table 5: Performance sensitivity study using the leave-one-out method. We enable all optimizations but leave each of the optimizations out in isolation to determine their relative impact on performance. Experiments are conducted using the *Synth* dataset with $|D| = 10^5$ and d=4096.

Disabled Optimization	Section	Derived TFLOPS
Block Tile Ordering	3.3.1	133.1
Block Tile	3.3.2	95.8
Memcpy Async & Multi-stage Pipeline	3.3.4–3.3.5	48.6
Multi-stage Pipeline	3.3.5	145.0
SM Block Residency	3.3.6	110.8
Warp Tile	3.3.7	38.0
Swizzled SMEM Layout	3.3.8	120.8
Shared Memory Alignment	3.3.9	120.7
All Optimizations Enabled	3.3	154

Despite these three optimizations having an outsized performance impact, none of the optimizations were insignificant, as they were all needed to reach our maximum throughput of 154 TFLOPS. As we will discuss later, the maximum throughput reported here is a lower bound on the maximum due to the throttling of clock speeds on our GPU.

4.4 Brute Force Tensor Core Throughput

We compare the performance of the two brute force TC algorithms to observe how they scale with increasing data dimensionality. Recall from Section 4.1.2 that TED-Join-Brute exceeds shared memory capacity and so we are unable to report its performance across a large range of data dimensionalities. Figure 9 compares the performance of our algorithm, FaSTED (FP16-32), to TED-Join-Brute (FP64). Regarding FaSTED, we find that the TC throughput increases as a function of d, which indicates that it is highly scalable in terms of data dimensionality. In contrast, the performance of TED-Join-Brute degrades significantly as d increases, which was also shown in the TED-Join paper [8]. At d=4096, FaSTED achieves a sustained throughput of 49% of the A100's peak throughput of 312 TFLOPS using FP16-32 TCs [18]. In contrast, TED-Join only achieves 6.8% of FP64 peak throughput when d=64, and it declines at higher dimensionalities.

To better understand why the performance of FaSTED scales well with increasing data dimensionality (d), while the performance of TED-Join-Brute degrades, we employ the Nvidia Nsight Compute profiler and present the results in Table 6. We observe that as d increases, the tensor pipe utilization metric increases for FaSTED but does not for TED-Join-Brute. Examining the percentage of shared memory bank conflicts, we observe that FaSTED is conflict-free (0%), whereas TED-Join-Brute yields ≥75% shared memory bank conflicts. TED-Join is constrained by the WMMA API, which has rigid load/store memory access patterns, which causes shared memory bank conflicts in both the TED-Join-Brute and TED-Join-Index variants.

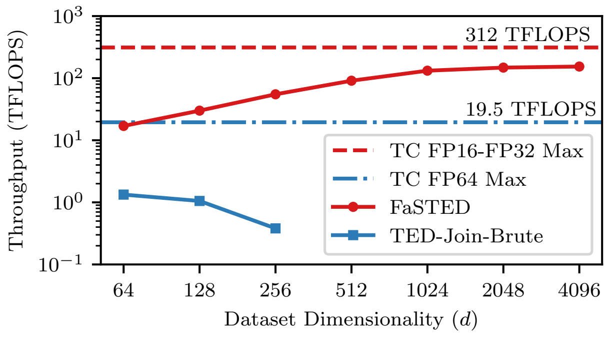


Figure 9: Comparison of derived TFLOPS (plotted on a log scale) achieved on the brute force TC algorithms, TED-Join-Brute (FP64) and FaSTED (FP16-32), as a function of data dimensionality, using the *Synth* dataset of size $|D|$=10^5. The standard deviation for all FaSTED measurements was < 0.7 TFLOPS. The peak throughput is shown for context.

Table 6: Nvidia Nsight Compute profiler results on the *Synth* dataset with $|D|$=10^5. We examine $d \in \{128, 256, 4096\}$ to illustrate how key performance metrics of the brute force TC algorithms (FaSTED and TED-Join-Brute) vary as a function of dimensionality. OOM refers to "out of shared memory".

