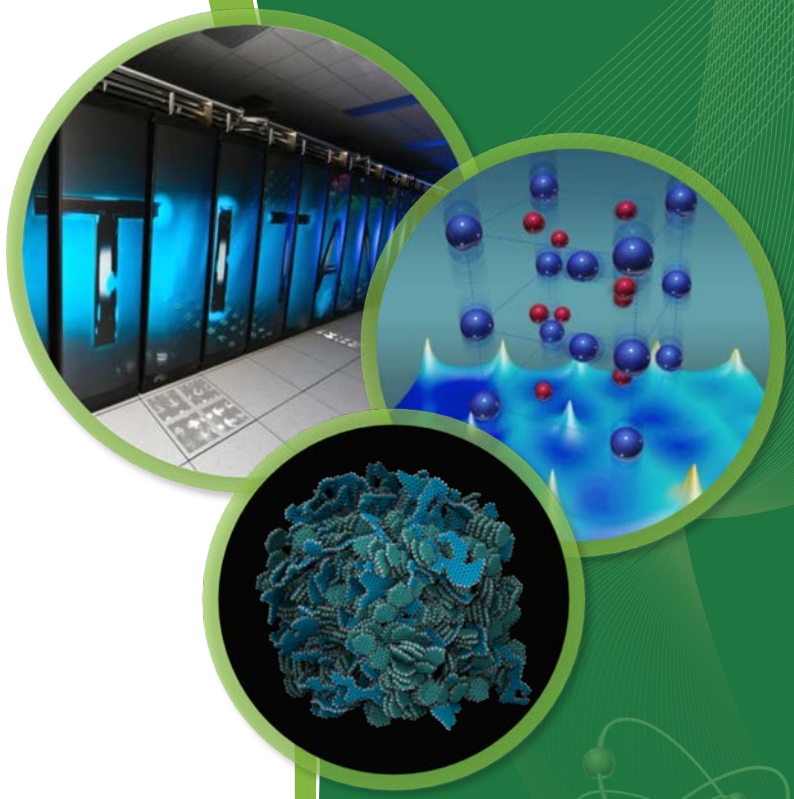
CoMet:
An HPC application for comparative genomics calculations



Wayne Joubert

Scientific Computing Group
Oak Ridge Leadership Computing Facility
Oak Ridge National Laboratory

Background

We focus on methods for exhaustive search to find relationships in genomic data

These methods are highly computationally expensive: datasets are large, and algorithmic computational complexity is quadratic, cubic or higher

These are relatively new methods to HPC

Here we will describe a new approach to solving these problems on Titan and Summit that is an advance over current state of the art

https://commons.wikimedia.org/wiki/File:DNA_com_GGN.jpg

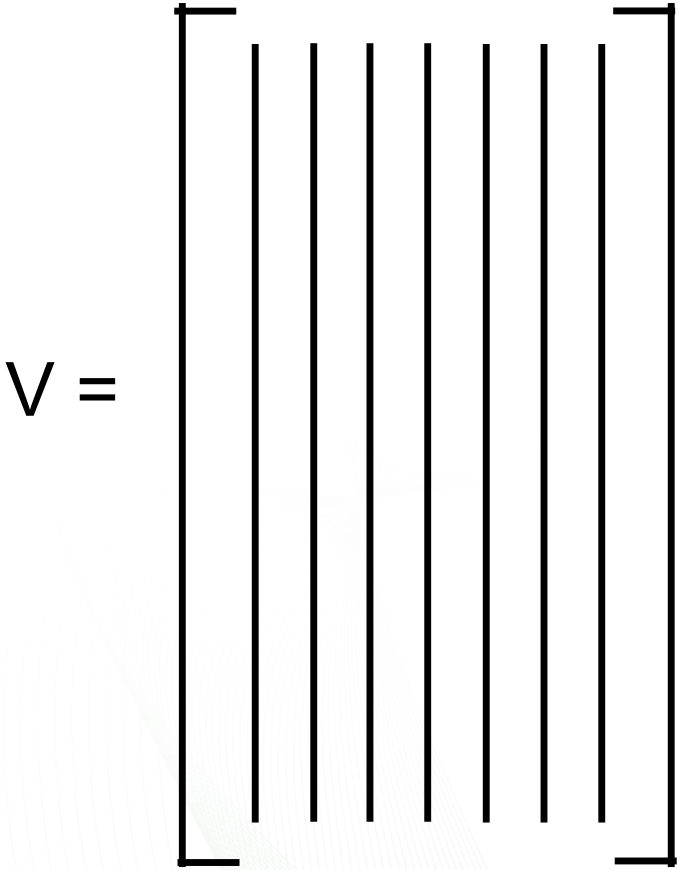
Vector similarity methods

Many of these methods can be represented as vector similarity calculations

Given a set of vectors, we seek pairs of vectors that are similar to each other

Example application: columns of matrix V represent nucleotide positions in a genome (SNPs), rows of V represent members of a population of individuals

Two columns are mathematically “similar” if a feature present in the two respective nucleotide positions for two vectors co-occurs in the same individuals in the population



Vector similarity

Many vector similarity measures have been proposed for various scientific uses

Perhaps most well-known: cosine similarity: d(u,v) = |(u,v)| / ||u|| ||v||, normalized vector inner product

