Optimization for Performance and Energy for Batched Matrix Computations on GPUs

Piotr Luszczek
University of Tennessee Knoxville

Typical Batch Operation Scenario

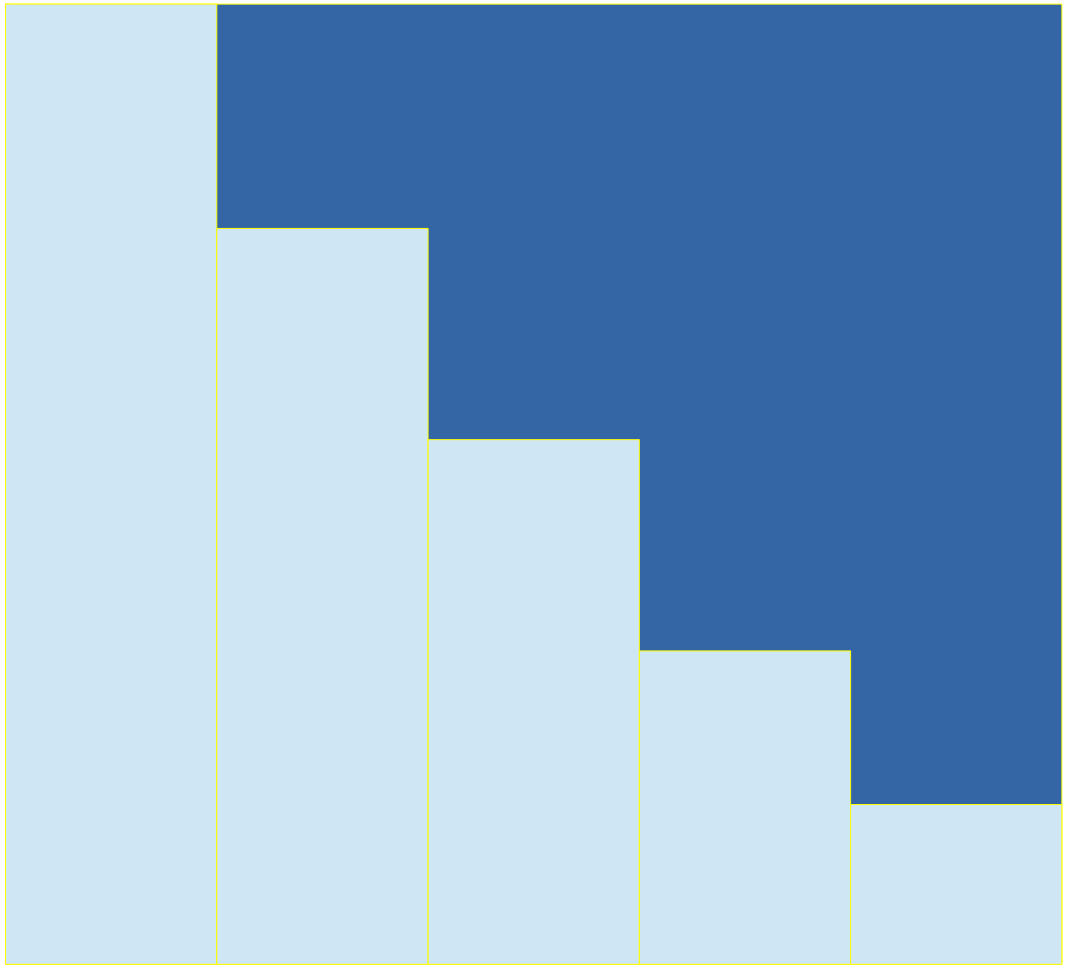
- A lot of small matrix problems available at once
  - `for A#i in [ A#1, A#2, …, A#n ]`
    `Generate(A#i)`
  - `for A#i in [ A#1, A#2, …, A#n ]`
    `Factorize(A#i)`
- Sources of batch computation
  - Astrophysics, Quantum Physics (Hall effect, tensor contraction)
  - Metabolic networks
  - Higher order FEM schemes and for hydrodynamics
  - Direct solvers for PDEs
  - Signal processing
  - Image processing

