Scalable Hamming Distance Computation Using Accelerated Matrix Transformations

Rabab Alomairy[1,4], Qinglei Cao[2,5], Hatem Ltaief[3,6], David Keyes[3,7], and Alan Edelman[1,8]

[1]Computer Science & Artificial Intelligence Laboratory,
Massachusetts Institute of Technology, USA.
[2]Department of Computer Science, Saint Louis University, USA.
[3]Division of Computer, Electrical, and Mathematical Sciences and Engineering,
King Abdullah University of Science and Technology, KSA.
[4]*rabab.alomairy@mit.edu* [5]*qinglei.cao@slu.edu* [6]*hatem.ktaief@kaust.edu.sa*
[7]*david.keyes@kaust.edu.sa* [8]*edelman@mit.edu*

***Abstract*—The Hamming distance, a fundamental measure of dissimilarity between data points, plays a crucial role in various fields, including error detection, machine learning, and genomic sequence alignment, where it is commonly used for identifying mismatches in nucleotide or protein sequences. This work introduces two implementations for computing Hamming distances for sequence alignment: synchronous and asynchronous matrix-based approaches. While most existing implementations rely on vector-based methods due to their simplicity and ease of use, they are not efficient for large-scale data. Our work focuses on enhancing performance by introducing matrix-based implementations that significantly improve computational efficiency and scalability. Our asynchronous implementation showcases Julia for sequential task flow and PaRSEC for parameterized task graph execution models on homogeneous and heterogeneous architectures. CPU computations use INT8 GEMM from oneMKL, while GPU implementations employ Tensor/Matrix Core INT8 GEMM from cuBLAS/hipBLAS and 1-bit TensorOps GEMM capabilities from CUTLASS. For constructing bitmask matrices on GPUs, we develop both a naive CUDA implementation using global memory and an optimized implementation utilizing shared memory at the warp level, with the optimized version achieving a 5X speedup over the naive approach. The results demonstrate significant performance improvements, with the asynchronous matrix-based implementation achieving up to 284X speedup over the vector-based approach on CPUs, while the asynchronous GPU-enabled implementation on A100 GPUs delivers a 15X speedup compared to the CPU matrix-based approach and a three orders of magnitude improvement over the CPU vector-based approach. Furthermore, the asynchronous implementation of PaRSEC scales well on up to 256 nodes of `Summit` and `Frontier`. These advancements highlight the scalability and efficiency of matrix-based Hamming distance computation, leveraging GPU acceleration and advanced asynchronous execution, paving the way forward for large-scale genomic sequence alignment and data analysis.**

***Index Terms*—Hamming Distance, Matrix Multiply, Genomics, 1-bit Integer, Julia, PaRSEC, CUTLASS, Mixed-Precision, GPU**

I. Introduction

The Hamming distance, a fundamental measure of dissimilarity between strings of equal length, plays a critical role in various applications, including error detection and correction, bioinformatics, machine learning, and genomic sequence alignment. In sequence alignment, the Hamming distance quantifies mismatches in genomic and proteomic data, enabling researchers to identify mutations, trace evolutionary relationships, and analyze biological sequences efficiently [30], [24]. However, as biological and computational data continue to grow exponentially, traditional methods for computing Hamming distances struggle to scale, necessitating optimized approaches that leverage advanced algorithms and hardware accelerators [29].

Conventional vector-based approaches, while intuitive and easy to implement, are inherently limited in their scalability due to sequential operations and inefficient memory usage. To address these challenges, this work introduces synchronous and asynchronous matrix-based approaches. By reformulating the problem in terms of matrix operations, we harness the inherent parallelism and computational efficiency of modern multi-core CPUs and multi-GPU architectures.

Several studies have explored methods to accelerate similarity measures to overcome the computational challenges associated with large-scale data analysis. For instance, [6] and [21] propose scalable methods for efficiently computing Jaccard similarity. In the context of Hamming distance, [32] introduces a GPU-based approach that accelerates both Hamming and Levenshtein distance computations. Furthermore, FPGA-based methods for accelerating Hamming distance computations have been explored in [18] and [29]. In this paper, we leverage Julia for asynchronous sequential task flow (STF) [3] and PaRSEC as an asynchronous parameterized task graph (PTG) [9] framework, demonstrating scalability and efficiency across CPU and GPU architectures. Additionally, we explore the CUTLASS (CUDA Templates for Linear Algebra Subroutines and Solvers) C++ library for synchronous matrix-based transformations.

In this work, the matrix-based Hamming distance transforms the traditional element-wise comparison into efficient matrix operations. It begins by constructing binary matrices and their complements to identify matches and mismatches for each unique value. The computation proceeds iteratively, using matrix multiplication to aggregate results for each unique value. The asynchronous implementation extends this approach by leveraging task-based runtime systems to overlap computational phases, thereby improving efficiency. Additionally, it avoids repetitive computations by exploiting the symmetry of the resulting distance matrix, skipping the redundant calculation of both upper or lower triangular parts.

For matrix multiplication on CPUs, INT8 GEMM operations from oneMKL are utilized. On CPUs, transitioning from the vector-based approach to the synchronous matrix-based implementation, we achieve a 280X speedup. For GPU acceleration, we utilize cuBLAS Tensor Core and hipBLAS Matrix Core operations, as well as NVIDIA's CUTLASS library. cuBLAS/hipBLAS provides highly optimized GEMM kernels for Tensor/Matrix Core operations, delivering high performance for matrix computations with various precision formats. CUTLASS complements this by offering a rich library of GEMM kernels that support a wide range of precisions, including 1-bit, 4-bit, and 8-bit integer, as well as FP16, FP32, and FP64 floating-point formats. The flexibility of CUTLASS enables us to fully exploit GPU Tensor Core capabilities for both synchronous and asynchronous implementations, achieving a 15X speedup compared to the CPU matrix-based approach and a three orders of magnitude improvement over the CPU vector-based approach. Furthermore, for constructing bitmask matrices on GPUs, we implement a CUDA naive approach using global memory and an optimized approach leveraging shared memory at the warp level, achieving a 5X speedup.

Our main contributions are as follows. First, we redesign the Hamming distance algorithm and transform its traditional element-wise comparison into efficient matrix operations. Second, we implement the algorithm leveraging CUTLASS for optimized GPU performance and using the Julia programming language for high user-productivity and the PaRSEC runtime system for targeting distributed-memory systems. Third, we conduct extensive large-scale experiments on homogeneous and heterogeneous architectures on leading HPC systems, equipped with NVIDIA and AMD GPUs, achieving up to three orders of magnitude improvement compared to the reference implementations.

This paper is organized as follows. Section 2 discusses related work, highlighting existing methods for Hamming distance computation and different task-based runtime systems. Section 3 provides background on Hamming distance, sequence alignment, and the computational frameworks used. Section 4 details the proposed algorithms for vector-based, synchronous matrix-based, and asynchronous matrix-based implementations. Section 5 outlines the implementation specifics for CPU and GPU, including the use of Julia, PaRSEC, and CUTLASS. Section 6 presents experimental results demonstrating the performance improvements and scalability of our approaches. Finally, Section 7 concludes the paper and discusses potential directions for future research.

II. Related Work

A. Similarity measure

Most available implementations of similarity measures rely on vectorized operations on the CPU for their computations. However, in high-performance computing (HPC) and large-scale applications, there is a growing need to optimize these operations and offload computations to hardware accelerators. The authors in [6], [21] propose scalable methods for efficiently computing Jaccard similarity on distributed-memory systems using bitmask operations. Similarly, the study in [32] presents a GPU-based approach that utilizes non-deterministic finite automata (NFAs) to accelerate Hamming and Levenshtein distance computations. FPGA-based methods have also been explored, such as in [29], which introduces a scalable, streaming-based system architecture to accelerate binary string comparisons using FPGAs. In addition, [18] focuses on genomic sequence alignment, leveraging FPGA-based XOR and shift bit operations for efficient computation.