Two methods of interest here:

1. Proportional Similarity (or Czekanowski) metric
2. Custom correlation coefficient

We define these methods and describe how they are mapped to GPUs

Proportional Similarity metric

Assume a set of floating point vectors (single or double precision)

Given two vectors u and v, define the *Proportional Similarity metric:*

$$c_2(u, v) = 2 \cdot n_2(u, v) / d_2(u, v)$$

denominator term: $d_2(u, v) = \sum_k (u_k + v_k) = \sum_k u_k + \sum_k v_k$

numerator term: $n_2(u, v) = \sum_k \mathbf{min}(u_k , v_k)$

The full problem: let $V = [v_1 \quad v_2 \quad v_3 \quad \bullet\bullet\bullet \quad v_n]$, (V is m X n),

Compute $c_2(v_i, v_j)$ for all n^2 vector pairs

(Note: only half are unique, due to symmetry)

Computing vector similarity

Simplistic approach: compute $c_2(v_i, v_j)$ individually for every pair – this is a memory bandwidth bound computation since two full vectors must be read from memory to compute a single result

However, the entire computation for n vectors of length m is $O(mn^2)$ operations on O(mn) data – suggesting a more computationally intensive approach may be possible

In fact, we already know how to solve problems of this type efficiently –

For example, the cosine similarity metric (inner products) for a set of vectors can be computed very efficiently with an optimized BLAS-3 DGEMM dense matrix-matrix product

For the Proportionality Similarity metric we can compute the numerator term $n_2(\bullet, \bullet)$ by replacing the scalar multiply add c = c + a * b in the DGEMM with the operation c = c + min(a,b)

Implementation

- Pre-existing highly optimized software for the xGEMM operation can be repurposed to compute the respective vector similarity metric
- Here we use the MAGMA library – well-tuned to the GPU (memory hierarchy, registers, threads, etc.) – modify the relevant xGEMM kernel in MAGMA to implement the specific similarity measure
- Added bonus: for the Proportional Similarity metric, one can use CUDA intrinsics **`fmin`** / **`fminf`** to take the minimum of two scalars, these map to hardware instructions on the GPU – to get very high fraction of GPU peak performance

3-way Proportional Similarity metric

Some genomic features can only be found by comparing 3 vectors at a time (higher dimensional relationships)

Given vectors u, v, w, define the *3-way Proportional Similarity metric:*

$$c_3(u, v) = (3 / 2) \cdot n_3(u, v, w) / d_3(u, v, w)$$

where $d_3(u, v, w) = \sum_k (u_k + v_k + w_k) = \sum_k u_k + \sum_k v_k + \sum_k w_k$

and $n_3(u, v, w) = n_2(u,v) + n_2(u,w) + n_2(v,w) - \sum_k \mathbf{min}(u_k, v_k, w_k)$

The full problem: Compute $c_3(v_i, v_j, v_k)$ for all n^3 vector triples

(Note: only 1/6 are unique, due to symmetries)

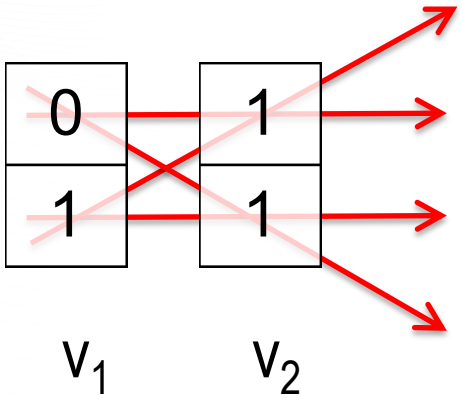
How to compute efficiently on the GPU?

- The numerator has three lower order terms: these can be computed using the 2-way approach described earlier (lower complexity, $O(mn^2)$)
- The 3-way term $\sum_k \mathbf{min}(u_k, v_k, w_k)$ is $O(mn^3)$ – a "BLAS-4-like" operation
- Strategy to solve this: convert the 3-way calculation into a series of 2-way problems:
- In particular, for matrix V, fix vector v_j, let X_j be the elementwise minimum of column v_j with all columns V
- Apply the 2-way computation method to the matrix pair [V, X_j], combine results to form 3-way term
- Exploit the optimized 2-way method performance on the GPU

Custom Correlation Coefficient (CCC)

- CCC is used to compute the interactions between alleles in a genomic dataset
- Here each vector entry is an allele encoded as a pair of bits
- The method applied to two vectors gives a 2X2 tally table of results

Custom Correlation Coefficient: illustration



Consider a pair of vectors, each with one element composed of 2 bits

1	1
0	1
1	1
0	1

Take all four 2-bit combinations of one bit from the first vector concatenated with one bit from the second vector

