A Tropical Semiring Multiple Matrix-Product Library on GPUs: (not just) a step towards RNA-RNA Interaction Computations

HiCOMB 2020
19th IEEE International Workshop on High Performance Computational Biology

Brandon Gildemaster
brandon.gildemaster@colostate.edu

Prerana Ghalsasi
prerana.ghalsasi@colostate.edu

Sanjay Rajopadhye
sanjay.rajopadhye@colostate.edu

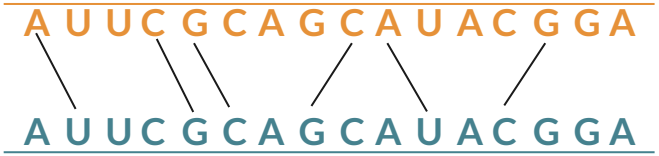
Colorado State University

Overview

- Background / motivation
- Algorithm
- Parallelization
- Memory optimizations
- GPU matrix-matrix multiplication library
- Modified matrix-matrix multiplication library
- Performance results
- Next steps

Background / Motivation

- RNA-RNA Interaction (RRI) plays an important role in biological processes
  - Gene expression
- Certain classes of RRI are well studied
  - Shown to play roles in various diseases
  - Other classes are not as well studied
- Biological function can be interpreted from interaction structure
- **Problem:** Current tools to predict structure are slow
  - O(N^4) space and O(N^6) time complexity
- **Goal:** Utilize massive parallelism of GPUs for acceleration while managing memory constraints



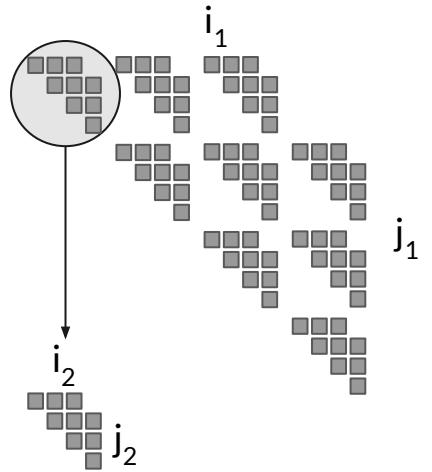
Algorithms

- Base pair maximization and free energy minimization
- O(N)^6 time and O(N)^4 space
- piRNA, BPPart, BPMax
- Much work on single strand folding, little on RRI

Algorithm

$$F_{i_1,j_1,i_2,j_2} = \begin{cases} S^{(2)}_{i_2,j_2} & j_1 < i_1 \\ S^{(1)}_{i_1,j_1} & j_2 < i_2 \\ \text{iscore}(i_1, i_2) & i_1 = j_1 \; and \; i_2 = j_2 \\ \max(F_{i+1,j-1,i_2,j_2} + score(i_1, j_1), & \\ \quad F_{i_1,j_1,i_2+1,j_2-1} + score(i_2, j_2), & \\ \quad H^{(1)}_{i_1,j_1,i_2,j_2}) & \text{otherwise} \end{cases}$$

$$H^{(1)}_{i_1,j_1,i_2,j_2} = \max \begin{pmatrix} \max_{k_1=i_1}^{j_1-1} \; \max_{k_2=i_2}^{j_2} (\mathrm{F}_{i_1,k_1,i_2,k_2} + \mathrm{F}_{k_1+1,j_1,k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2-1} (\mathrm{F}_{i_1,j_1,i_2,k_2} + \mathrm{S}^{(2)}_{k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2} (\mathrm{S}^{(2)}_{i_2,k_2} + \mathrm{F}_{i_1,j_1,k_2+1,j_2}), \\ \max_{k_1=i_1}^{j_1} (\mathrm{S}^{(1)}_{i_1,k_1} + \mathrm{F}_{k_1+1,j_1,i_2,j_2}) \end{pmatrix}$$

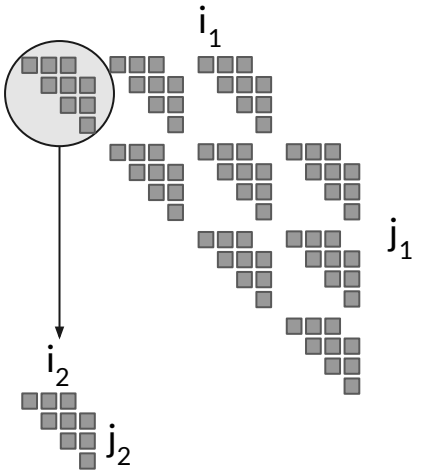


Algorithm

- BPMax
  - Maximizes the score of weighted interactions
  - Restricts certain structures
- Fills up 4D dynamic programming table
  - Trapezoidal grid of trapezoids
- Full recurrence equation is complex
  - One O(N^6) term
  - Several O(N^5) terms and constant lookups
- Double max reduction (boxed in red) is the most dominant O(N^6) term
  - Most important optimization for performance

$$F_{i_1,j_1,i_2,j_2} = \begin{cases} S^{(2)}_{i_2,j_2} & j_1 < i_1 \\ S^{(1)}_{i_1,j_1} & j_2 < i_2 \\ \text{iscore}(i_1,i_2) & i_1 = j_1 \text{ and } i_2 = j_2 \\ \max(F_{i+1,j-1,i_2,j_2} + score(i_1,j_1), & \\ \quad F_{i_1,j_1,i_2+1,j_2-1} + score(i_2,j_2), & \\ \quad H^{(1)}_{i_1,j_1,i_2,j_2}) & \text{otherwise} \end{cases}$$