This paper advances the state-of-the-art by accelerating Hamming distance computations for identifying mismatches in nucleotide sequences using matrix transformations and task-based runtime systems, incorporating Tensor Core operations with bit-masking. To our knowledge, this is the first time to optimize Hamming distance computation down at the bit level on GPUs, utilizing Tensor/Matrix Core operations while deploying GPU-resident matrix construction and bit-masking.

B. Runtime Systems

In the last decade, several engines have emerged to support task-based programming models, enabling scientific applications to scale on today's fastest supercomputers [12], [2]. OpenMP [28], a prominent standard for shared-memory parallelism, incorporates task-based capabilities that enable dynamic and flexible task execution. By combining directives with library functions, OpenMP empowers developers to effectively manage concurrency through cooperation between the compiler and runtime environment. Extending OpenMP's functionality, OmpSs [20] introduces mechanisms for supporting heterogeneous platforms, asynchronous execution, and task dependencies, thereby offering enhanced adaptability. Similarly, the COMP Superscalar (COMPSs) framework [25] provides a user-friendly interface and runtime system designed to exploit application parallelism across distributed systems. StarPU [4] addresses the needs of distributed, heterogeneous multicore platforms by allowing kernel-specific annotations and ensuring optimal task scheduling and data handling through its runtime layer. The HPX (High-Performance ParalleX) runtime system [22], developed using the ParalleX execution model, is tailored for scalable, parallel, and distributed HPC applications using modern C++ features. Finally, Legion [5] presents a novel approach tailored for distributed and heterogeneous environments. Its programming model focuses on decoupling the specification of tasks and data from the hardware mapping process, enabling automatic parallelism discovery, improved data locality, and better scalability for complex workloads.

In this work, we focus on Julia [7] and PaRSEC [9], [11]. We demonstrate how Julia offers a high-level, expressive programming environment that facilitates algorithms with both flexibility and ease of use, and how PaRSEC serves as a task-based runtime system to enable high-performance computations on large-scale platforms.

III. Background

This section provides an overview of the programming models and libraries utilized in this study, comprising two asynchronous task-based programming models and one synchronous approach. The first is Julia, paired with the Dagger.jl package, which implements the Sequential Task Flow (STF) programming model to manage task dependencies and execution efficiently. The second is PaRSEC, a runtime system for dynamic and scalable task scheduling on distributed multi-core heterogeneous systems. Lastly, we incorporate CUTLASS, an NVIDIA library specifically designed for GPU-accelerated linear algebra computations, offering a wide range of highly optimized GEMM kernels.

A. Julia

Julia [7], as a high-performance programming language, combines the expressiveness of a dynamic language with the speed of compiled code, making it an ideal choice for scientific computing [33]. Built on LLVM, Julia provides powerful type inference, just-in-time compilation, and multiple dispatch, enabling users to develop optimized algorithms without requiring deep expertise in low-level programming. Julia manages STFs via Dagger.jl [31] on CPUs and GPUs, providing an accessible and high-performance implementation for domain applications, especially for non-expert users. Dagger is a powerful Julia package designed to simplify parallelization. It allows users to write Dagger-based applications in a manner as close as possible to sequential code. In this model, users focus on (i) identifying tasks for asynchronous execution, and (ii) annotating these tasks with data direction. Dagger analyzes this information and builds a task execution flow represented as a Directed Acyclic Graph (DAG), where nodes are computational tasks and edges are data dependencies. The DAG-based runtime system then effectively exposes concurrency, reduces synchronization points, ensures load balancing, shortens the critical path, and abstracts hardware complexity, similar to out-of-order task scheduling on superscalar processors.

B. PaRSEC

PaRSEC [9], [11] is a versatile framework for architecture-aware scheduling and micro-task management on distributed, multi-core, heterogeneous architectures. It encompasses a runtime system; various programming interfaces for expressing task graphs, which are employed in both dense matrix operations [8], [15] and irregular applications (e.g., mixed-precision algorithms [1], [13], [26], low-rank approximation [16], and sparse computations [14]); support libraries to assist with transitioning algorithms from legacy programming paradigms; and performance analysis tools [17] to aid developers in optimizing applications for large-scale hybrid environments. PaRSEC's interaction with users is facilitated through a suite of Domain-Specific Languages (DSLs), including the Parameterized Task Graph (PTG) [19], Template Task Graph (TTG) [10], and Dynamic Task Discovery (DTD) [23]. These DSLs offer significant flexibility, enabling scientists to intuitively express complex algorithms. PTG, in particular, represents the entire DAG in a compressed and parameterized form, enabling the identification and utilization of collective communications and the decoupling of data distribution from task operations. This level of abstraction forms the technical foundation of our research. Consequently, PTG is employed herein.

C. CUTLASS

CUTLASS [27] is a high-performance library developed by NVIDIA for accelerating dense linear algebra operations on NVIDIA GPUs. It leverages CUDA Tensor Core operations to achieve optimized performance for matrix multiplications, including mixed-precision and low-precision computations. CUTLASS is designed with a modular and template-based architecture, enabling developers to customize operations for various data types and computational needs. A key strength of CUTLASS lies in its support for low-bit TensorOps, including 1-bit operations. This allows efficient binary matrix operations, such as those required for Hamming distance computations, where the matching mask matrix and its complement are efficiently represented at the 1-bit level. By explicitly constructing these matrices and performing operations at a low precision, CUTLASS maximizes throughput while utilizing the full computational capabilities of modern NVIDIA GPUs.

IV. Algorithm

This section provides details of two approaches for computing Hamming distances: the vector-based algorithm and the matrix-based algorithm. The vector-based method processes individual row pairs iteratively, focusing on element-wise comparisons, while the matrix-based approach leverages matrix operations to exploit computational efficiency and parallelism. The methods are compared in terms of computational complexity and suitability for large-scale datasets.

A. Vector-Based Hamming Distance Computation

The typical vector-based Hamming distance algorithm computes the number of differing elements between all pairs of rows in a given matrix as in algorithm 1. Starting with an $n \times m$ input matrix, where n represents the number of rows and m represents the number of columns, the algorithm initializes an $n \times n$ matrix to store the results. For each pair of rows i and j, it compares their corresponding elements column by column, incrementing a distance counter whenever a mismatch is found. This computed distance is then stored symmetrically in the result matrix, as the Hamming distance between row i and row j is identical to the distance between row j and row i. The final output is a fully populated distance matrix representing the pairwise Hamming distances for all rows of the input matrix.

Algorithm 1 Pairwise Hamming distance computation.

Input: Matrix X of size $n \times m$
Output: Matrix H of size $n \times n$ where $H[i][j]$ is the Hamming distance between row i and row j of X
for $i = 0$ to $n - 1$ **do**
  for $j = i + 1$ to $n - 1$ **do**
    $distance \leftarrow 0$ {Initialize distance counter for row pair (i, j)}
    for $k = 0$ to $m - 1$ **do**
      if $M[i][k] \neq M[j][k]$ **then**
        $distance \leftarrow distance + 1$
      end if
    end for
    $H[i][j] \leftarrow distance$ {Store the computed distance}
    $H[j][i] \leftarrow distance$ {Use symmetry property}
  end for
end for
return H // Return matrix H containing Hamming distances

B. *Matrix-Based Hamming Distance Computation*

The matrix-based Hamming distance Algorithm 2 transforms traditional element-wise comparisons into efficient matrix operations, making it highly scalable for large datasets. The algorithm identifies unique values in the input matrix X and constructs binary (matching bit-mask) matrices B to capture matches and their complements B^c to capture mismatches. It computes pairwise distances iteratively by aggregating matrix multiplications involving B and B^c, updating the distance matrix H at each step. This matrix transformation approach is highly compatible with modern accelerators, which are specifically designed to support highly optimized matrix multiplication kernels, a key component in many AI workloads.