00 0	10 0
01 2	11 2

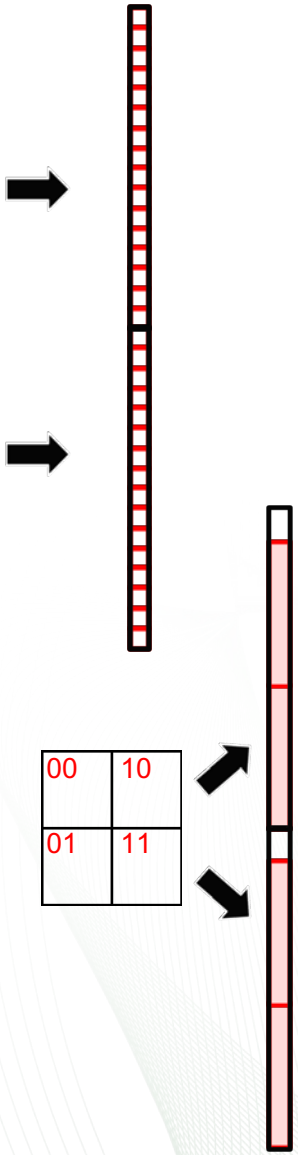
Tally the counts of number of occurrences of each of these possibilities into a 2x2 table

How to compute efficiently on the GPU?

- This again has the same computational pattern as xGEMM
- Use the same approach, modifying the MAGMA to support this operation
- Requires special attention given to bit-manipulation operations

Implementation

- Based on the double precision complex ZGEMM kernel from MAGMA
- Each vector entry has 2 doubles (128 bits)
- Input vector: pack 64 of these 2-bit values into each double-complex MAGMA vector element
- Output result: the 4 values in the tally table are stored as 25-bit integers packed into the two 52-bit mantissas of the two double words
- Requires bitwise operations (mask, shift, etc.) on GPU
- Added bonus: can use the CUDA population count **`__popcll`** intrinsic for fast counting of bits in hardware
- Very high computational intensity
- A variant of the method, CCC/sp is used for the case when the input data may have missing entries



3-way Custom Correlation Coefficient

- As before, compares three vectors at a time
- Has analogous definition
- Mapped to GPU as before, by converting to a series of 2-way computations
- Each element requires “3” 2-way computations (instead of 1 for PS)
- Do not have the three 2-way terms in the formula – thus shorter GPU pipeline startup cost

Strategy to map to thousands of GPUs

- Major challenge: the comparison metrics are symmetric: for example, $c_2(u, v) = c_2(v, u)$, so only half of the values need be computed (1/6 for 3-way case)
- Want to avoid 2X or 6X factor of wasted computations from symmetries
- The symmetry patterns are not amenable to easy load balancing