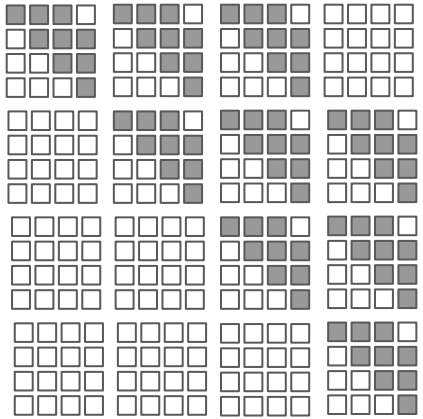
$$H^{(1)}_{i_1,j_1,i_2,j_2} = \max \begin{pmatrix} \max_{k_1=i_1}^{j_1-1} \max_{k_2=i_2}^{j_2} (\mathrm{F}_{i_1,k_1,i_2,k_2} + \mathrm{F}_{k_1+1,j_1,k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2-1} (\mathrm{F}_{i_1,j_1,i_2,k_2} + \mathrm{S}^{(2)}_{k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2} (\mathrm{S}^{(2)}_{i_2,k_2} + \mathrm{F}_{i_1,j_1,k_2+1,j_2}), \\ \max_{k_1=i_1}^{j_1} (\mathrm{S}^{(1)}_{i_1,k_1} + \mathrm{F}_{k_1+1,j_1,i_2,j_2}) \end{pmatrix}$$



Algorithm

- Skip bottom half of each matrix
  - Subsequence [i,j] is the same as subsequence [j,i]
- Top right corner also can be skipped
  - Controlled by window size
  - Limits range of intra-RNA interaction

Memory space



Set of points evaluated

Parallelization

$$H^{(1)}_{i_1,j_1,i_2,j_2} = \max \begin{pmatrix} \max_{k_1=i_1}^{j_1-1} \max_{k_2=i_2}^{j_2} (\mathrm{F}_{i_1,k_1,i_2,k_2} + \mathrm{F}_{k_1+1,j_1,k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2-1} (\mathrm{F}_{i_1,j_1,i_2,k_2} + \mathrm{S}^{(2)}_{k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2} (\mathrm{S}^{(2)}_{i_2,k_2} + \mathrm{F}_{i_1,j_1,k_2+1,j_2}), \\ \max_{k_1=i_1}^{j_1} (\mathrm{S}^{(1)}_{i_1,k_1} + \mathrm{F}_{k_1+1,j_1,i_2,j_2}) \end{pmatrix}$$

- Imbalanced workload
- Naive parallelization: all points along a diagonal can be computed in parallel
  - Poor locality
  - No optimizations such as vectorization
- **Key insight:** The double max reduction can be cast as specialized matrix-matrix multiplication
  - Rearrange order of evaluation
  - Apply memory transformations to the dynamic programming table

Depiction of naive parallelization: all terms for the red cells are evaluated in parallel

Double max reduction

$$H^{(1)}_{i_1,j_1,i_2,j_2} = \max \begin{pmatrix} \max_{k_1=i_1}^{j_1-1} \max_{k_2=i_2}^{j_2} (\mathrm{F}_{i_1,k_1,i_2,k_2} + \mathrm{F}_{k_1+1,j_1,k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2-1} (\mathrm{F}_{i_1,j_1,i_2,k_2} + \mathrm{S}^{(2)}_{k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2} (\mathrm{S}^{(2)}_{i_2,k_2} + \mathrm{F}_{i_1,j_1,k_2+1,j_2}), \\ \max_{k_1=i_1}^{j_1} (\mathrm{S}^{(1)}_{i_1,k_1} + \mathrm{F}_{k_1+1,j_1,i_2,j_2}) \end{pmatrix}$$

Double max reduction

$$H^{(1)}_{i_1,j_1,i_2,j_2} = \max \begin{pmatrix} \max_{k_1=i_1}^{j_1-1} \max_{k_2=i_2}^{j_2} (\mathrm{F}_{i_1,k_1,i_2,k_2} + \mathrm{F}_{k_1+1,j_1,k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2-1} (\mathrm{F}_{i_1,j_1,i_2,k_2} + \mathrm{S}^{(2)}_{k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2} (\mathrm{S}^{(2)}_{i_2,k_2} + \mathrm{F}_{i_1,j_1,k_2+1,j_2}), \\ \max_{k_1=i_1}^{j_1} (\mathrm{S}^{(1)}_{i_1,k_1} + \mathrm{F}_{k_1+1,j_1,i_2,j_2}) \end{pmatrix}$$

Double max reduction

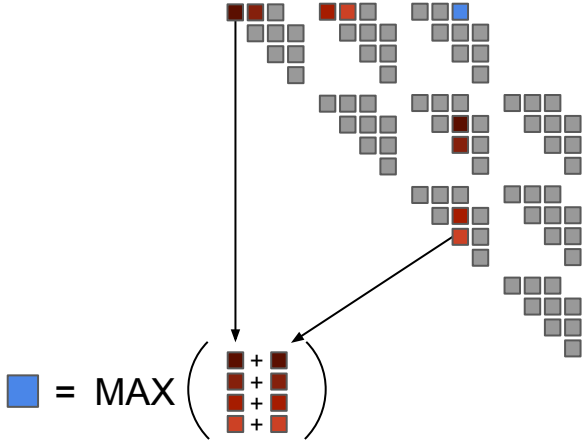
$$H^{(1)}_{i_1,j_1,i_2,j_2} = \max \begin{pmatrix} \max_{k_1=i_1}^{j_1-1} \max_{k_2=i_2}^{j_2} (\mathrm{F}_{i_1,k_1,i_2,k_2} + \mathrm{F}_{k_1+1,j_1,k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2-1} (\mathrm{F}_{i_1,j_1,i_2,k_2} + \mathrm{S}^{(2)}_{k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2} (\mathrm{S}^{(2)}_{i_2,k_2} + \mathrm{F}_{i_1,j_1,k_2+1,j_2}), \\ \max_{k_1=i_1}^{j_1} (\mathrm{S}^{(1)}_{i_1,k_1} + \mathrm{F}_{k_1+1,j_1,i_2,j_2}) \end{pmatrix}$$