Figure 1 demonstrates the matrix-based Hamming distance algorithm with an example focused on genomic sequence alignment, as it is essential for mutation analysis, evolutionary studies, and other bioinformatics applications. The input matrix X represents a set of DNA sequences, where each row corresponds to a sequence and each column represents a nucleotide (A, T, G, C). The algorithm begins by identifying the unique nucleotides in the input matrix. For each unique nucleotide, a matching bitmask matrix B is constructed to indicate where the nucleotide matches, and a complement bitmask matrix B^c identifies mismatches. The Hamming distance matrix H is updated iteratively by aggregating results from matrix multiplications involving B and B^c. Each step of the process is illustrated, with intermediate results showing how contributions from individual nucleotides are combined to form the final distance matrix. The final matrix H accurately captures the pairwise Hamming distances between DNA sequences, validating the correctness of the algorithm.

Algorithm 2 Matrix-based Hamming distance calculation.

Input: Matrix X of size $n \times m$
Output: Matrix H of size $n \times n$
uniqs $\leftarrow$ unique(X) // Get unique values in X
$H \leftarrow$ zeros(m, n) // Initialize H as a matrix of zeros
for each uni **in** uniqs **do**
  B, $B^c \leftarrow (X ==$ uni$), (X! =$ uni$)$ // Calculate a binary matrix and its complement for current unique value
  $H \leftarrow B^c \times B^T + H$ // Calculate matrix H
end for
Return H // Return matrix H containing Hamming distances

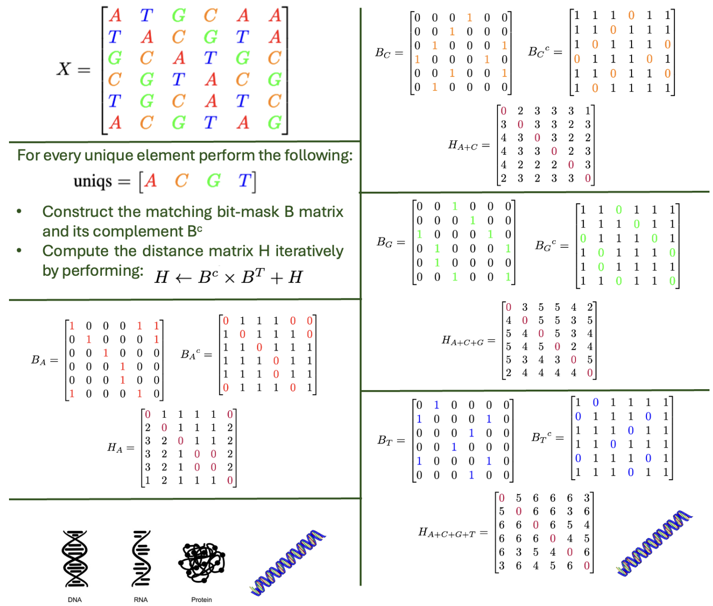

Fig. 1: Matrix-based Hamming distance.

This algorithm is extended in a task-based programming model to utilize asynchronous fine-grained tasks to compute only the triangular part of the matrix, effectively exploiting its symmetry. By decomposing the matrix into tiles, computational tasks are launched for the relevant triangular region, avoiding redundant operations. Each tile corresponds to a submatrix, and GEMM kernels are used to compute the contributions from these tiles iteratively. This approach allows the binary matrices to be constructed on-the-fly during tile computation, eliminating the need for explicit preconstruction. This not only streamlines the computation process but also reduces the memory footprint, enabling large datasets to reside higher in the memory hierarchy. We provide implementations of this asynchronous approach using the Dagger.jl package in Julia and the PaRSEC runtime system. These frameworks enable efficient task scheduling and execution, ensuring scalability across heterogeneous computing architectures.

C. *Computational Efficiency of Hamming Distance*

When comparing the computational efficiency of the vector-based, synchronous matrix multiplication-based, and asynchronous symmetry-based approaches for Hamming distance computation, the number of operations highlights significant differences. The vector-based approach iterates through all possible row pairs (i, j) and computes the Hamming distance by comparing each element, resulting in $O(n^2 \cdot m)$ operations, where n is the number of rows and m is the number of columns. This approach is conceptually straightforward but computationally expensive, primarily due to repeated element-wise operations that exhibit low arithmetic intensity.

Matrix-based approaches, on the other hand, leverage binary matrices and compute distances through matrix multiplication ($B^c \cdot B^T$), which requires $2 \cdot p \cdot n^2 \cdot m$ operations, where p is the number of unique values in the matrix. This approach leverages level-3 BLAS GEMM operations and takes full advantage of Tensor/Matrix Cores available on modern NVIDIA/AMD GPUs, ensuring efficient and high-performance computations. While more efficient for large n, this method redundantly computes the upper and lower triangular parts of the resulting symmetric matrix. The asynchronous approach improves upon exploiting the symmetry of the distance matrix, reducing the number of operations to $p \cdot n^2 \cdot m$. This reduction cuts the operation count by half compared to standard matrix multiplication, making the symmetry-based approach the most efficient choice when working with symmetric output.

V. Algorithm Correctness

In this section, we provide a formal proof to establish the correctness of the proposed matrix-based Hamming distance algorithm.

A. Problem Definition

Given a matrix X of size $n \times m$, where each row represents a sequence (e.g., DNA, RNA, or any categorical sequence), the Hamming distance between two rows i and j is defined as:

$$H(i,j) = \sum_{k=1}^{m} \mathbb{1}(X_{ik} \neq X_{jk}) \tag{1}$$

where $\mathbb{1}(\cdot)$ is the indicator function, which returns 1 if the condition is true and 0 otherwise.

Our goal is to prove that the algorithm computes this function efficiently using matrix operations.

B. Reformulation Using Binary Matrices

Instead of direct pairwise comparisons, we construct matching bit-mask matrices for each unique symbol in X.

C. Matching Bit-Mask Matrices

For every unique symbol uni appearing in X, define a binary mask $B^{(\text{uni})}$, where:

$$B_{ik}^{(\text{uni})} = \mathbb{1}(X_{ik} = \text{uni}) \tag{2}$$

This matrix has 1s wherever the symbol appears in X and 0s elsewhere.

Similarly, define the complement matrix $(B^c)^{(\text{uni})}$:

$$(B^c)_{ik}^{(\text{uni})} = 1 - B_{ik}^{(\text{uni})} = \mathbb{1}(X_{ik} \neq \text{uni}) \tag{3}$$

which has 1s where the symbol is absent and 0s where it is present.

D. Formulating the Hamming Distance Calculation

We initialize a zero matrix H of size $n \times n$:

$$H = 0_{n \times n} \tag{4}$$

For each unique symbol uni, the algorithm updates H using the formula:

$$H \leftarrow H + (B^c)^{(\text{uni})} \cdot (B^{(\text{uni})})^T \tag{5}$$

Expanding this formula for each pair of rows i, j:

$$H(i,j) \leftarrow H(i,j) + \sum_{k=1}^{m} \mathbb{1}(X_{ik} \neq \text{uni}) \cdot \mathbb{1}(X_{jk} = \text{uni}) \tag{6}$$

This counts how many times row i has a different symbol from row j at position k due to the current unique symbol.