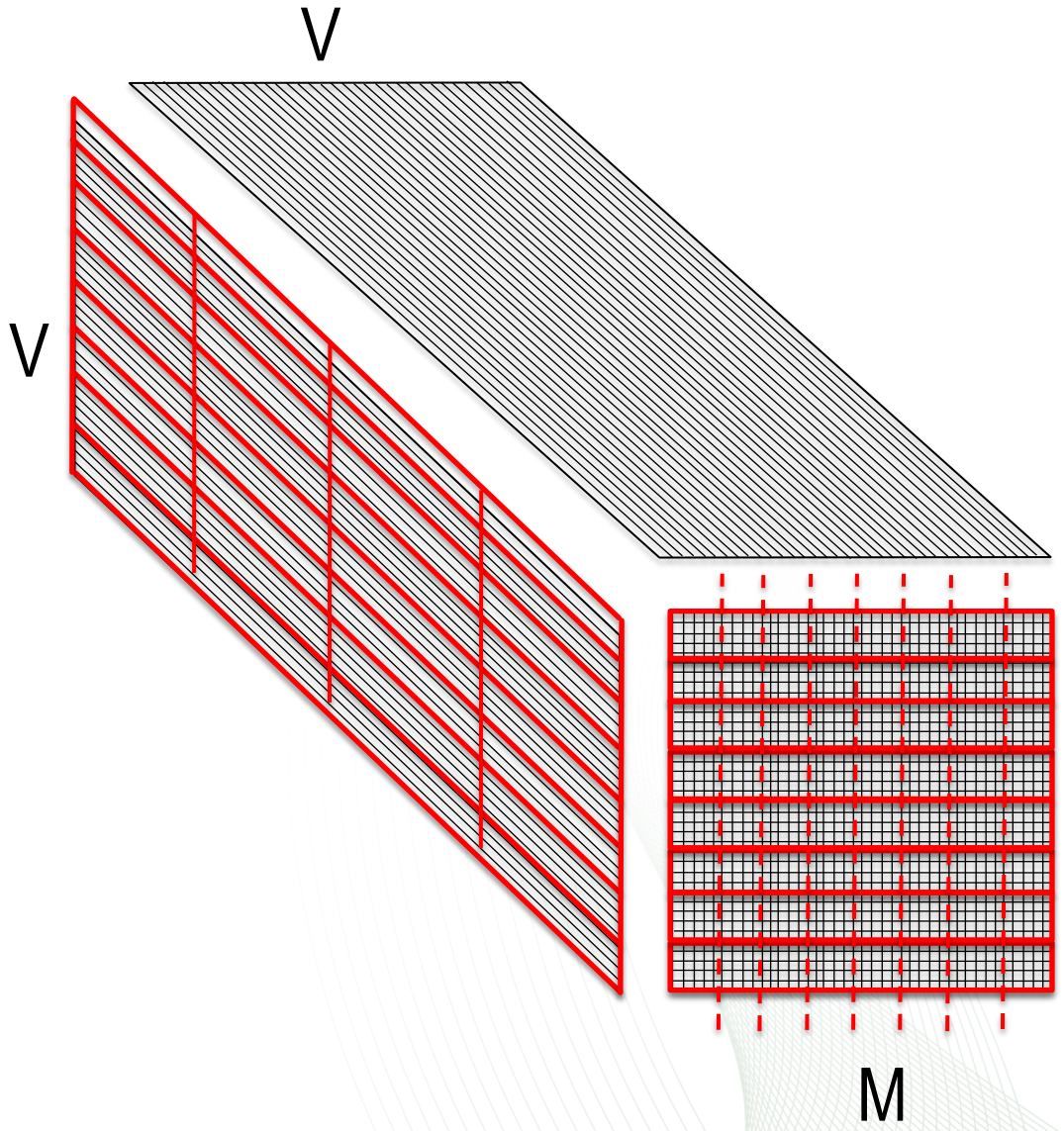
Parallelism strategy: 2-way methods

Must perform an all-to-all comparison of all vectors against all vectors

The computed results form a 2-D square matrix M

We parallelize across three axes:

1. The set of vectors is partitioned into subsets assigned to different GPUs—the matrix M is likewise divided into block rows assigned to GPUs
2. Each vector is subdivided into pieces assigned to different GPUs
3. Each block row of the matrix M is further subdivided into smaller pieces assigned to different GPUs

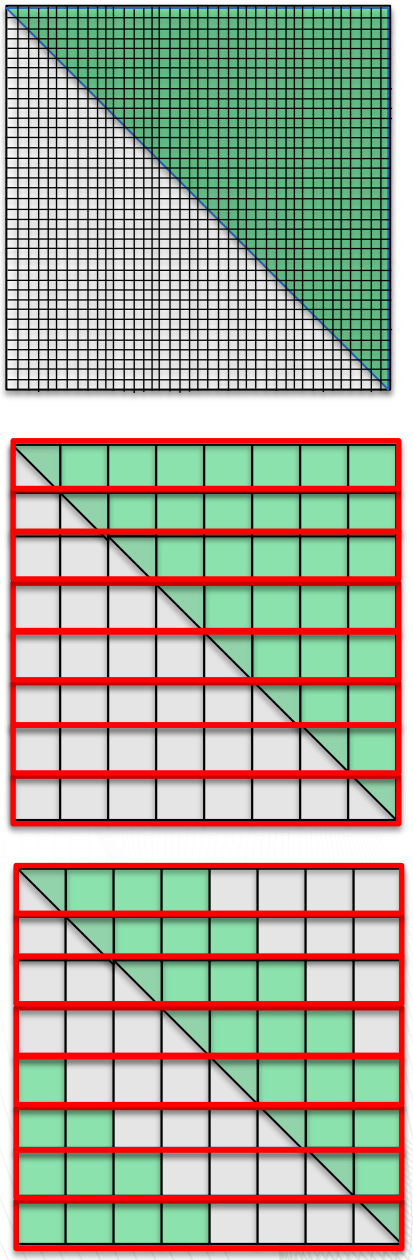


Eliminating redundancies

The upper triangular elements of the matrix M uniquely represent all the required values

Computing this triangle of entries would result in load imbalance – the block rows assigned to processors have different lengths

However, taking a *block-circulant* subset of the blocks will correctly capture every unique value and is also load balanced