Metric	FaSTED			TED-Join-Brute		
Dimensionality (d)	128	256	4096	128	256	4096
DRAM Throughput (%)	1.98	3.54	16.0	0.04	0.04	OOM
SMEM Throughput (%)	6.49	10.5	36.1	42.3	16.0	OOM
Bank Conflicts (%)	0.00	0.00	0.00	92.3	75.0	OOM
L2 Hit Rate (%)	89.8	89.6	84.4	98.9	98.9	OOM
TC Pipe Utilization FP16-32 (%)	10.1	17.8	64.0	N/A	N/A	OOM
TC Pipe Utilization FP64 (%)	N/A	N/A	N/A	5.75	1.99	OOM
Clock Speed (GHz)	1.37	1.40	1.12	1.40	1.41	OOM

While analyzing our results from Table 6, we discovered a discrepancy between the TC Pipe Utilization metric and our derived TFLOPS. The profiler reported 64% utilization while, by our calculations, we only achieved 49%. The explanation for this discrepancy is that at d=4096, the 250 W power budget for our PCIe A100 is exceeded, and the clock is throttled to 1.12 GHz. The implications of this finding are discussed in the conclusion.

4.5 Comparison to the SOTA on Real-World Datasets

We compare our brute force algorithm, FaSTED, to the SOTA index-supported GPU similarity search algorithms (MiSTIC, GDS-Join, and TED-Join-Index) that prune the search to reduce comparing points that are located at sufficiently large distances of each other. This affords the index-supported algorithms a performance advantage over FaSTED as it does not use an index and thus performs $|D|^2$ point comparisons.

Figure 10 plots the response time on the four real-world datasets which span dataset sizes $|D|$=60,000–10,000,000 and data dimensionalities of d=128–960 for the three selectivity values (S_s=64, S_m=128, and S_l=256), which refer to the mean number of neighbors found by each point in each dataset (see Table 4 for details). Remarkably, we find that FaSTED outperforms all of the index-supported GPU methods and make two observations.

(1) On a given dataset, the speedup of FaSTED over the reference implementations increases with increasing selectivity. This is because FaSTED is a brute force method, and so it computes the distance between all pairs of points, and so the response time of FaSTED is independent of increasing selectivity (or search radius, ϵ). In contrast, the response time of the reference implementations (MiSTIC, GDS-Join, and TED-Join-Index) increase with selectivity because the number of distance calculations performed increases with search distance.

(2) The maximum dimensionality on the real-world datasets is d=960. However, as shown in Section 4.2, a minimum dimensionality of roughly d=2048 is needed for FaSTED to reach ~150 TFLOPS. This demonstrates that even in cases where the dimensionality is insufficient to reach ~150 TFLOPS, FaSTED still outperforms the SOTA with a minimum speedup of 2.5× (over MiSTIC and GDS-Join) and a maximum of 51× over TED-Join-Index.

Table 7: The accuracy of FaSTED compared to the FP64 GDS-Join algorithm using the overlap accuracy metric (Equation 3) across the three selectivity levels for all real-world datasets.

	Sift10M	*Tiny5M*	*Cifar60K*	*Gist1M*
S_s=64	1.0	0.99998	0.99971	0.99999
S_m=128	1.0	0.99997	0.99955	0.99998
S_l=256	OOM	0.99996	0.99946	0.99997

4.6 FaSTED: Accuracy of FP16-32

FaSTED uses FP16-32, so there is greater machine rounding error compared to the FP32 or FP64 reference implementations. We examine the accuracy of FaSTED and configure GDS-Join to use FP64 values (instead of FP32 used in Figure 10) and employ it as the ground truth. We examine two measures of accuracy; the first quantifies the overlap between the points found in the result set for each point in the dataset. The second method compares the difference in the distances computed by the algorithm.

Overlap Between Result Sets: For each point in the dataset, $p_i \in D$, we compute its neighbors in FaSTED and GDS-Join, denoted as N_i^{FaSTED} and $N_i^{\text{GDS-Join}}$, respectively. We compare the neighbors for each point in the ground truth set to FaSTED and determine if the sets match. The intersection over the union of the sets for each point yields the overlap between the sets; if the intersection and union of the sets are equal, then we assign an accuracy score of 1.0 for the point, otherwise the score is <1.0. Equation 3 defines the accuracy score for an entire dataset where a score of 1.0 indicates perfect accuracy:

$$Accuracy = \frac{1}{|D|} \cdot \sum_{i=1}^{|D|} \frac{\left| N_i^{\text{FaSTED}} \cap N_i^{\text{GDS-Join}} \right|}{\left| N_i^{\text{FaSTED}} \cup N_i^{\text{GDS-Join}} \right|}. \quad (3)$$