E. Summing Over All Unique Symbols

Since every element in X belongs to exactly one unique symbol, summing over all unique symbols ensures that we count every position where two rows differ:

$$H(i,j) = \sum_{\text{uni} \in X} \sum_{k=1}^{m} \mathbb{1}(X_{ik} \neq \text{uni}) \cdot \mathbb{1}(X_{jk} = \text{uni}) \tag{7}$$

Rearranging the summation:

$$H(i,j) = \sum_{k=1}^{m} \sum_{\text{uni} \in X} \mathbb{1}(X_{ik} \neq \text{uni}) \cdot \mathbb{1}(X_{jk} = \text{uni}) \tag{8}$$

Since exactly one unique symbol satisfies $X_{jk} = \text{uni}$, the inner sum reduces to:

$$H(i,j) = \sum_{k=1}^{m} \mathbb{1}(X_{ik} \neq X_{jk}) \tag{9}$$

which is precisely the definition of the Hamming distance.

VI. Implementation Details

This section details the implementation of the matrix-based Hamming distance algorithm.

A. Using Dagger.jl

The implementation in Table I of Hamming distance leverages Dagger.jl, a task-based parallel programming framework in Julia, to enable efficient and asynchronous computation. Designed with generality in mind, it takes advantage of Julia's powerful type inference capabilities to support any numeric data type, ensuring flexibility and adaptability for diverse applications. The approach partitions the matrices A and H into tiles manageable for parallel processing. For each unique value in a specified set, the algorithm constructs binary matrices on-the-fly and performs GEMM operations, specifically using the `cblas_gemm_s8u8s32` routine from OneMKL. Furthermore, Julia's multiple dispatch feature is employed to overload the multiplication operation, providing

```
function Binary_GEMM(A1::AbstractMatrix{T}, A2::AbstractMatrix{T}, H::AbstractMatrix{T}, unique) where
    {T<:Number}

    m, k = size(A1) # Get the size of matrix A1 (rows and columns)
    m, n = size(H)  # Get the size of matrix H (rows and columns)
    Bc = ones(T, m, n) - (A1 .== unique) # Compute the complement of the mask for matrix A1
    B = A2 .== unique # Create a mask for matrix A2
    H += Bc * B` # Multiply and update using BLAS-level operations (e.g., cblas_gemm_s8u8s32)
end

# Function to compute the Hamming distance asynchronously using Dagger.jl
# A and H are distributed arrays (DArray), and `uniques_array` contains unique values for masking.
function HammingDistanceAsync(A::DArray{T,2}, H::DArray{T,2},  uniques_array) where {T<:Number}

    Ac = parent(A).chunks # Access the chunks of the distributed array A
    Hc = parent(H).chunks # Access the chunks of the distributed array H
    Amt, Ant = size(Ac) # Get the dimensions of the chunked representation of A
    Cmt, Cnt = size(Hc) # Get the dimensions of the chunked representation of H

    Dagger.spawn_datadeps() do # Use Dagger.jl to handle asynchronous data dependencies
        for unique in uniques_array[1:end] # Iterate over unique values
            for m in range(1, Cmt) # Iterate over chunk rows of H
                for n in range(1, m) # Iterate over lower triangular chunks of H
                    for k in range(1, Amt) # Iterate over chunk columns of A
                        # Spawn asynchronous tasks to compute BitMask_GEMM for each chunk
                        Dagger.@spawn Binary_GEMM(In(Ac[m, k]), In(Ac[n, k]), InOut(Hc[m, n]), unique)

                    end end end end end
end
```

TABLE I: Julia Dagger asynchronous matrix-based implementation.

a unified interface. On architectures that do not support this specific GEMM variant, the implementation seamlessly falls back to existing GEMM operations available in Julia, ensuring compatibility across different platforms. Using Dagger.jl, tasks are dynamically spawned for each chunk combination within the triangular part of the matrix, respecting data dependencies. The scheduler executes these tasks according to the DAG, optimizing parallelism and ensuring efficient task execution.

B. Using PaRSEC

Table II provides a code snippet defining a PaRSEC's PTG-based implementation for matrix-based Hamming distance computations, executed for each uni in Algorithm 2. The implementation defines two task classes, `HAMM` and `READ_X`, which operate on tiled matrices. The `HAMM` task class performs the core computation of the Hamming distance, while `READ_X` handles the retrieval of input data from matrix X. Dependencies between tasks are expressed through data flows, ensuring proper synchronization and efficient data access across tasks. The datatypes of the flows are defined based on the associated matrices: flows related to X use `INT8`, while flows associated with H use `INT32`. The task body is designed to execute on either GPUs (NVIDIA or AMD) or CPUs, providing flexibility in deployment. The body's precision is not inherently bounded; however, in this matrix-based Hamming distance computation, we adopt specific precision formats based on the underlying hardware and libraries used. Specifically: (1) B and B^c are represented in `INT8`, while H is stored in `INT32`, for computations executed on CPUs, NVIDIA GPUs (using cuBLAS), and AMD GPUs; (2) B and B^c are represented in 1-bit precision, while H remains in `INT32`, for computations leveraging `CUTLASS`, as detailed in the following section. This design enables asynchronous execution and fine-grained parallelism, offering unprecedented flexibility and efficiency on multi-core heterogeneous architectures.

C. Enhancing Hamming Distance using CUTLASS

Here, we describe the matrix-based Hamming distance calculation, utilizing a custom CUDA kernel to construct efficient matching bitmask matrices and leveraging the bit-level GEMM Tensor Core operations provided by the CUTLASS library.

1) Matching and Complement Mask Matrix: CUTLASS requires matrices of type `uint1b_t` (1-bit integers) to be packed as multiples of bytes. Since each `uint1b_t` represents a single bit, eight such bits must be packed together to form a single byte. This packing ensures efficient memory access and alignment when performing operations on Tensor Cores, which are optimized for such compact data layouts. To construct this matching bit-mask matrix and its complement we provided incremental custom CUDA kernels. The first one directly accesses global memory for each element without leveraging shared memory or other caching mechanisms. In the second one, we optimize this step to maximize GPU occupancy utilizing shared memory for bit aggregation.

Table III shows the implementation details of the naive approach. The kernel initializes binary matrices B and B^c by mapping elements of the input matrix to their corresponding 1-bit positions in memory. Each CUDA thread processes one element of the input matrix, identified through its global index. From this index, the byte and bit positions within the binary

```
// Task class for hamming distance computations
HAMM(m, n, k)

// Execution space
m = 0 .. descH->mt-1   // descH: data descriptor for H; mt: number of tiles in the row dimension
n = 0 .. m             // Only lower
k = 0 .. descX->mt-1   // descX: data descriptor for X

// Parallel partitioning
: descH(m, n)

// Flows
READ X <-  X READ_X(k, m)                              [ type_remote = FULL_INT8 ]
READ Y <-  X READ_X(k, n)                              [ type_remote = FULL_INT8 ]
RW   H <- (k == 0)              ? descH(m, n)
       <- (k != 0)              ? H HAMM( m, n, k-1 )   [ type_remote = FULL_INT32 ]
       -> (k == (descX->mt-1)) ? descH(m, n)
       -> (k != (descX->mt-1)) ? H HAMM( m, n, k+1 )   [ type_remote = FULL_INT32 ]

BODY [ type = CUDA / HIP / CPU ]   // List the BODY type; there can be multiple of them
{
  Computations on Nvidia GPUs / AMD GPUS / CPUs
}

// Task class for reading X, ensuring proper data retrieval across distributed tiles
READ_X(k, m)

k = 0 .. descX->mt-1   // mt: number of tiles in the row dimension
m = 0 .. descX->nt-1   // nt: number of tiles in the column dimension

: descX(k, m)

READ X <- descX(k, m)
       -> X GEMM(m, 0 .. m, k)             [ type_remote = FULL_INT8 ]
       -> Y GEMM(m .. descH->mt-1, m, k)   [ type_remote = FULL_INT8 ]

BODY [ type = CPU ]   // Specify the BODY type
{
  // Nothing
}
```

TABLE II: PaRSEC's async-matrix-based implementation using PTG. Computations include $B^c \leftarrow X$, $B \leftarrow Y$ and $H \leftarrow B^c \times B^T + H$. "0 .. m" represents a loop from 0 to m (inclusive) with an incremental of 1, introducing a broadcast operation.

matrices are determined, where the byte index is $\text{idx}/8$, and the bit index within the byte is $\text{idx}\%8$.