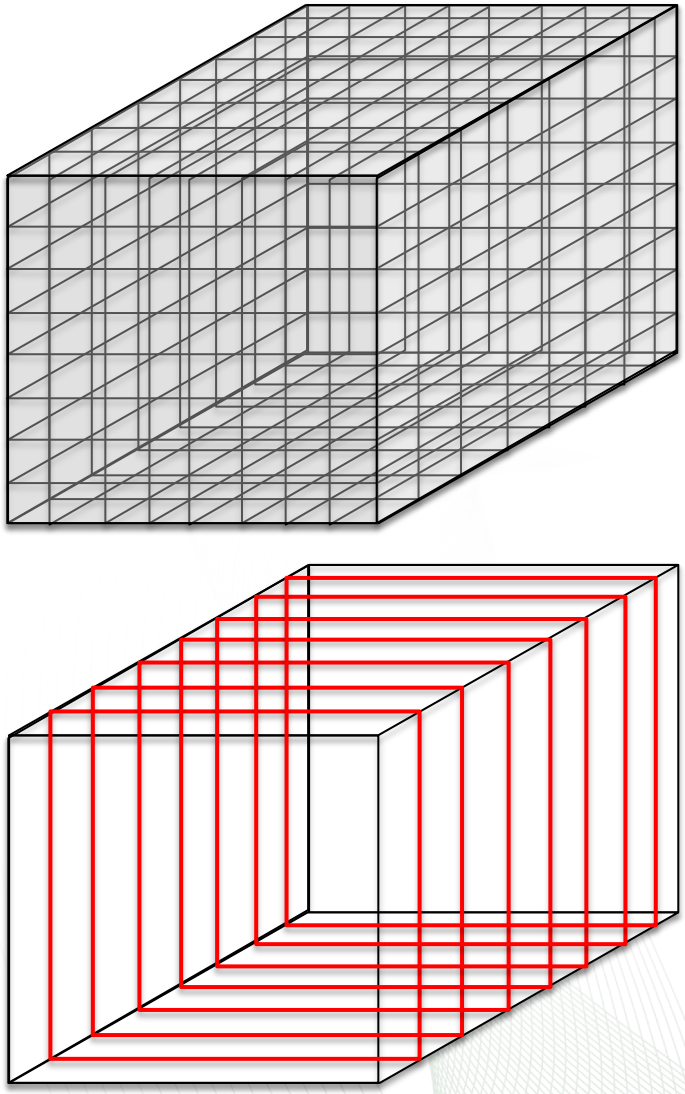


Parallelism strategy: 3-way methods

For the 3-way methods, now have a cube-shaped (i,j,k) tensor of result values to compute

Deploy the same three axes of parallelism as before: (1) partition the set of vectors into subsets assigned to GPUs, (2) divide each vector into pieces assigned to GPUs, and (3) further parallelize each 2-D slab of results across GPUs

This results in a decomposition on the cube of results into blocks and slabs



Eliminating redundancies

A tetrahedral region of the cube that represents all unique values

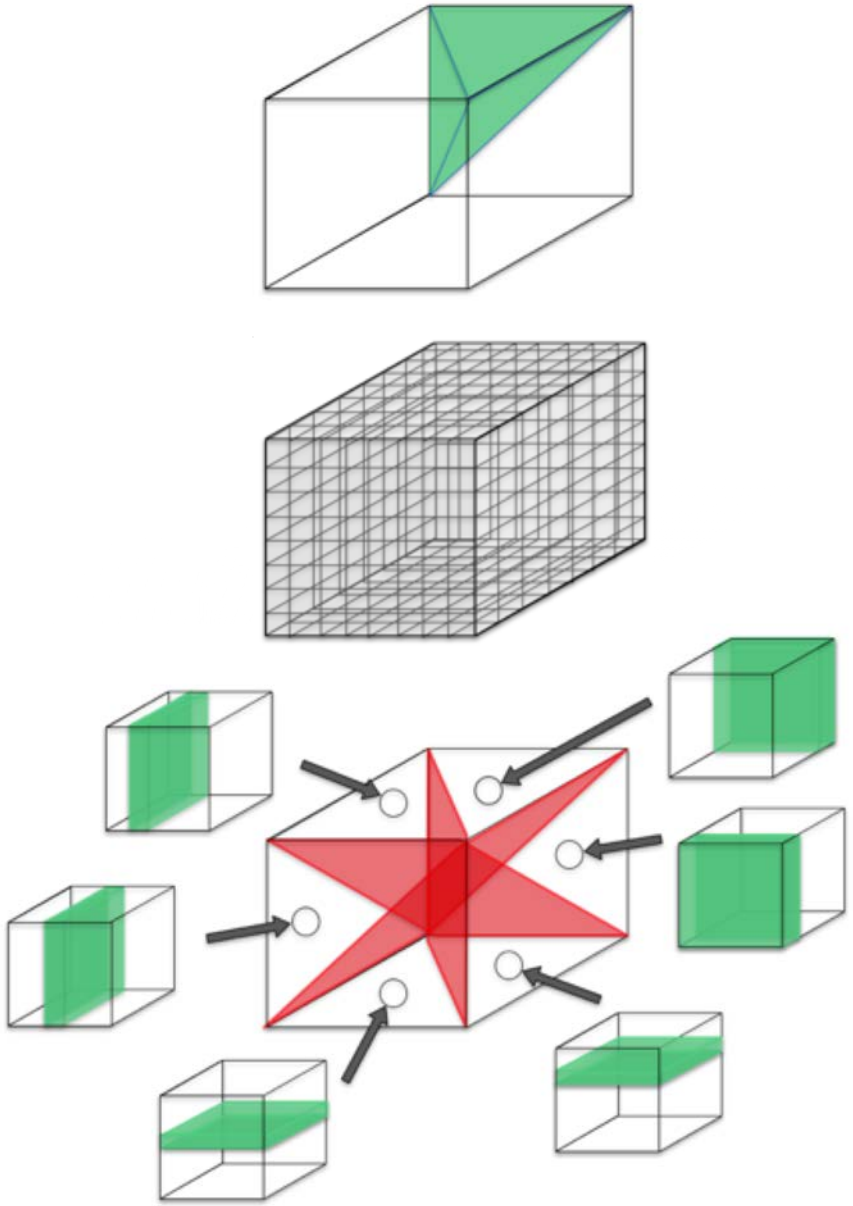
Have a potential 6X inefficiency due to redundancies

Computing this tetrahedron with this parallel decomposition is not load balanced

To solve this problem, for each block we select a special 1/6-sized slice and compute the results only for this slice

Can see by a folding/reflection argument that all blocks in a single tetrahedral region are fully covered

The computation is load balanced since every block has the same amount of work



Implementation details

- For high performance everything must be overlapped: GPU work, async communications, async GPU transfers, CPU work (uses OpenMP threading on CPU cores)
- The enormous size of the computed data requires dividing the computation into “phases” (2-way, 3-way) or “stages” (3-way) to reduce size of stored results
- For input, each MPI rank reads one part of a single file
- Output is typically thresholded by factor of 10^6 or more, and each rank writes to a single output file