Difference Between Computed Distances: We further our accuracy analysis by examining the difference between the computed distances for each $p_i \in D$ of the points found in the result sets of both FaSTED and GDS-Join using FP64 (the ground truth). The distance error for a single pair of points p_i and p_j is defined as

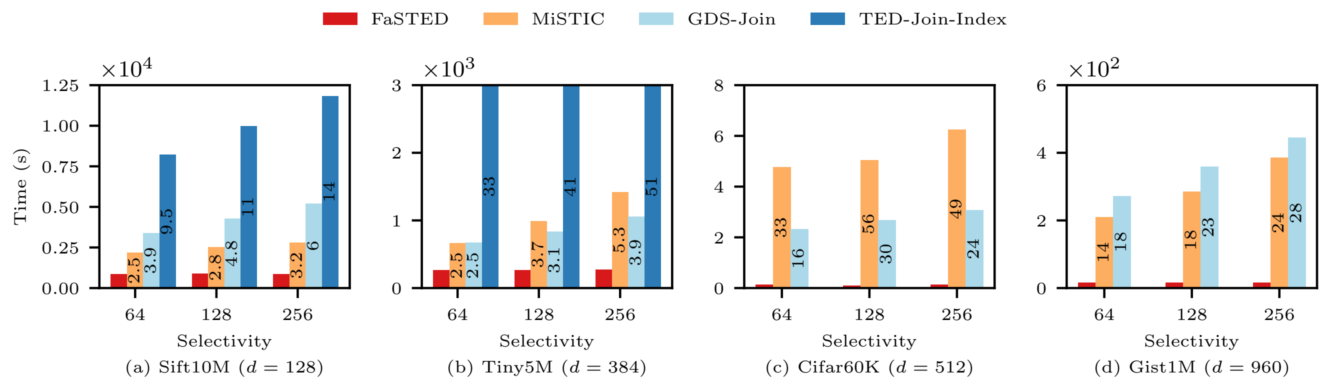


Figure 10: Comparison of FaSTED to the SOTA GPU index-supported similarity search algorithms (MiSTIC, GDS-Join, and TED-Join-Index) on four real-world datasets. The total end-to-end response time is plotted, which includes all associated overheads for each method (e.g., index construction and transferring data to/from the GPU). TED-Join-Index exceeds shared memory at d=384, so it is not shown in panels (c)–(d). The speedup of FaSTED over the reference implementations is reported inside each bar. The standard deviation of the response times for FaSTED in each plot is (a) $\sigma = 42$, (b) $\sigma = 0.11$, (c) $\sigma = 0.12$, and (d) $\sigma = 0.054$. In other words, the difference between time trial measurements is negligible on all datasets except for *Cifar60K* because the execution time was negligible on that dataset.

Table 8: The accuracy of FaSTED compared to the FP64 GDS-Join algorithm using the distance metric with the smallest selectivity level, S_s=64, for all real-world datasets.

	Sift10M	*Tiny5M*	*Cifar60K*	*Gist1M*
Mean	2.6×10^{-6}	-1.5×10^{-7}	-5.2×10^{-7}	-1.6×10^{-6}
Std. Dev.	2.4×10^{-4}	9.4×10^{-6}	3.4×10^{-5}	3.7×10^{-5}

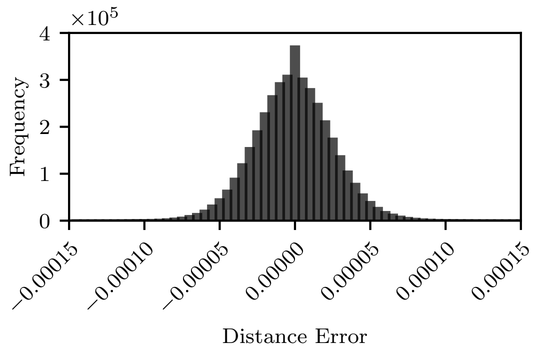


Figure 11: The distribution of the errors in the distance measurements for the *Cifar60K* dataset is shown.

$dist_{i,j}^{\text{FaSTED}} - dist_{i,j}^{\text{GDS-Join}}$. We then compute the mean and standard deviation of this error across all pairs of points that appear in both the FaSTED and GDS-Join result sets.

Table 7 presents the Overlap Between Result Sets metric across all real-world datasets, where we find that there is a negligible accuracy loss. The maximum accuracy loss occurs on *Cifar60K* ($S_l = 256$), which still achieves a very high accuracy of 99.946%.