Potential Solutions for Batch Factorizations

- Library
  - The current (HPC) libraries will handle correctly small matrices
    - We know, we wrote some!
    - Often, small matrices won't even be sent to GPU.
    - APIs are oriented toward single large matrix.
  - New interfaces emerge
    - CUBLAS makes new batch routines available in each release, and
    - cuSolvers started addressing factorizations.
- Compiler
  - Trivial: only an extra outer loop over all batch calls
  - But inter-iteration optimization non-trivial
    - Needed: introspection into the batch routines (telescoping, etc.)
    - To in-line a factorization routine is a tall task as most factorization codes:
      - are not trivial
        - switch yards
        - clean-up code, …
      - call other routines, mostly proprietary

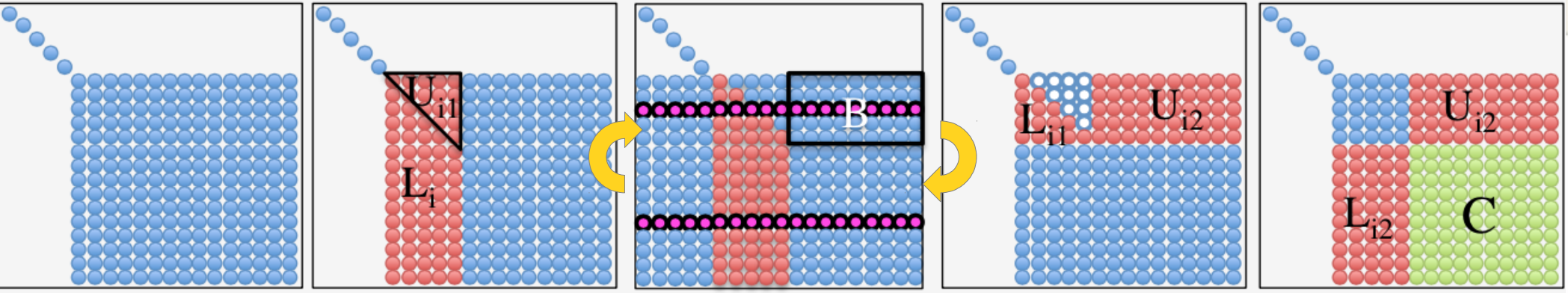
One-Sided Factorization: the Basics

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \rightarrow \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \times \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}$$



- Factorization loops over:
  - Panel factorization
    - Fits well on CPU
    - Latency-bound
      - Vector operations
      - Dependent computation
      - Bandwidth-sensitive
    - Few flop/s overall
  - Trailing matrix update
    - Fits well on GPU
    - Compute-bound
      - Matrix operations
      - Abundant parallelism
      - Good data reuse
    - Majority of flop/s