Results: timings for a sample run

2-way CCC/sp, realistic dataset, 28M vectors of length 44,100 elements, 21 phases computed out of 200 total phases, on 6,000 Titan nodes

For large cases most of the time is spent in the core computation of the metrics

Operation	21 out of 200 phases	200 phases (est)	%
core metrics comp	938.345	*8936.615*	***89.54%***
vectors init	0.025	*0.025*	*0.00%*
metrics init	11.741	*111.817*	*1.12%*
input	515.546	*515.546*	*5.17%*
output	416.034	*416.034*	*4.17%*
TOTAL	**1,509.340**	***9,980.037***	***100.00%***

GPU kernel performance: Summit Volta V100

Measurements of GPU kernel only, for a large problem

Operation rate is **> 75%** of the xGEMM operation rate

Volta still maintains very high performance after replacing fused-multiply-add FMA with "**+**" and "**`fminf`**" for PS method

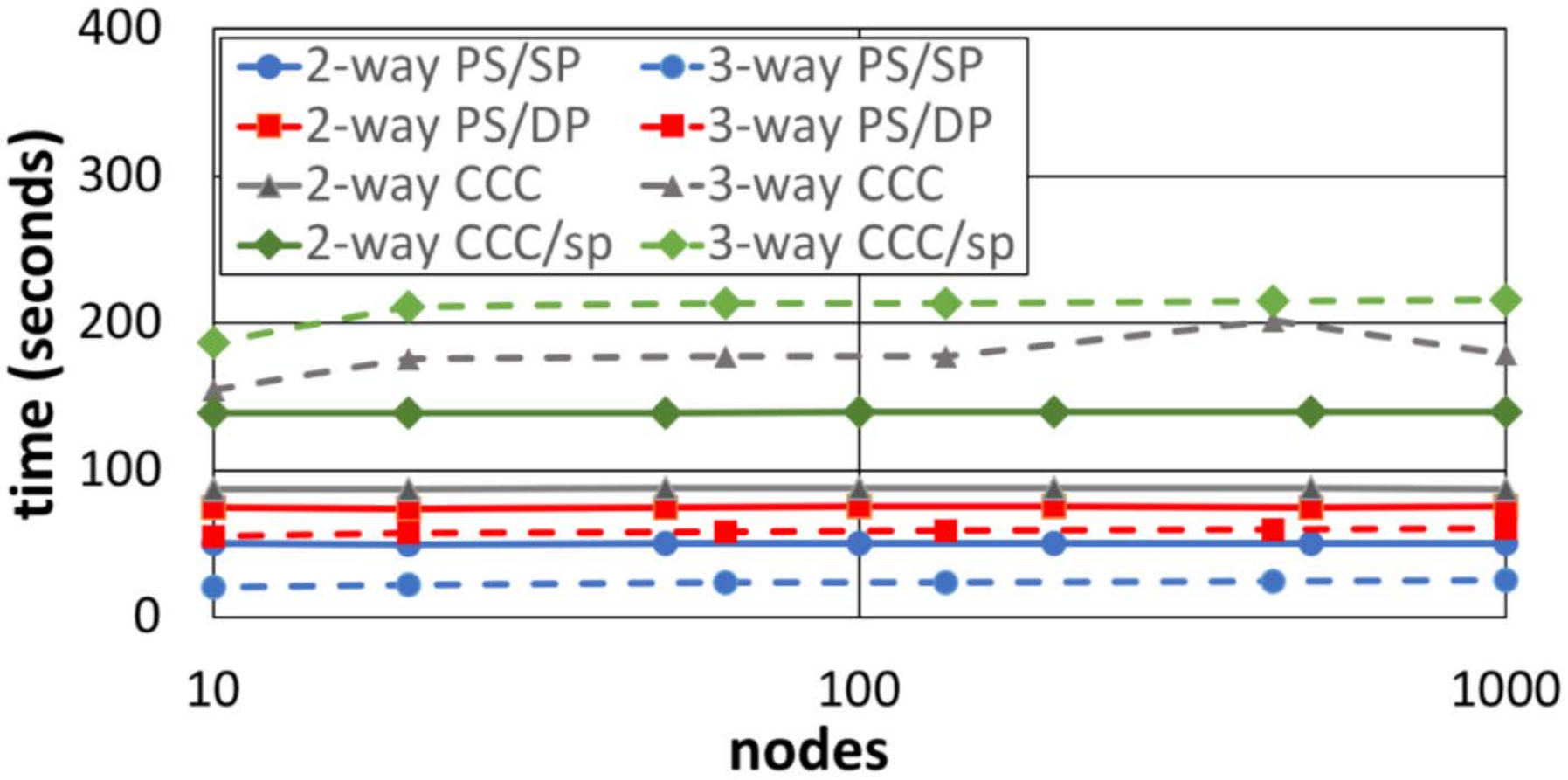
Kernel	elt pairs / sec $\times 10^{12}$	ops / elt pair	ops / sec $\times 10^{12}$	IPC
PS/SP	4.969	2	9.939	3.422
MAGMA SGEMM	6.453	2	12.907	2.559
SGEMM limit	7.834	2	**15.667**	—
PS/DP	1.276	4	5.104	1.364
MAGMA DGEMM	3.142	2	6.285	1.442
DGEMM limit	3.917	2	**7.834**	—
CCC 2-way	3.880	60	7.274	1.325
CCC 3-way	2.982	82	7.641	1.298
CCC/sp 2/3-way	2.501	88	6.879	1.208
MAGMA ZGEMM	0.649	6	3.894	1.148
ZGEMM limit	1.306	6	**7.834**	—