The kernel then evaluates the value of the current matrix element against a unique value. If the matrix value matches the unique value, a corresponding bit in the binary matrix B is set. If it does not match, the bit in the complement matrix B^c is set. To achieve this, the kernel constructs bit masks using bitwise shift operations. Specifically, the bit mask for B is generated as $1U \ll \text{bit\_position}$ if the condition matrix[idx] == target_value holds `true`, and similarly for B^c when the condition is `false`. The bit mask shifts the value 1 to the appropriate position within the byte, allowing precise control over individual bits. To ensure thread safety when updating the binary matrices in global memory, the kernel uses `atomicOR` operations. These operations atomically combine the newly generated bit masks with the existing values in the memory, preventing race conditions when multiple threads attempt to update the same byte simultaneously. While this approach works as intended, it is not optimized due to direct global memory accesses coupled with atomic operations. Without leveraging shared memory, this method incurs significant memory latency and leads to underutilization of the GPU's computational resources.

Table IV demonstrates an optimized kernel for matrix construction that leverages shared memory as an intermediate buffer to reduce global memory access overhead. Each thread calculates a global index to determine the byte and bit positions within the binary matrices as before. Instead of directly updating global memory, shared memory is dynamically allocated for both matrices, with each warp having dedicated regions to store intermediate results. The shared memory index for `tensor_b` is calculated as shared_index_b $=$ $\text{threadIdx.x}/32$, ensuring all threads in a warp share the same memory region. For `tensor_bc`, an offset equal to the number of warps $\text{blockDim.x}/32$ ensures its memory region starts after `tensor_b`. This mapping allows threads within a warp to aggregate bit updates efficiently using atomic operations in shared memory. This significantly reduces contention on global memory because atomic operations occur within the low-latency shared memory space. After all threads in the

```
__global__ void BitMaskNaive(int8_t *matrix, cutlass::uint1b_t *tensor_b,
cutlass::uint1b_t *tensor_bc, int m, int k, int unique) {
    // Calculate the global index for the current thread
    uint64_t idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Compute the total number of elements in the matrix
    uint64_t num_elements = m * k;

    // Ensure the current thread's index is within bounds
    if (idx < num_elements) {
        // Calculate the byte index and bit position for the current element
        int byte_index = idx / 8; // Identify which byte the element belongs to
        int bit_index = idx % 8; // Identify the bit position within the byte

        // Compute pointers to the 4-byte aligned memory locations for tensor_b and tensor_bc
        unsigned int* ptr_b = (unsigned int*)&tensor_b[byte_index & ~3];   // 4-byte aligned pointer for tensor_b
        unsigned int* ptr_bc = (unsigned int*)&tensor_bc[byte_index & ~3]; // 4-byte aligned pointer for tensor_bc

        // Calculate the exact bit position within the aligned 4-byte word
        int bit_position = ((byte_index & 3) * 8) + bit_index; // Determine the bit offset in the 32-bit region

        // Create a bitmask for tensor_b: set the bit if the matrix value matches the unique value
        unsigned int mask_b = (matrix[idx] == unique) ? (1U << bit_position) : 0;
        // Create a bitmask for tensor_bc: set the bit if the matrix value does not match the unique value
        unsigned int mask_bc = (matrix[idx] != unique) ? (1U << bit_position) : 0;

        // Perform an atomic OR operation to safely update the bit in tensor_b if the mask is non-zero
        if (mask_b != 0)
            atomicOr(ptr_b, mask_b);

        // Perform an atomic OR operation to safely update the bit in tensor_bc if the mask is non-zero
        if (mask_bc != 0)
            atomicOr(ptr_bc, mask_bc);

    }
}
```

TABLE III: Bit-mask matrix construction (naive).

block finish updating shared memory, a synchronization point `__syncthreads()` ensures that all updates are complete before writing back to global memory.

To write results back, only one thread per warp, referred to as the warp leader (determined by $threadIdx.x\%32 == 0$), writes the aggregated results from shared memory back to global memory. This avoids redundant memory accesses and reduces contention on global memory. The global memory pointers for `tensor_b` and `tensor_bc` are calculated by aligning the byte index to a 4-byte boundary, ensuring efficient and coalesced memory operations. The aggregated bitmasks stored in shared memory are written back to global memory as 4-byte integers, minimizing the number of memory transactions. This approach amortizes the cost of global memory writes across threads within a warp, as all threads first contribute their updates in shared memory, and only the warp leader performs the final write.

The next step is to instantiate the CUTLASS GEMM object. The CUTLASS framework divides the GEMM kernel into a hierarchical tile-based structure to efficiently map computations to GPU hardware. At the threadblock level, we compute tiles of size $64 \times 128 \times 512$. These tiles are further subdivided into warp tiles of size $64{\times}64{\times}512$, with each warp responsible for a specific section of the computation. At the lowest level, we use Tensor Core operations to compute MMA (Matrix Multiply-Accumulate) tiles of size $16{\times}8{\times}256$. In CUTLASS, this multi-level tiling strategy ensures high utilization of the GPU's computational resources, minimizing memory access latency and improving data reuse.

VII. Performance Results

A. Experimental Settings

The experiments utilized five distinct computing platforms.

- `Guyot`: This platform consists of a single compute node, including two 64-core EPYC 7742 CPUs running at 2.25 GHz, 2063 GB of memory, and eight NVIDIA A100-SXM4-80GB GPUs.
- `Qaysar`: This platform consists of a single compute node, including two 28-core Intel(R) Xeon(R) Gold 6330 CPU running at 2.00 GHz, 1008 GB of memory, and four NVIDIA A100-SXM4-80GB GPUs.
- `Vulture`: This platform consists of a single compute node, including two 20-core Intel(R) Xeon(R) Gold 6148

```
__global__ void BitMaskOptimized(int8_t *matrix, cutlass::uint1b_t *tensor_b, cutlass::uint1b_t
    *tensor_bc, int m, int k, int unique) {
    extern __shared__ unsigned int shared_data[];  // Shared memory for both tensors' bit aggregation
    uint64_t idx = blockIdx.x * blockDim.x + threadIdx.x;

    uint64_t num_elements = m * k;

    int byte_index = idx / 8;
    int bit_index = idx % 8;

    if (idx < num_elements) {
        // Calculate the index within the shared memory (doubling the allocation for tensor_b and
            tensor_bc)
        int shared_index_b = threadIdx.x / 32;  // For tensor_b
        int shared_index_bc = (blockDim.x / 32) + shared_index_bc;  // Offset for tensor_bc

        if (threadIdx.x % 32 == 0) {
            shared_data[shared_index_b] = 0;  // Initialize shared memory for tensor_b
                shared_data[shared_index_bc] = 0;  // Initialize shared memory for tensor_bc
        }
        __syncthreads();  // Ensure initialization is completed before proceeding

        // Calculate bit position and mask
        int bit_position = (threadIdx.x % 32) * 8 + bit_index;  // Local bit position within shared memory
        unsigned int mask_b = (matrix[idx] == unique) ? (1U << bit_position) : 0;
        unsigned int mask_bc = (matrix[idx] != unique) ? (1U << bit_position) : 0;

        // Use atomic operations within shared memory to set bits
        if (mask_b != 0)
            atomicOr(&shared_data[shared_index_a], mask_b);

        if (mask_bc != 0)
            atomicOr(&shared_data[shared_index_b], mask_bc);

        __syncthreads();  // Ensure all threads have updated shared memory

        // Write back to global memory from shared memory
        if (threadIdx.x % 32 == 0) {
            unsigned int* global_ptr_b = (unsigned int*)&tensor_b[byte_index & ~3];
            unsigned int* global_ptr_bc = (unsigned int*)&tensor_bc[byte_index & ~3];
            *global_ptr_b = shared_data[shared_index_b];
            *global_ptr_bc = shared_data[shared_index_bc];
        } }
}
```

TABLE IV: Bit-mask matrix construction (optimized).