Double max reduction

$$H^{(1)}_{i_1,j_1,i_2,j_2} = \max \begin{pmatrix} \max_{k_1=i_1}^{j_1-1} \max_{k_2=i_2}^{j_2} (\mathrm{F}_{i_1,k_1,i_2,k_2} + \mathrm{F}_{k_1+1,j_1,k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2-1} (\mathrm{F}_{i_1,j_1,i_2,k_2} + \mathrm{S}^{(2)}_{k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2} (\mathrm{S}^{(2)}_{i_2,k_2} + \mathrm{F}_{i_1,j_1,k_2+1,j_2}), \\ \max_{k_1=i_1}^{j_1} (\mathrm{S}^{(1)}_{i_1,k_1} + \mathrm{F}_{k_1+1,j_1,i_2,j_2}) \end{pmatrix}$$

Double max reduction

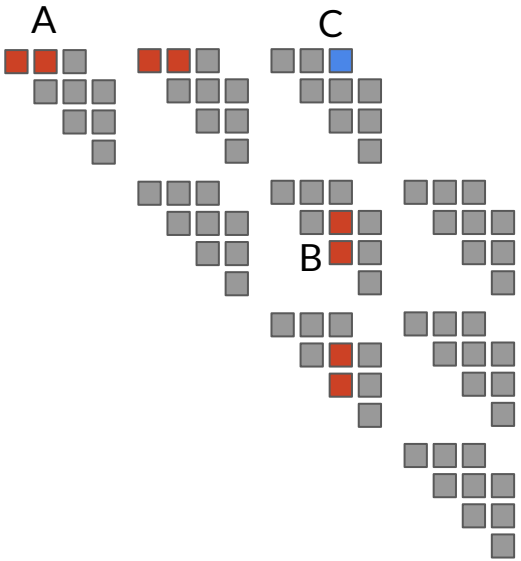
$$H^{(1)}_{i_1,j_1,i_2,j_2} = \max \begin{pmatrix} \max_{k_1=i_1}^{j_1-1} \max_{k_2=i_2}^{j_2} (\mathrm{F}_{i_1,k_1,i_2,k_2} + \mathrm{F}_{k_1+1,j_1,k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2-1} (\mathrm{F}_{i_1,j_1,i_2,k_2} + \mathrm{S}^{(2)}_{k_2+1,j_2}), \\ \max_{k_2=i_2}^{j_2} (\mathrm{S}^{(2)}_{i_2,k_2} + \mathrm{F}_{i_1,j_1,k_2+1,j_2}), \\ \max_{k_1=i_1}^{j_1} (\mathrm{S}^{(1)}_{i_1,k_1} + \mathrm{F}_{k_1+1,j_1,i_2,j_2}) \end{pmatrix}$$

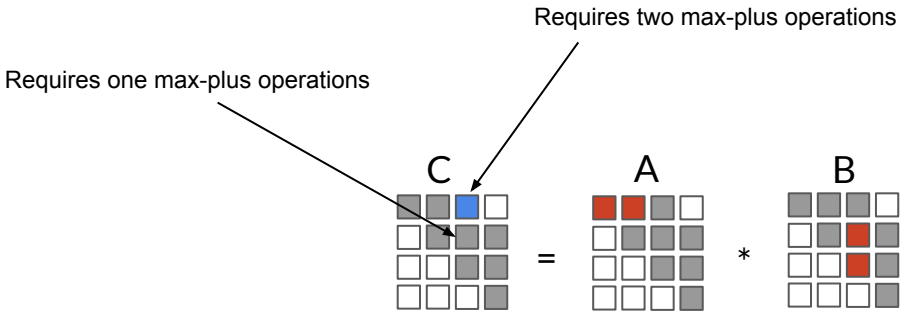


- Evaluation of blue cell is the maximum of the pairwise addition of the row and column of red cells
- Interchanging j and k loops exploits vectorization on CPUs
  - Basically doing tropical matrix multiplication
- Can be applied to all points in one matrix in parallel
  - And all matrices along a diagonal to exploit coarse grain parallelism

Double max reduction

- Imbalanced workload

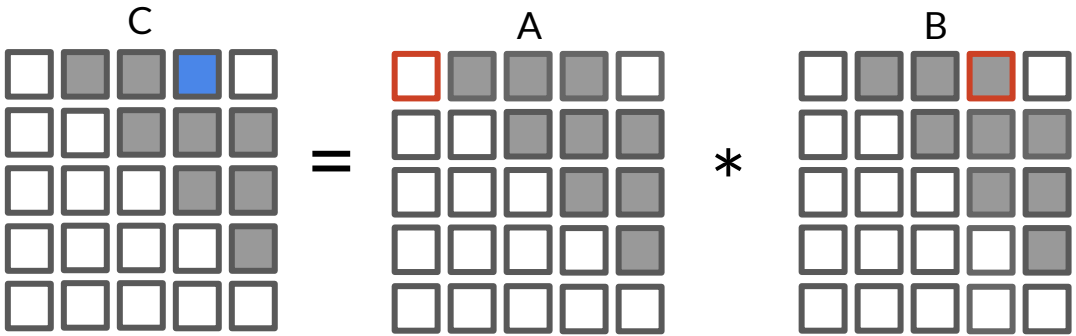




Double max reduction

- Pad each matrix with an extra row and column
  - Shift cells in each matrix one row to the right
- Initialize white cells to max-plus semiring additive identity
- Avoids thread divergence

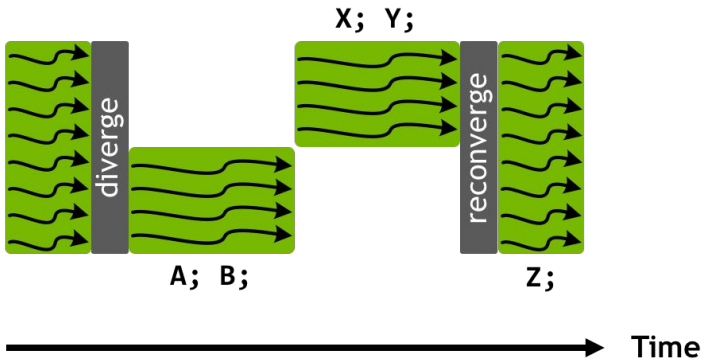
$$\text{MAX}(C[0,3] , -\infty + B[0,3]) = C[0,3]$$