LU Factorization Details - DGETRF

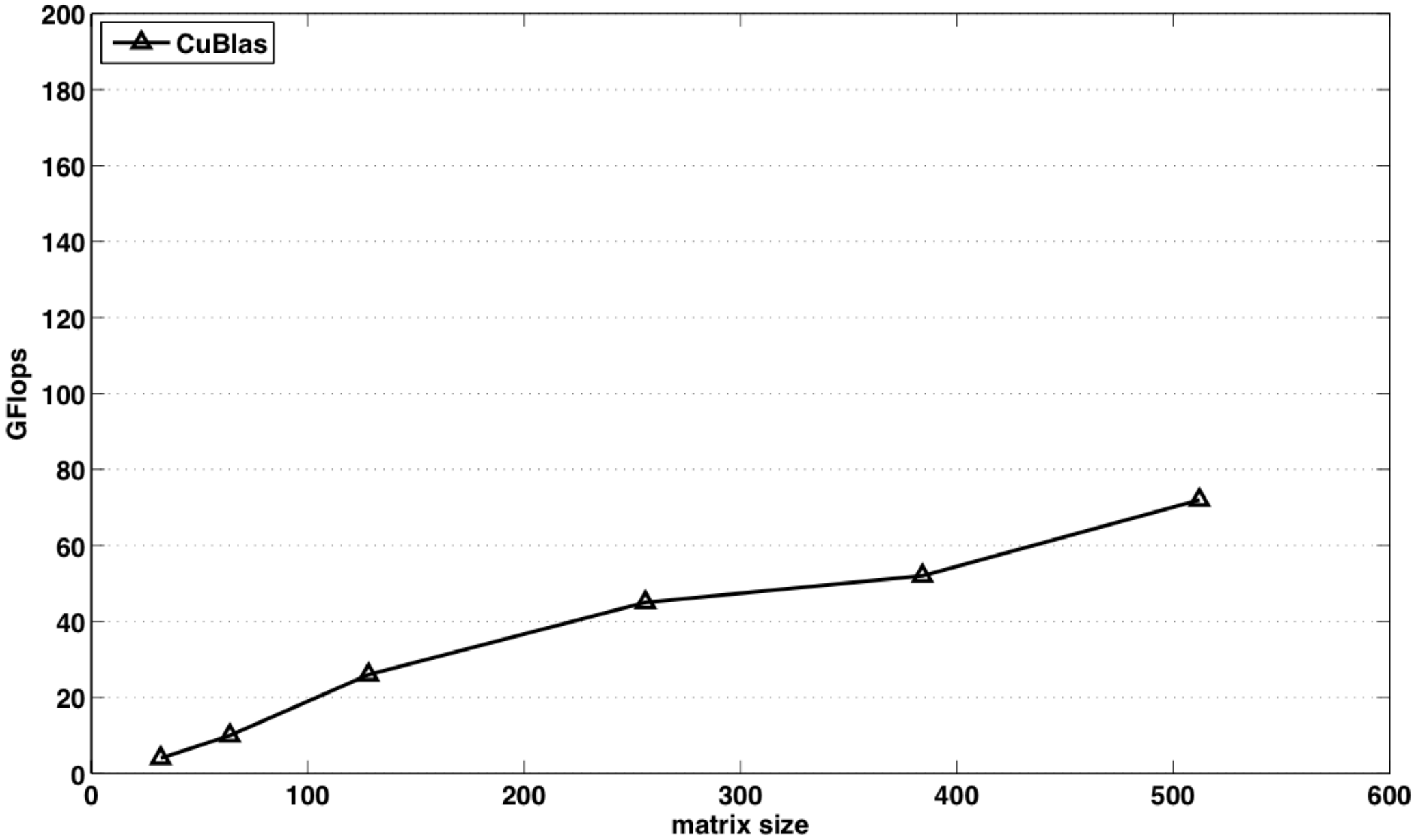


- **DGETF2** Panel factorization mostly sequential due to memory bottleneck
- **DTRSM** Triangular solve has little parallelism
- **DGEMM** Schur complement update is the only easy to parallelize task
- **DLASWP** Partial pivoting complicates things even further
- Bulk synchronous parallelism (fork-join) unless dynamic runtime is employed that provides dataflow scheduling