CPU CPU running at 2.40 GHz, 377 GB of memory, and four NVIDIA V100-PCIE-16GB GPUs.

- `Summit`: This system is built on GPU-based IBM technology, comprising 4,356 nodes. Each compute node features two 22-core Power9 CPUs clocked at 3.07 GHz, paired with 256 GB of memory. Additionally, every CPU integrates three NVIDIA Tesla V100 GPUs.
- `Frontier`: This AMD GPU-based supercomputer features 9,408 nodes. Each node is equipped with a 64-core AMD Optimized 3rd Generation EPYC processor, complemented by four AMD MI250X GPUs and 512 GB of DDR4 memory.

For the dataset, we use simulated DNA sequences generated using the msprime package. Each sequence consists of four unique nucleotide symbols—A (Adenine), C (Cytosine), T (Thymine), and G (Guanine).

B. Comparing Vector- and Matrix-based Implementation

Figure 2 illustrates the time-to-solution for Hamming distance computation, comparing a vector-based implementation with three matrix-based implementations on a CPU. The vector-based implementation employs Julia's threading parallelism. The matrix-based implementations include one using Julia's Dagger.jl framework with a task-based scheduler and another utilizing the PaRSEC task-based runtime system. All matrix-based implementations leverage the same matrix multiplication kernel, `cblas_gemm_s8u8s32`, provided by OneMKL. Notably, the task-based approaches achieve up to a 280X speedup over the vector-based implementation.

Fig. 3 illustrates the performance of matrix-based implementations, measured in TeraOps Per Second (TOPS). Four configurations are compared: Julia's CPU implementation, Julia's Dagger.jl framework with a task-based scheduler, PaRSEC on CPU, and PaRSEC on the `Qaysar` machine and a single NVIDIA V100 GPU on `Vulture`. The GPU-accelerated

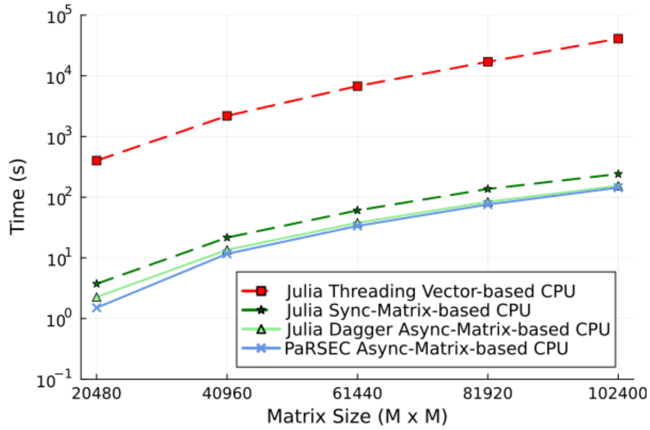

Fig. 2: Comparison of time to solution on `Qaysar` CPU for a vector-based Hamming distance and three matrix-based approaches: (1) Julia threading parallelism, (2) Julia Dagger.jl task-based framework, and (3) PaRSEC task-based runtime.

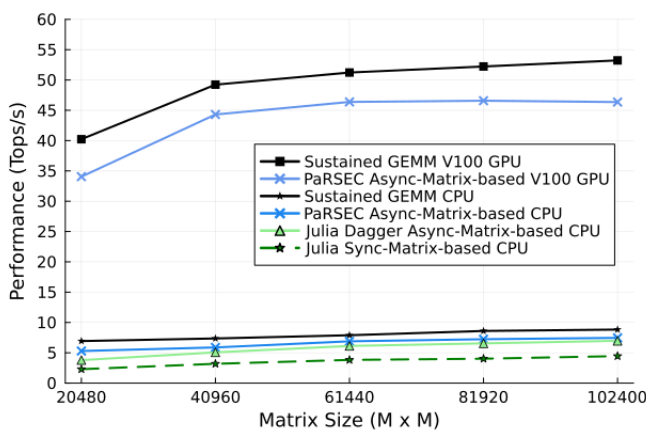

Fig. 3: Performance comparison of matrix-based $(8bi * 8bi + 32bi = 32bi)$ implementations measured in TOPS. The configurations include Julia CPU, Julia Dagger CPU, PaRSEC CPU, and PaRSEC on `Qaysar` and single NVIDIA V100 GPU on `Vulture`.

PaRSEC implementation significantly outperforms all CPU-based approaches. Among the CPU-based methods, PaRSEC and Julia Dagger consistently outperform the synchronous approach, benefiting from their efficient task-based runtime systems. This comparison highlights the performance advantages of leveraging GPUs and optimized task-based runtimes for large-scale matrix operations.

Fig. 4 describes the shared-memory performance on `Guyot` and `Frontier`. The performance is compared between using a single GPU (red triangles) and eight GPUs (blue stars). `Guyot` achieves a performance improvement of 6.72X with eight GPUs compared to one GPU, while the `Frontier` node achieves a 7.17X improvement. This near-linear scaling observed for eight GPUs highlights the ability to effectively distribute workloads across multiple GPUs, maximizing computational efficiency, particularly for larger matrix sizes.

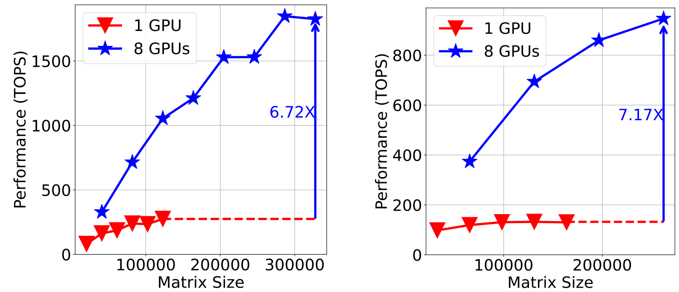

Fig. 4: Performance on shared memory. Left, `Guyot`; right, one `Frontier` node using PaRSEC with $(8bi * 8bi + 32bi = 32bi)$ GEMM from cuBLAS and hipBLAS.

TABLE V: Performance on Summit and Frontier (TOPS).

Number of Nodes	1	4	16	64	256
Summit	259	1,046	4,184	17,093	65,172
Summit Parallel Efficiency	100%	101%	101%	103%	98%
Summit % Theoretical Peak	70%	70%	70%	72%	68%
Frontier	931	3,604	14,748	53,900	198,361
Frontier Parallel Efficiency	100%	97%	99%	90%	83%
Frontier % Theoretical Peak	61%	59%	60%	55%	51%

Fig. 5 depicts the weak scaling performance on `Summit`. The x-axis indicates the number of compute nodes, while the y-axis presents the normalized performance per node in TOPS. Each curve represents a specific matrix size, scaled with the square root of the number of nodes. The results demonstrate that performance remains almost constant as the number of nodes increases, achieving over 68% of the theoretical peak in all cases for larger matrix sizes, as showcased in Table V. This experiment underscores the capability to sustain high throughput in a weak scaling configuration.