Thread divergence

- One program counter (PC) per thread warp
- PC loads instruction and all threads execute it
- Divergence introduces overhead
  - Threads must be masked (basically turned on/off)

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



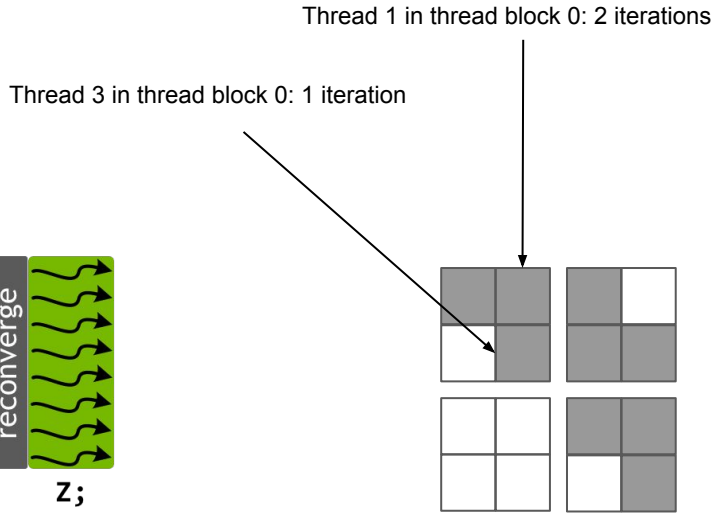
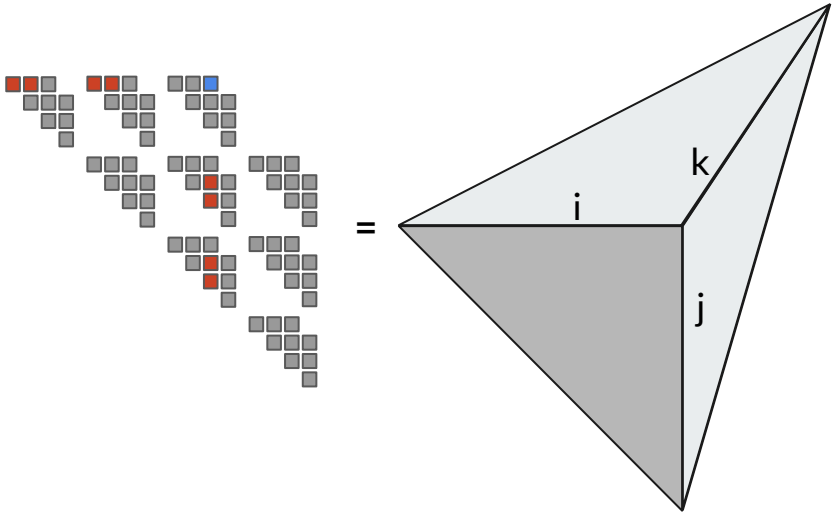
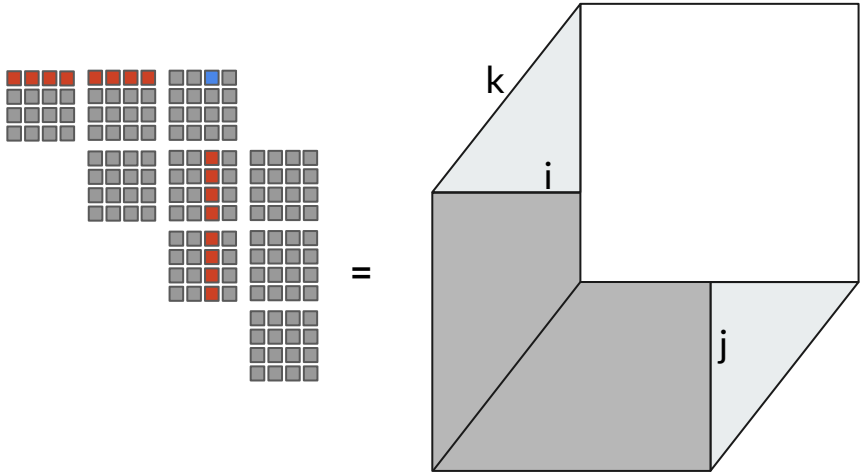


Image from NVIDIA Volta architecture whitepaper

Matrix Multiplication

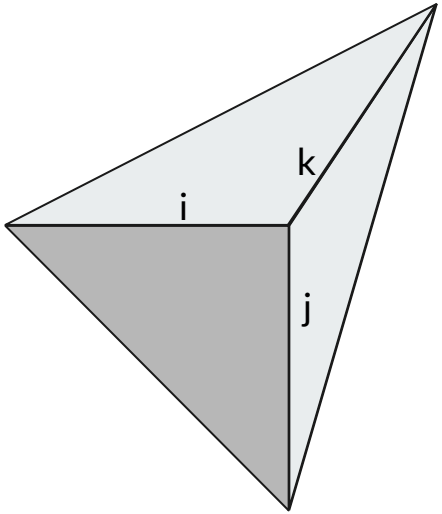
- Visualizing iteration space

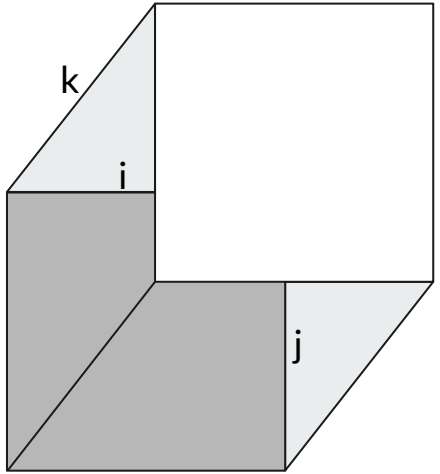




Triangular or Trapezoidal Matrix Multiplication

- **Goal:** Get as close to the iteration space on the left without introducing thread divergence
- Thread divergence happens at the warp level in CUDA
  - Diverging threads in a warp execute different instructions
- Skip computations at the thread-block level
- No standard library performs triangular-triangular matrix multiplication
  - Triangular-square