Parallel performance: Summit weak scaling

Large problem, similar problem size per node

Near-perfect weak scaling for all cases

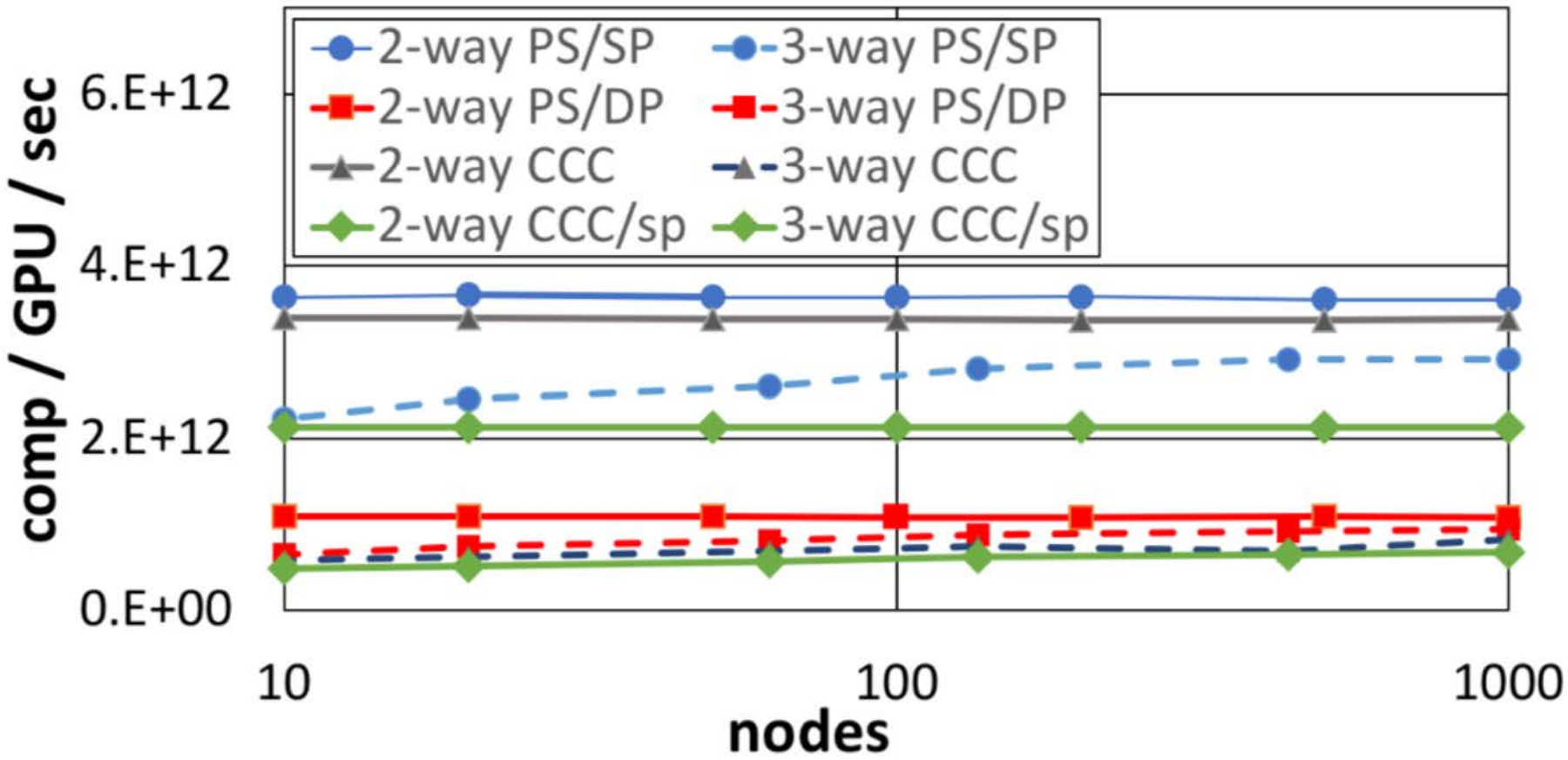
Expect near-perfect scaling to 4,608 nodes by using Summit fat tree topology and Adaptive Routing



Summit weak scaling: comparisons per rank per second

3-way methods less efficient at small node counts

All methods are efficient at high node counts



Summit performance at 1000 nodes, 4608 nodes

@ 1000 nodes of Summit: up to 22 quadrillion element comparisons (**43 PetaOps**)