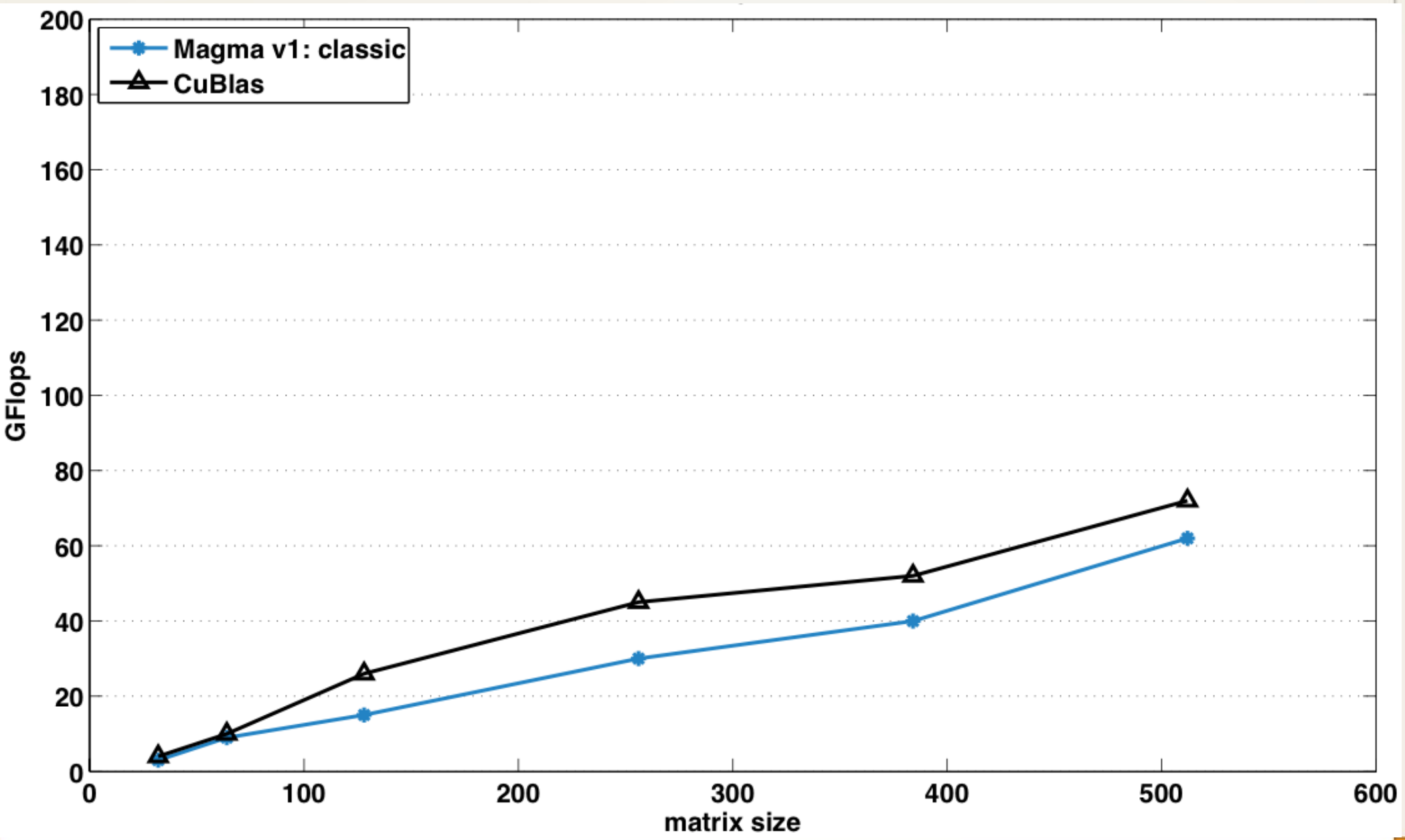
Optimization Summary

- **Cholesky factorization**
  - **Panel factorization**
    - **Vectors loaded to shared memory and reused**
    - **Custom kernels instead of NVIDIA CUBLAS**
      - **Replace DSYRK with DGEMM and extra cost absorbed by performance gains**
      - **Violate symmetry of storage in GPU memory and only preserve it in CPU memory**
- **LU factorization**
  - **Current column loaded to shared memory**
    - **Reused for MAX and SCALE operations**
  - **Explicitly invert diagonal blocks**
    - **Numerical stability becomes an issue**
      - **But it's already widely used on GPUs**
      - **Mostly stable due to pivoting**