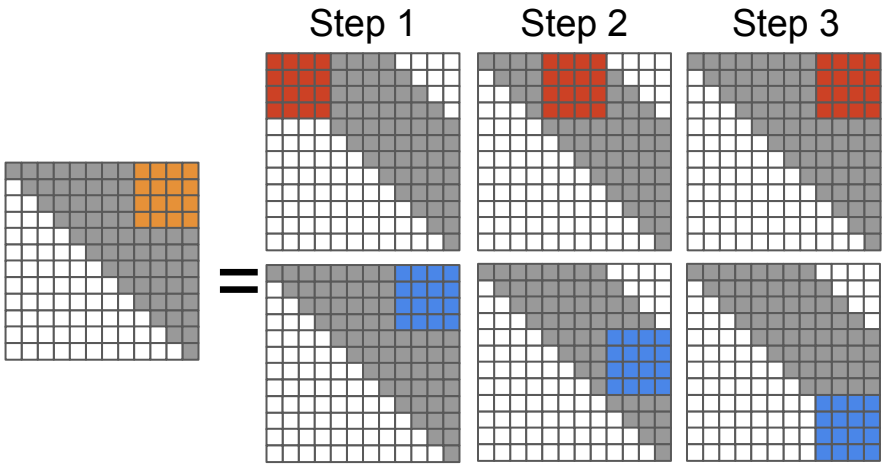


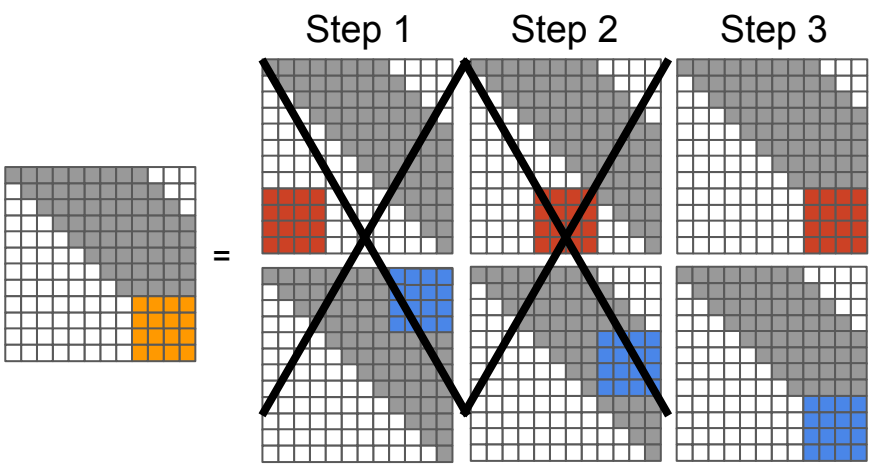


6x the amount of work!

Algorithm

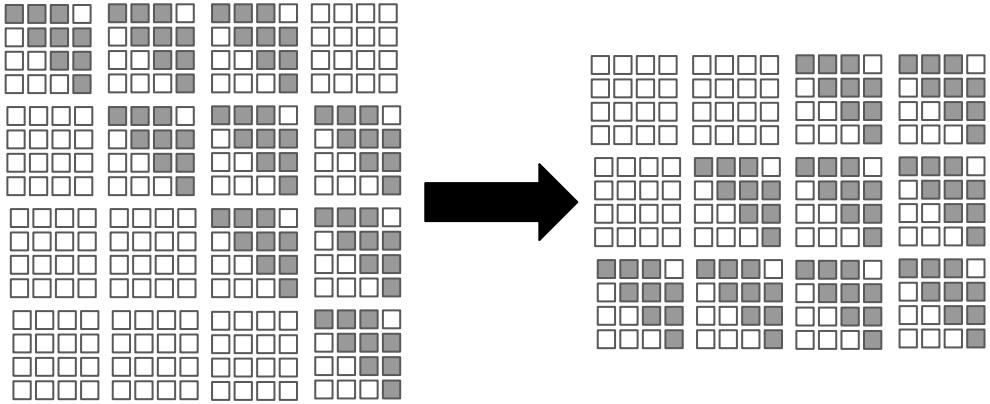
- Skip computations at thread block level



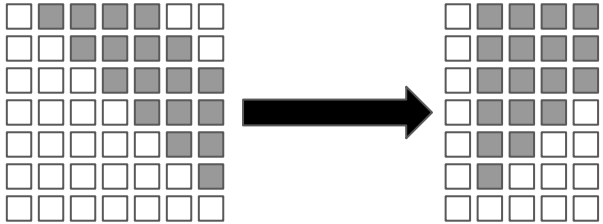


Modifications

- Two memory transformations
- $N^2*M^2 \rightarrow N*M*W^2$
- 102 GB → 10.5 GB for N = M = 400 and W = 128