Table 8 presents the Difference Between Computed Distances metric for all pairs of points within the ϵ corresponding to S_s=64. Figure 11 shows the distribution of the errors in the distance calculations for the *Cifar60K* dataset (the dataset with the highest error, described above). The distribution in Figure 11 and summary statistics in Table 8 show that the distances FaSTED computes have no measurable bias and have minimal error. Thus, for similarly distributed datasets that are commensurate with the dynamic range of FP16, FaSTED can compute Euclidean distances without concern for accuracy loss due to machine rounding error. We observed a slight decrease in accuracy with increasing search radius. This detail may be relevant to a reader interested in applications where large distances are of particular importance. In the case of distance similarity searches, and other algorithms that are interested in the local neighborhood of points (e.g., k nearest neighbors), the error in the distance computed is not of consequence.

5 Discussion & Conclusions

To our knowledge, we have proposed the first algorithm in the literature that computes Euclidean distances using mixed precision TCs. Similarly, others have successfully used mixed precision arithmetic in other application domains. Cui [4] used mixed precision TCs on GPUs for finite element methods, Genome-Wide Association Studies [15] employed mixed precision arithmetic on GPU TCs, Geostatistics applications [3] and plasma turbulence simulations [12] used mixed precision arithmetic on the A64FX CPU. Many of these papers demonstrate that the loss of accuracy can be recovered, and in some cases, they describe that using FP32 or FP64 may be excessive. Similarly to these other papers, we find that the accuracy loss of our mixed precision computation for distance similarity searches is negligible.

We find that our method scales well with increasing data dimensionality and only requires a small to moderate dataset size to reach our maximum throughput of roughly 150 TFLOPS. We find that compared to the SOTA GPU distance similarity search algorithms that employ an index to prune the search, our approach is significantly faster, despite performing more distance calculations. FaSTED's performance can be primarily attributed to carefully designed optimizations that enable sufficient data reuse, hide memory access latency, and eliminate shared memory bank conflicts. There are numerous algorithms that employ Euclidean distance calculations, so our algorithm will be of great utility to many application areas beyond similarity searches.

Despite the datasets that we used not being normalized to the full range of FP16 values, the lowest search accuracy is 99.946%, which is an impressive finding. It is likely that scaling the input data could further increase the accuracy of our results, and in the case where a dataset is adversely affected by conversion to FP16, it would mitigate this numerical sensitivity. Future work will investigate this research avenue.

We believe that our maximum performance of ~150 TFLOPS is an underestimate of FaSTED's potential. We observe that when our algorithm executes with sufficiently large dataset sizes and dimensionalities, the clock speed is dynamically reduced because we exceed the 250 W power budget of our PCIe A100. For example, if we were to use an SXM A100 that has a 400 W power budget, we believe that FaSTED would achieve even greater throughput. Thus, the 150 TFLOPS result reported in this paper should be considered the lower bound on the performance of FaSTED when using $|D|\geq46416$ and $d\geq2048$.

Given that we have significantly utilized the capabilities of modern GPU hardware, future work should focus on efficiency optimizations, such as incorporating an indexing data structure to prune the search, which would further improve performance. However, such an algorithm will have to be carefully designed to ensure that TC performance is not severely diminished.

While we have reached a performance ceiling for FaSTED, we found that the FP64 TC algorithm in the literature, TED-Join, is only reaching 6.8% of peak performance. We hope that the detailed rationales, description, and study of our optimizations will enable them to be applied to redesigning TED-Join and other GPU-accelerated algorithms in the literature to realize significant performance improvements.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 2042155. We thank Benoit Gallet for updating his repository so we could execute experiments with TED-Join and Brian Donnelly for kindly running experiments with MiSTIC.

References

[1] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symp. on Security and Privacy.* 353–367.

[2] Thomas Bottesch, Thomas Bühler, and Markus Kächele. 2016. Speeding up k-means by approximating Euclidean distances via block vectors. In *Intl. Conf. on Machine Learning.* PMLR, 2578–2586.

[3] Qinglei Cao, Sameh Abdulah, Rabab Alomairy, Yu Pei, Pratik Nag, George Bosilca, Jack Dongarra, Marc G Genton, David E Keyes, Hatem Ltaief, et al. 2022. Reshaping geostatistical modeling and prediction for extreme-scale environmental applications. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 1–12.

[4] Cu Cui. 2024. Acceleration of tensor-product operations with tensor cores. *ACM Transactions on Parallel Computing* 11, 4 (2024), 1–24.