Table V summarizes the maximum performance achieved

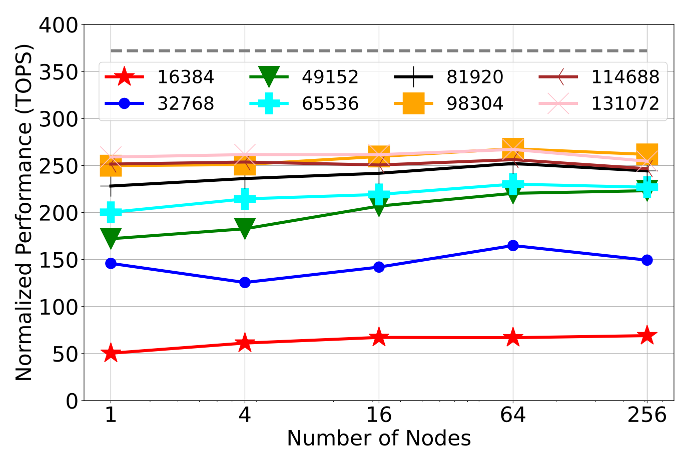

Fig. 5: Performance scalability on `Summit`: normalized performance per node (i.e., 6 NVIDIA V100 GPUs) in a weak scaling setting. The dashed gray line is the theoretical peak per node. Each solid line represents a normalized size N, i.e., the matrix size on n nodes is $N \times sqrt(n)$.

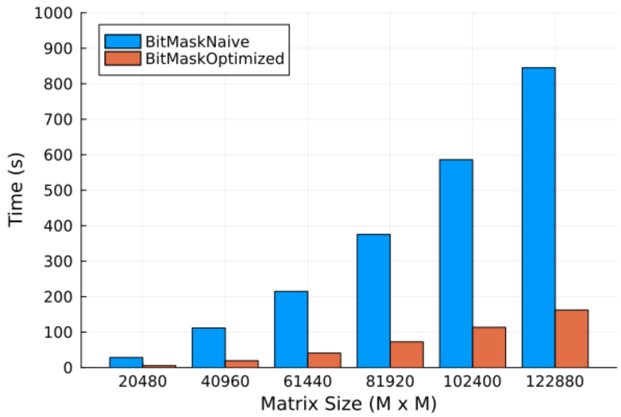

Fig. 6: Execution time comparison of naive (blue) and optimized (orange) bitmasking on a single NVIDIA A100 GPU.

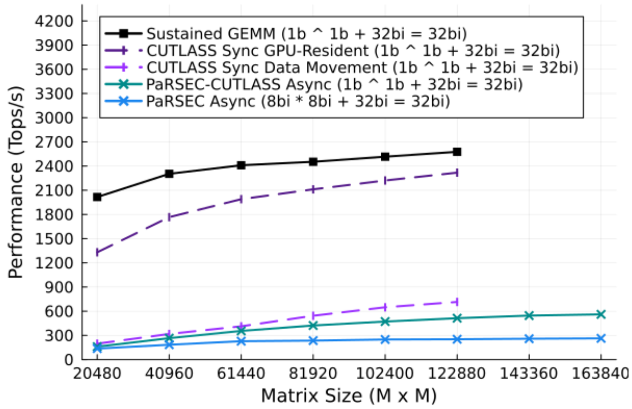

Fig. 7: Performance comparison of matrix-based implementations between CUTLASS with GEMM $(1b \;\hat{}\; 1b + i32 = i32)$ and PaRSEC with GEMM $8bi \times 8bi + 32bi = 32bi$ on a single NVIDIA A100 GPU.

on `Summit` and `Frontier` for various node counts, along with the corresponding parallel efficiency and the ratio to the theoretical peak performance. On `Summit`, performance scales from 259 TOPS on a single node to 65,172 TOPS across 256 nodes, achieving 98% parallel efficiency and 68% of the theoretical peak. On `Frontier`, the results exhibit even higher performance, starting at 931 TOPS on a single node and scaling to 198,361 TOPS with 256 nodes. While the parallel efficiency decreases slightly as node count increases—likely due to the increased impact of communication overhead caused by faster computations—`Frontier` still achieves 83% parallel efficiency and 51% of the theoretical peak with 256 nodes. These findings highlight the exceptional performance and efficient scalability on `Summit` and `Frontier`.

Fig. 6 highlights the significant impact of shared memory optimization and warp-level cooperation on execution time. The naive approach, represented by the blue bars, directly performs atomic operations on global memory. In contrast, the novel optimized approach, represented by the orange bars, utilizes shared memory at the warp-level to aggregate updates within a thread block before writing back to global memory. By assigning one thread per warp (the warp leader) to handle the final write-back, the approach minimizes redundant memory transactions and ensures efficient use of shared memory. This approach achieves a 5X speedup over the naive one.

TABLE VI: Improvement of different implementations.

	Vector-based CPU	Matrix-based CPU	Async-Matrix-based CPU 8-bit Int	Async-Matrix-based GPU 8-bit Int	Async-Matrix-based GPU 1-bit Int
Time (s)	40877.9	241.23	143.943	16.699	8.84
Incremental Speedup	–	169x	1.67x	8.62x	1.89x
Overall Speedup	–	169x	284x	2447x	4624x

Figure 7 compares the GPU-based performance across different implementations. It includes the pure CUTLASS resident implementation using 1-bit integers to showcase the performance of GPU-resident matrices without accounting for data movement. Additionally, it evaluates the pure CUTLASS implementation while considering data movement, representing the ideal scenario for typical applications. The figure further highlights the performance of PaRSEC-CUTLASS with 1-bit integers, incorporating data movement in 8-bit precision, and PaRSEC with the cuBLAS 8-bit integer standardized implementation in CUDA. Notably, PaRSEC-CUTLASS achieves performance comparable to pure CUTLASS with data movement. From a memory perspective, CUTLASS explicitly constructs both the matching mask matrix and its complement, requiring storage for the entire symmetric matrix H. In contrast, the PaRSEC asynchronous implementation leverages the symmetry of H, allocating storage only for its upper or lower triangular part while dynamically constructing the matching mask and its complement on-the-fly.

Table VI summarizes the performance improvements of various implementations, along with their overall speedup to the vector-based CPU approach and the incremental speedup. The results demonstrate a substantial reduction in execution time, transitioning from the vector-based CPU implementation to asynchronous matrix-based implementations on GPUs, achieving improvements of up to three orders of magnitude.

VIII. Conclusion

This paper introduces a novel matrix-based Hamming distance algorithm that achieves unprecedented efficiency and scalability on modern heterogeneous architectures. The algorithm leverages the combined strengths of CUTLASS, Julia, and the PaRSEC runtime system to deliver high performance for large-scale computations. By transforming the traditional element-wise comparisons into efficient matrix operations, this approach significantly enhances both computation and resource utilization, making it well-suited for diverse hardware platforms, including NVIDIA and AMD GPUs.

Looking forward, we plan to reduce the communication volume in Julia's and PaRSEC's implementations to as little as 1 bit, as our current experiments indicate a communication bottleneck for low-precision computations. We also intend to explore sparsity in the data, particularly in scenarios where the number of unique elements is large.