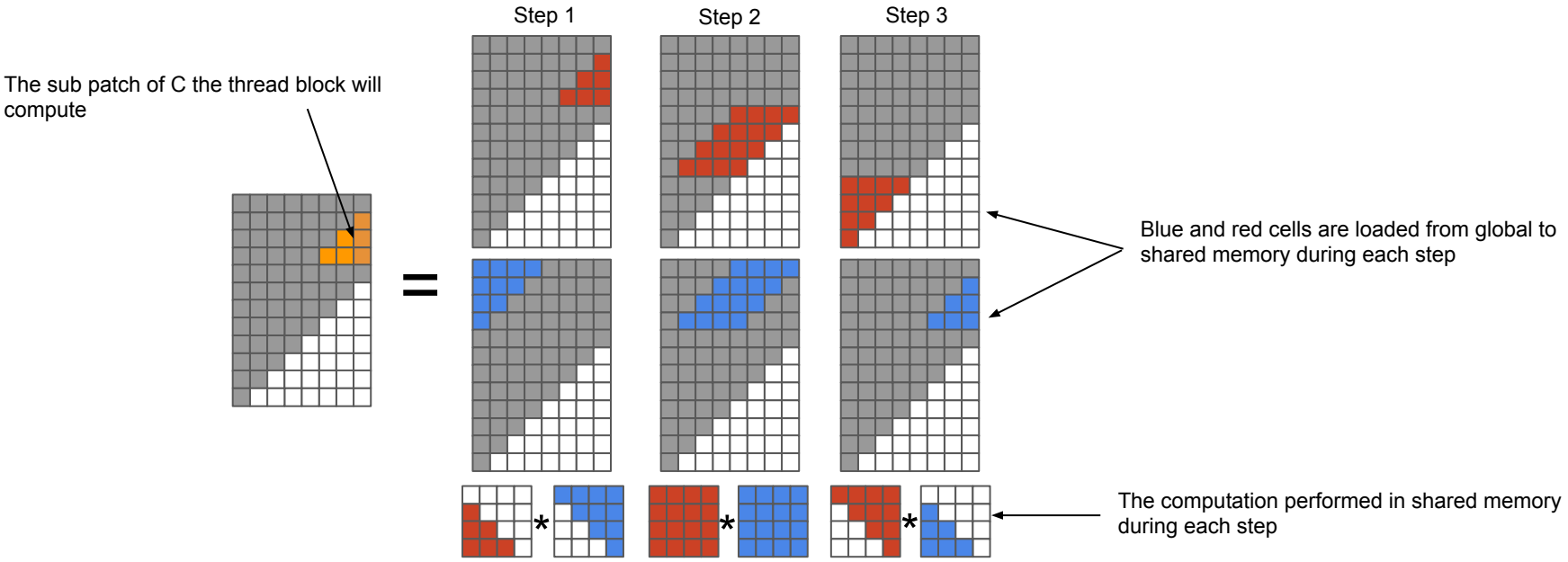


$$i_1, j_1 \rightarrow i_1 + N - j_1, j_1$$



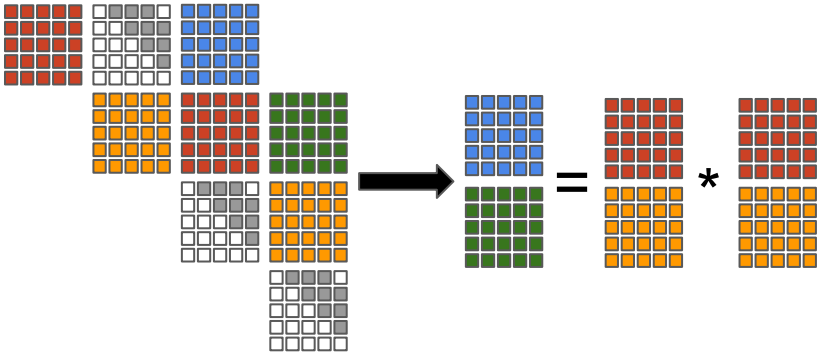
$$i_2, j_2 \rightarrow i_2, j_2 - i_2$$

Final algorithm

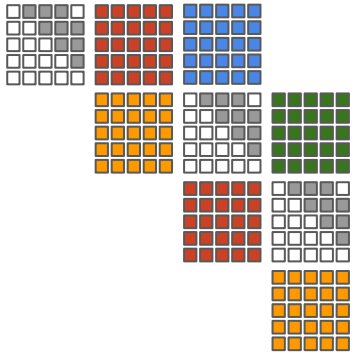


GPU Library

- Library call multiplies a column of matrices by another column of matrices in the max-plus semiring



One call to the GPU library



The full double reduction for blue/green matrices requires two library calls

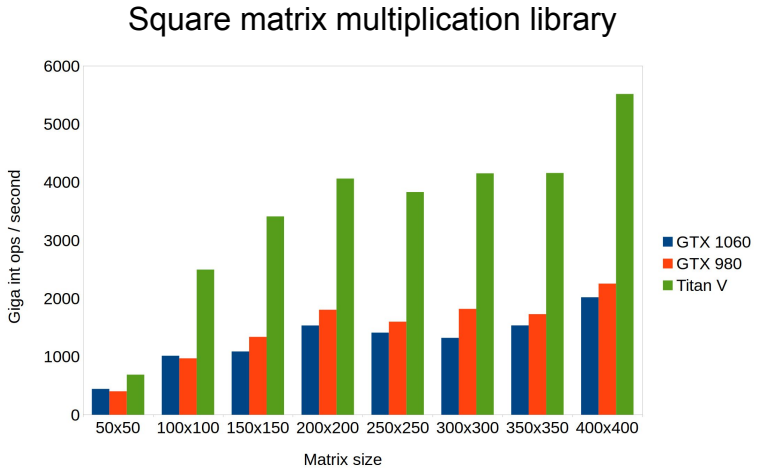
Max plus theoretical peak

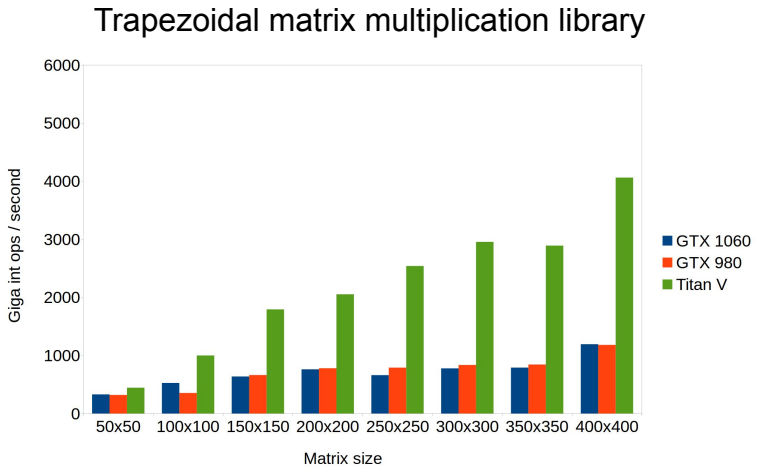
- Can’t utilize FMA or tensor cores

	Architecture	Memory	Cores	Clock speed	Calculated peak
GTX 980	Maxwell	4 GB	2048	1216 MHz	2490
GTX 1060	Pascal	6 GB	1280	1708 MHz	2184
Titan V	Volta	12 GB	5120	1455 MHz	7450