[5] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating reduction and scan using tensor core units. In *Proc. of the ACM Intl. Conf. on Supercomputing.* 46–57.

[6] Brian Donnelly and Michael Gowanlock. 2024. Multi-Space Tree with Incremental Construction for GPU-Accelerated Range Queries. In *2024 IEEE 31st Intl. Conf. on High Performance Computing, Data, and Analytics.* 132–142.

[7] M. Fasi, N.J. Higham, M. Mikaitis, and S. Pranesh. 2021. Numerical behavior of NVIDIA tensor cores. *PeerJ Computer Science* 7 (2021), e330.

[8] Benoit Gallet and Michael Gowanlock. 2022. Leveraging GPU Tensor Cores for Double Precision Euclidean Distance Calculations. In *2022 IEEE 29th Intl. Conf. on High Performance Computing, Data, and Analytics.* 135–144.

[9] Michael Gowanlock, Benoit Gallet, and Brian Donnelly. 2023. Optimization and Comparison of Coordinate-and Metric-Based Indexes on GPUs for Distance Similarity Searches. In *Intl. Conf. on Computational Science.* Springer, 357–364.

[10] Michael Gowanlock and Ben Karsin. 2019. GPU-accelerated Similarity Self-join for Multi-dimensional Data. In *Proc. of the 15th Intl. Workshop on Data Management on New Hardware.* 1–9.

[11] Qiang Huang and Anthony KH Tung. 2023. Lightweight-yet-efficient: Revitalizing ball-tree for point-to-hyperplane nearest neighbor search. In *IEEE 39th Intl. Conf. on Data Engineering.* 436–449.

[12] Yasuhiro Idomura, Takuya Ina, Yussuf Ali, and Toshiyuki Imamura. 2020. Acceleration of fusion plasma turbulence simulations using the mixed-precision communication-avoiding Krylov method. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 1–13.

[13] Zhuoran Ji and Cho-Li Wang. 2021. Accelerating DBSCAN algorithm with AI chips for large datasets. In *Proc. of the 50th Intl. Conf. on Parallel Processing.* 1–11.

[14] Binrui Li, Shenggan Cheng, and James Lin. 2021. tcFFT: A Fast Half-Precision FFT Library for NVIDIA Tensor Cores. In *2021 IEEE Intl. Conf. on Cluster Computing.* 1–11.

[15] Hatem Ltaief, Rabab Alomairy, Qinglei Cao, Jie Ren, Lotfi Slim, Thorsten Kurth, Benedikt Dorschner, Salim Bougouffa, Rached Abdelkhalak, and David E Keyes. 2024. Toward capturing genetic epistasis from multivariate genome-wide association studies using mixed-precision kernel ridge regression. In *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis.* IEEE, 1–12.

[16] Jiwen Lu, Venice Erin Liong, and Jie Zhou. 2017. Deep hashing for scalable image search. *IEEE Transactions on Image Processing* 26, 5 (2017), 2352–2367.

[17] Nvidia. 2025. Double Precision GEMM Using the WMMA API. https://github.com/NVIDIA/cuda-samples Accessed Apr. 1, 2025.

[18] Nvidia. 2025. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf Accessed Apr. 1, 2025.

[19] Chaitanya Ryali, John Hopfield, Leopold Grinberg, and Dmitry Krotov. 2020. Bio-inspired hashing for unsupervised similarity search. In *Intl. Conf. on Machine Learning.* PMLR, 8295–8306.

[20] Hanan Samet. 2008. K-Nearest Neighbor Finding Using MaxNearestDist. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30, 2 (2008), 243–252.

[21] Lipeng Wan, Fangyu Zheng, Guang Fan, Rong Wei, Lili Gao, Yuewu Wang, Jingqiang Lin, and Jiankuo Dong. 2022. A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator. In *European Symp. on Research in Computer Security.* Springer, 514–534.

[22] Bei Xie, Jiaohua Qin, Xuyu Xiang, Hao Li, and Lili Pan. 2018. An Image Retrieval Algorithm Based on Gist and Sift Features. *Intl. Journal of Network Security* 20, 4 (2018), 609–616.

[23] Liang Zheng, Yi Yang, and Qi Tian. 2017. SIFT meets CNN: A decade survey of instance retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, 5 (2017), 1224–1244.

[24] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. 2012. A survey on unsupervised outlier detection in high-dimensional numerical data. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 5, 5 (2012), 363–387.