ACKNOWLEDGMENT

For computer time, this research used the Ibex cluster at the Supercomputing Laboratory of the King Abdullah University of Science and Technology (KAUST) in Thuwal, Saudi Arabia; Frontier and Summit at the Oak Ridge Leadership Computing Facility of the US DOE's Oak Ridge National Laboratory, and Guyot at Innovative Computing Lab of the University of Tennessee, Knoxville. Authors want to thanks the supported by the U.S. National Science Foundation (awards CNS-2346520, PHY-2028125, RISE-2425761, DMS-2325184, OAC-2103804, and OSI-2029670), the Defense Advanced Research Projects Agency (DARPA HR00112490488), the Department of Energy, National Nuclear Security Administration (DE-NA0003965) and by the United States Air Force Research Laboratory (FA8750-19-2-1000). In addition, Authors would like to acknowledge KAUST Ibn Rushd post-doctoral fellowship which supported Rabab Alomairy.

REFERENCES

[1] Sameh Abdulah, Allison H. Baker, George Bosilca, Qinglei Cao, Stefano Castruccio, Marc G. Genton, David E. Keyes, Zubair Khalid, Hatem Ltaief, Yan Song, Georgiy L. Stenchikov, and Ying Sun. Boosting earth system model outputs and saving petabytes in their storage using exascale climate emulators. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024.

[2] Rabab Alomairy, Hatem Ltaief, Mustafa Abduljabbar, and David Keyes. Abstraction Layer for Standardizing APIs of Task-Based Engines. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2482–2495, 2020.

[3] Rabab Alomairy, Felipe Tome, Julian Samaroo, and Alan Edelman. Dynamic task scheduling with data dependency awareness using julia. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2024.

[4] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency Computat. Pract. Exper.*, 23:187–198, 2011.

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012.

[6] Maciej Besta, Raghavendra Kanakagiri, Harun Mustafa, Mikhail Karasikov, Gunnar Rätsch, Torsten Hoefler, and Edgar Solomonik. Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1122–1132. IEEE, 2020.

[7] Jeff Bezanson, Jiahao Chen, Stefan Karpinski, Viral Shah, and Alan Edelman. Array Operators Using Multiple Dispatch: A design methodology for array implementations in dynamic languages. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, page 56–61, New York, NY, USA, 2014. Association for Computing Machinery.

[8] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Hérault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack J. Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*, pages 1432–1441. IEEE, 2011.

[9] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Thomas Hérault, Pierre Lemarinier, and Jack J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Comput.*, 38(1-2):37–51, 2012.

[10] George Bosilca, Robert J Harrison, Thomas Herault, Mohammad Mahdi Javanmard, P Nookala, and Edward F Valeev. The Template Task Graph (TTG)-an Emerging Practical Dataflow Programming Paradigm for Scientific Simulation at Extreme Scale. In *IEEE/ACM 5th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 2020.

[11] Aurelien Bouteiller, Thomas Herault, Qinglei Cao, Joseph Schuchart, and George Bosilca. PaRSEC: Scalability, flexibility, and hybrid architecture support for task-based applications in ECP. *International Journal of High Performance Computing Applications*, 2024.

[12] Qinglei Cao, Sameh Abdulah, Rabab Alomairy, Yu Pei, Pratik Nag, George Bosilca, Jack Dongarra, Marc G Genton, David E Keyes, Hatem Ltaief, and Others. Reshaping Geostatistical Modeling and Prediction for Extreme-Scale Environmental Applications. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2022.

[13] Qinglei Cao, Sameh Abdulah, Hatem Ltaief, Marc G Genton, David Keyes, and George Bosilca. Reducing data motion and energy consumption of geospatial modeling applications using automated precision conversion. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 330–342. IEEE, 2023.

[14] Qinglei Cao, Rabab Alomairy, Yu Pei, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. A framework to exploit data sparsity in tile low-rank cholesky factorization. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 414–424. IEEE, 2022.

[15] Qinglei Cao, George Bosilca, Nuria Losada, Wei Wu, Dong Zhong, and Jack Dongarra. Evaluating data redistribution in PaRSEC. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1856–1872, 2021.

[16] Qinglei Cao, Yu Pei, Kadir Akbudak, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Leveraging PaRSEC runtime support to tackle challenging 3d data-sparse matrix problems. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 79–89. IEEE, 2021.

[17] Qinglei Cao, Yu Pei, Thomas Herault, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Performance analysis of tile low-rank cholesky factorization using PaRSEC instrumentation tools. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pages 25–32, 2019.

[18] David Castells-Rufas, Santiago Marco-Sola, Juan Carlos Moure, Quim Aguado, and Antonio Espinosa. FPGA Acceleration of Pre-Alignment Filters for Short Read Mapping with HLS. *IEEE Access*, 10:22079–22100, 2022.

[19] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. PTG: an Abstraction for Unhindered Parallelism. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 21–30. IEEE, 2014.

[20] A. Duran, R. Ferrer, E. Ayguade, R. M. Badia, and J. Labarta. A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks. *International Journal of Parallel Programming*, 37(3), 2009.

[21] Youssef Elmougy, Akihiro Hayashi, and Vivek Sarkar. Asynchronous Distributed Actor-Based Approach to Jaccard Similarity for Genome Comparisons. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pages 1–11. Prometeus GmbH, 2024.

[22] T. Heller, H. Kaiser, and K. Iglberger. Application of the ParalleX Execution Model to Stencil-Based Problems. *Computer Science - Research and Development*, 28(2-3), 2013.

[23] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra. Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17, 2017.

[24] Gengjie Jia, Yu Li, Xue Zhong, Kanix Wang, Milton Pividori, Rabab Alomairy, Aniello Esposito, Hatem Ltaief, Chikashi Terao, Masato Akiyama, Koichi Matsuda, David Keyes, Hae Kyung Im, Takashi Gojobori, Yoichiro Kamatani, Michiaki Kubo, Nancy J. Cox, James Evans, Xin Gao, and Andrey Rzhetsky. The high-dimensional space of human diseases built from diagnosis records and mapped to genetic loci. *Nature Computational Science*, 3(5):403–417, 2023.

[25] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Alvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M Badia. Servicess: An Interoperable Programming Framework for the Cloud. *Journal of grid computing*, 12(1):67–91, 2014.

[26] Hatem Ltaief, Rabab Alomairy, Qinglei Cao, Jie Ren, Lotfi Slim, Thorsten Kurth, Benedikt Dorschner, Salim Bougouffa, Rached Abdelkhalak, and David E. Keyes. Toward capturing genetic epistasis from

multivariate genome-wide association studies using mixed-precision kernel ridge regression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024.
[27] NVIDIA. CUTLASS. https://github.com/NVIDIA/cutlass, 2025.
[28] OpenMP. OpenMP 6.0 Complete Specifications, 2024.
[29] Sarah Pilz, Florian Porrmann, Martin Kaiser, Jens Hagemeyer, James M Hogan, and Ulrich Rückert. Accelerating Binary String Comparisons with a Scalable, Streaming-Based System Architecture based on FPGAs. *Algorithms*, 13(2):47, 2020.
[30] Hildete Prisco Pinheiro, Aluísio de Souza Pinheiro, and Pranab Kumar Sen. Comparison of Genomic Sequences using the Hamming Distance. *Journal of Statistical Planning and Inference*, 130(1-2):325–339, 2005.
[31] Julian Samaroo, Rabab Alomairy, and Felipe Tome. Dagger.jl. https://github.com/JuliaParallel/Dagger.jl, 2024.
[32] Andrew Todd, Marziyeh Nourian, and Michela Becchi. A Memory-Efficient GPU Method for Hamming and Levenshtein Distance Similarity. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 408–418. IEEE, 2017.
[33] Sophie Xuan, Evelyne Ringoot, Rabab Alomairy, Felipe Tome, Julian Samaroo, and Alan Edelman. Synthesizing Numerical Linear Algebra using Julia. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2024.