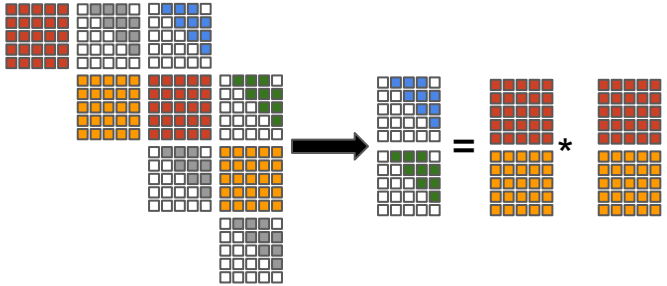
Library performance

- We developed a square matrix multiplication library which attains close to machine peak
  - Performs many unnecessary computations
- A trapezoidal matrix multiplication library which does less operations
  - but introduces some irregularities affecting performance
- Graphs showing performance of a single library call on a column of 50 matrices

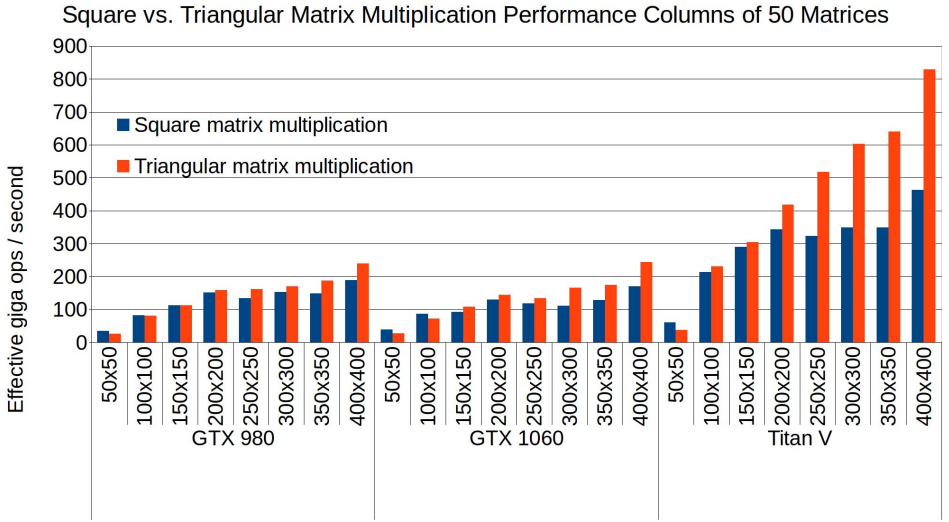




Library performance

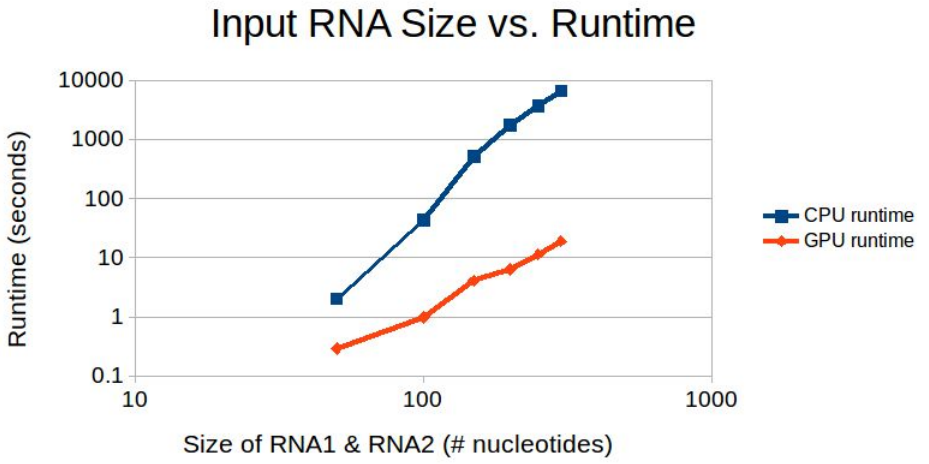


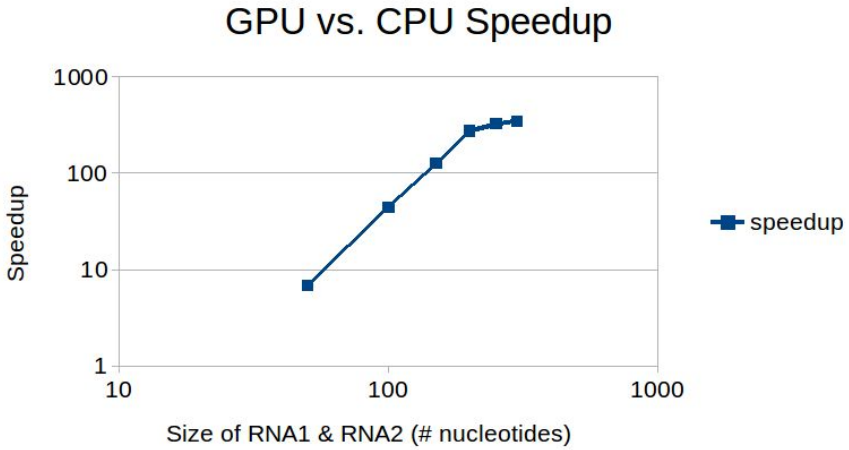
- Graph is showing effective operations per second: counting only the operations on cells that matter divided by runtime
- Previous graph was showing performance considering all operations
  - This graph is more specific to BPMax
- When computing operations per second and ignoring useless computations (effective ops/second) the trapezoidal library performance is higher
  - Because it is doing less operations



Full BPMax performance

- At the time of paper submission we completed the full implementation of BPMax on a GPU
- CPU experiments ran with the original BPMax implementation
  - Naive CPU implementation / parallelization
  - We plan to implement an optimized CPU version for a more fair comparison
- Intel(R) Xeon(R) E-2278G CPU
  - 5 GHz max clock speed
  - 16 cores
- GPU results include data transfer time from CPU to GPU and back
- BPMax attains ~.5 Giga ops /second currently