Per-GPU performance at 1000 nodes is **59-87%** of single GPU kernel performance

Projection to full Summit: 100 quadrillion element comparisons (**199 PetaOps**)

method	num way	cmp / sec max nodes $\times 10^{15}$	cmp / sec per GPU $\times 10^{12}$	max limit per GPU $\times 10^{12}$	ratio per GPU	cmp / sec 4,608 nodes $\times 10^{15}$
PS/SP	2	21.643	3.607	4.969	72.6%	*99.731*
PS/DP	2	6.488	1.081	1.276	84.7%	*29.896*
CCC	2	20.337	3.389	3.880	87.4%	*93.711*
CCC/sp	2	12.764	2.127	2.501	85.1%	*58.815*
PS/SP	3	17.458	2.918	4.969	58.7%	*80.663*
PS/DP	3	5.644	0.943	1.276	73.9%	*26.079*
CCC	3	4.894	0.818	0.994	82.3%	*22.613*
CCC/sp	3	4.058	0.678	0.834	81.3%	*18.747*

Related work: 2-way methods

@ 1000 Summit nodes, 2-way CCC is **1,464X** faster than best competing result (512 nodes of Edison)

Full Summit: estimate **6,747X** faster

(Note: the $GWIS_{FI}$ code shown here is already **10,000 times faster** than the commonly used PLINK code)

code	problem	node config	nodes used	cmp/sec ($\times 10^9$)
GBOOST[32]	2-way GWAS	1 Nvidia GTX 285	1	64.08
GWISFI[33]	2-way GWAS	1 Nvidia GTX 470	1	767
[36]	2-way GWAS	1 Nvidia GTX 470	1	649
[36]	2-way GWAS	IBM Blue Gene/Q	4096	2520
epiSNP[37]	2-way GWAS	2 Intel Phi SE10P	126	1593
[34]	2-way GWAS	2 Nvidia K20m + 1 Intel Phi 5110P	1	1053
multiEpistSearch [39]	2-way GWAS	1 Nvidia GTX/Titan	24	12,626
[40]	2-way GWAS	2 Intel Xeon E5-4603	512	13,889
CoMet, Titan	2-way PS/SP	1 Nvidia K20X	17472	4.289e6
CoMet, Titan	2-way PS/DP	1 Nvidia K20X	17472	1.697e6
CoMet, Titan	2-way CCC	1 Nvidia K20X	17955	9.108e6
CoMet, Summit	2-way PS/SP	6 Nvidia V100	1000	21.643e6
CoMet, Summit	2-way PS/DP	6 Nvidia V100	1000	6.488e6
CoMet, Summit	2-way CCC	6 Nvidia V100	1000	20.337e6
CoMet, Summit	2-way CCC/sp	6 Nvidia V100	1000	12.764e6
CoMet, Summit	2-way PS/SP	6 Nvidia V100	*4608*	*99.731e6*
CoMet, Summit	2-way PS/DP	6 Nvidia V100	*4608*	*29.896e6*
CoMet, Summit	2-way CCC	6 Nvidia V100	*4608*	*93.711e6*
CoMet, Summit	2-way CCC/sp	6 Nvidia V100	*4608*	*58.815e6*

Related work: 3-way methods

Few comparable methods in the literature

@ 1000 Summit nodes, 2-way CCC is **18,493X** faster than best competing result

Full Summit: estimate **85,429X** faster

code	problem	node config	nodes used	cmp/sec ($\times 10^9$)
GPU3SNP[35]	3-way GWAS	4 Nvidia GTX/Titan	1	264.7
CoMet, Titan	3-way PS/SP	1 Nvidia K20X	18424	5.695e6
CoMet, Titan	3-way PS/DP	1 Nvidia K20X	18424	2.445e6
CoMet, Titan	3-way CCC	1 Nvidia K20X	18424	2.058e6
CoMet, Summit	3-way PS/SP	6 Nvidia V100	998	17.458e6
CoMet, Summit	3-way PS/DP	6 Nvidia V100	998	5.644e6
CoMet, Summit	3-way CCC	6 Nvidia V100	998	4.895e6
CoMet, Summit	3-way CCC/sp	6 Nvidia V100	998	4.058e6
CoMet, Summit	3-way PS/SP	6 Nvidia V100	*4608*	*80.663e6*
CoMet, Summit	3-way PS/DP	6 Nvidia V100	*4608*	*26.079e6*
CoMet, Summit	3-way CCC	6 Nvidia V100	*4608*	*22.613e6*
CoMet, Summit	3-way CCC/sp	6 Nvidia V100	*4608*	*18.747e6*

Conclusions

- CoMet running on Summit represents *3 to 4 orders of magnitude improvement* over current state of the art in comparative genomics metrics calculations
- The code achieves a high fraction of peak attainable performance on the GPU and gives near-perfect weak scaling on Summit
- This will enable analysis of very large datasets that could not be analyzed before, e.g., 10 million SNPs, 4 million population size

References

- W. Joubert, J. Nance, D. Weighill, D. Jacobson, “Parallel Accelerated Vector Similarity Calculations for Genomics Applications,” arxiv 1705.08210 [cs], *Parallel Computing,* 2018.
- W. Joubert, J. Nance, S. Climer, D. Weighill, D. Jacobson, “Parallel Accelerated Custom Correlation Coefficient Calculations for Genomics Applications,” arxiv 1705.08213 [cs], *Parallel Computing,* in review.

Questions?

Wayne Joubert
joubert@ornl.gov

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.