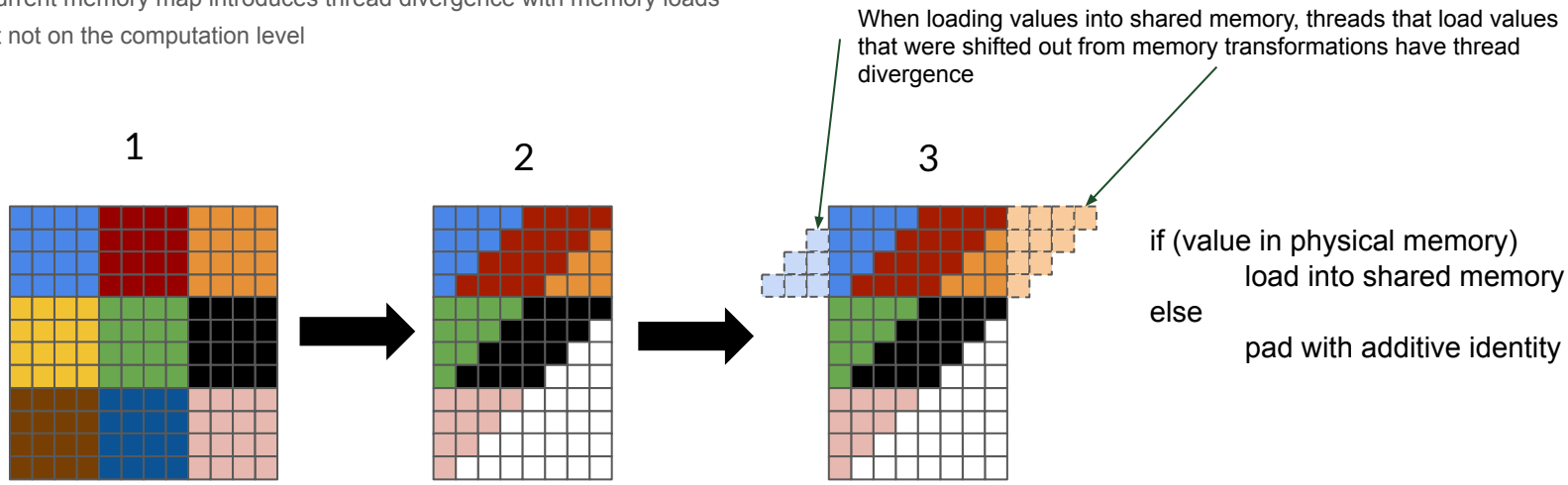


Current / future work

- Current library call attains ~10-11% of theoretical peak of GPU across 3 architectures
  - Room for 10x improvement
- Bottleneck: Memory mappings we implemented introduce thread divergence with memory loads
  - We are exploring alternate strategies that reduce memory requirements without introducing irregularities
- Optimized CPU implementation of BPMax that exploits vectorization / multithreading

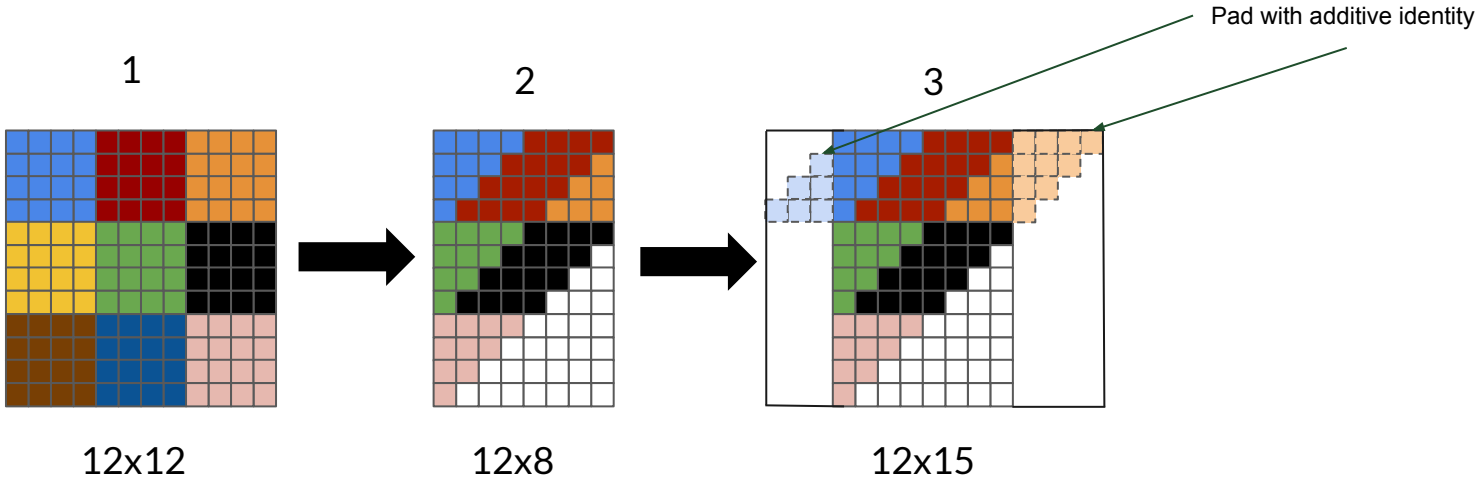
Current work - eliminating thread divergence with memory loads

- Problem: current memory map introduces thread divergence with memory loads
  - But not on the computation level



Current work - possible solution 1

- Pad each matrix out to the next multiple of the thread block dimensions
  - In this example the memory allocation is worse simply because the problem size is so small
  - For larger RNA / window sizes it will save memory and eliminate divergence



Current work - possible solution 2

- Allocate memory based on the dimensions of the thread blocks
- This is the minimum memory we can allocate while avoiding thread divergence
  - Since it is based off the thread block dimensions