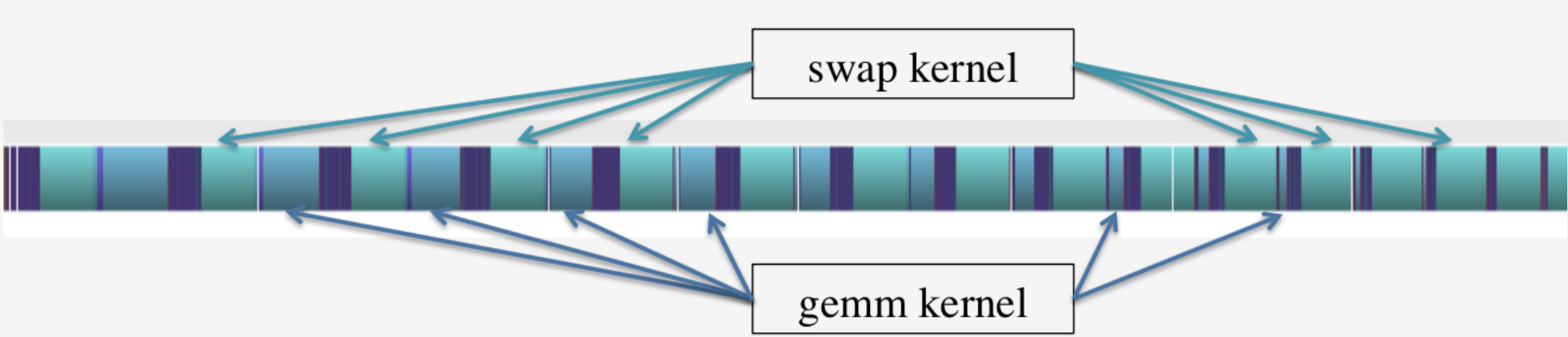
Performance: LU x 2000



Performance: LU x 2000

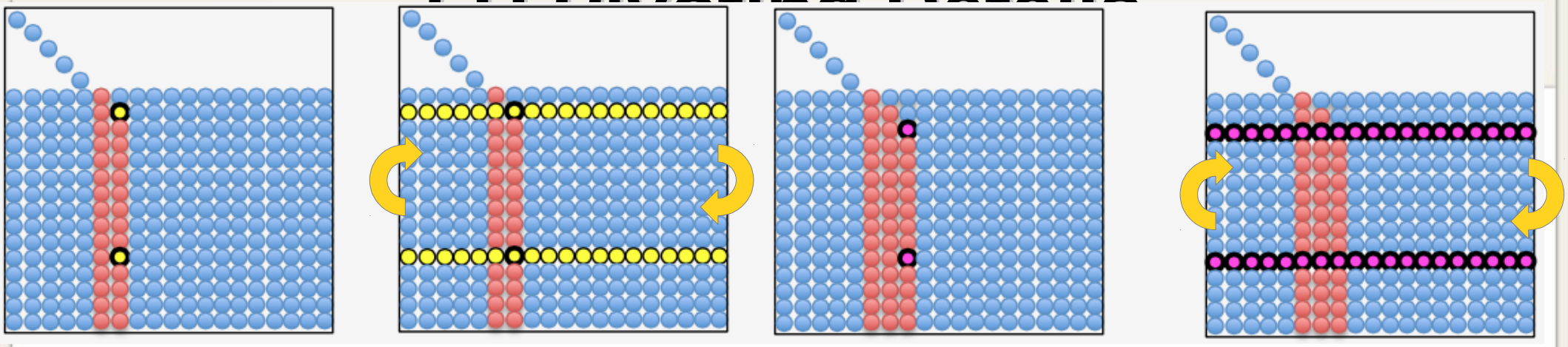


The Cost of Pivoting - DLASWP



Swap is expensive: it consumes around 60% of the total time.

LU Pivoting Details



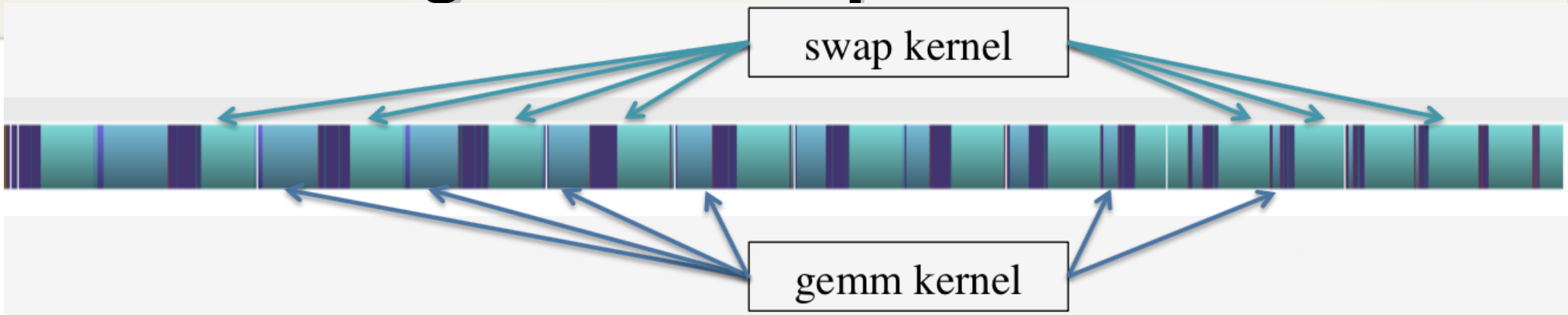
Problems and bottlenecks:

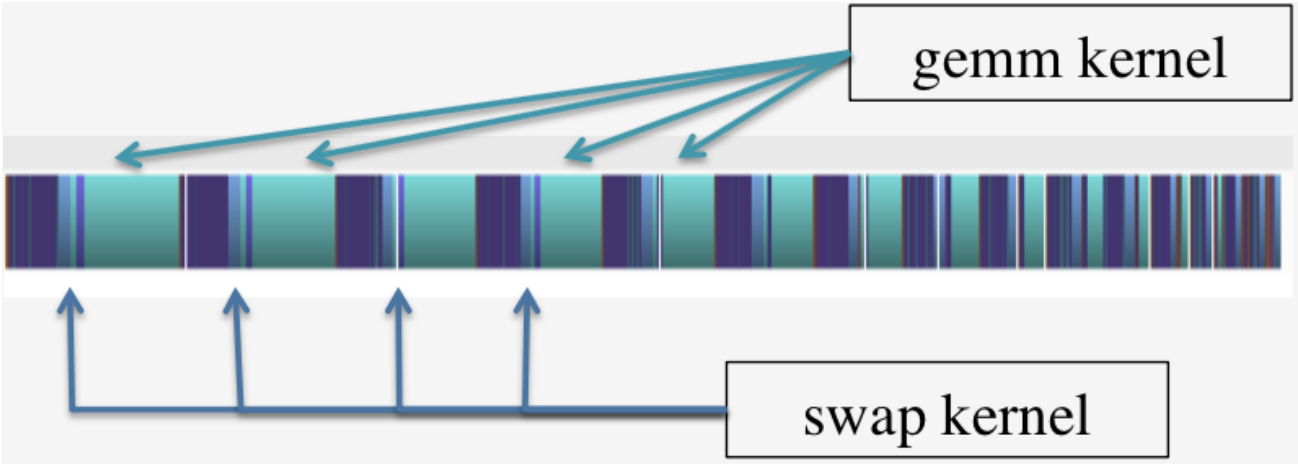
- Swap is sequential and data dependent.
- Data is not coalescent: a GPU warp cannot read 32 values at the same time unless matrix is stored in transposed form. However, if matrix is stored in transposed form the swap is fast BUT the other components becomes very slow.

Solutions:

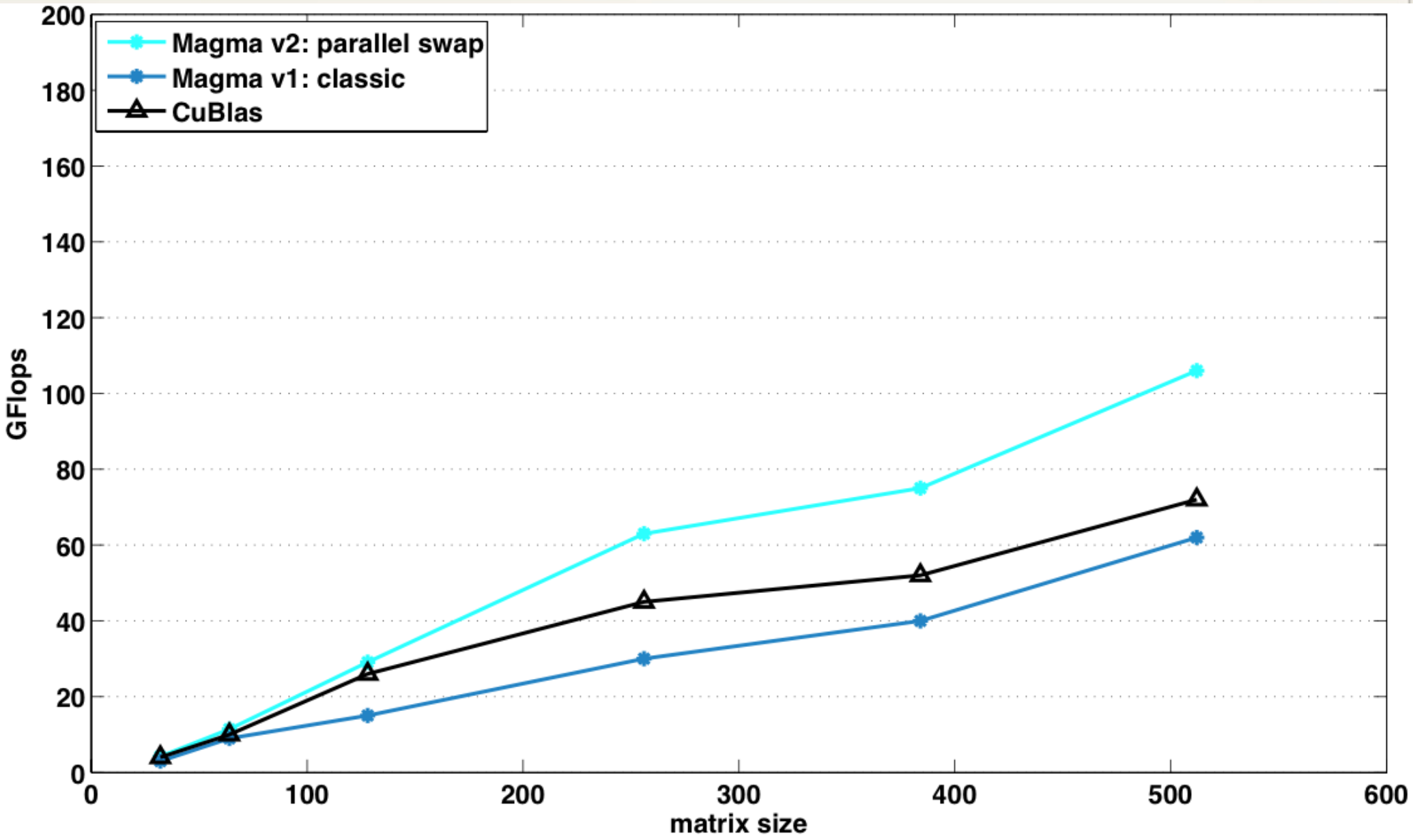
- Develop a parallel swap and new permutation representation.
- Improve the write-back of the swapped rows as the data is now coalescent.

LU Pivoting: from Sequential to Parallel

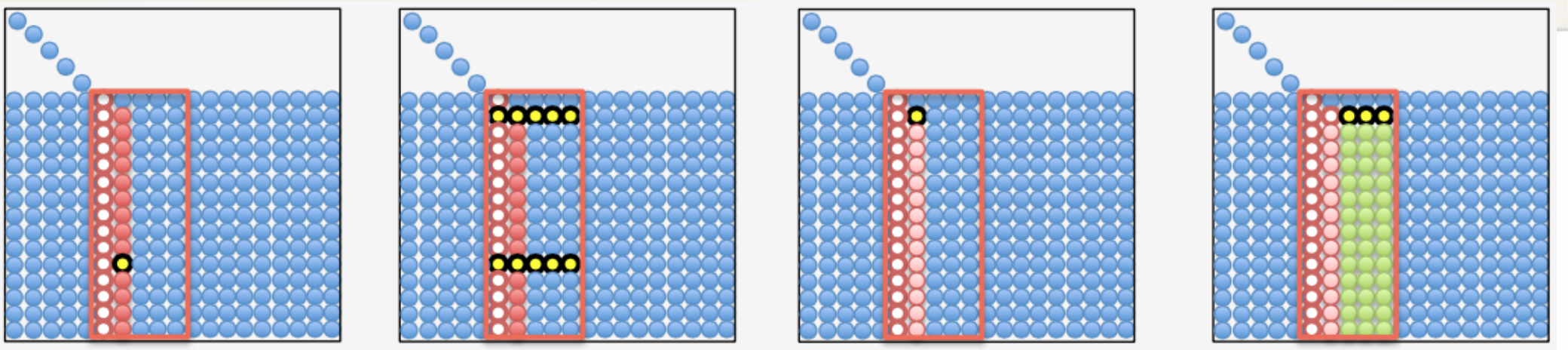




Performance: LU x 2000



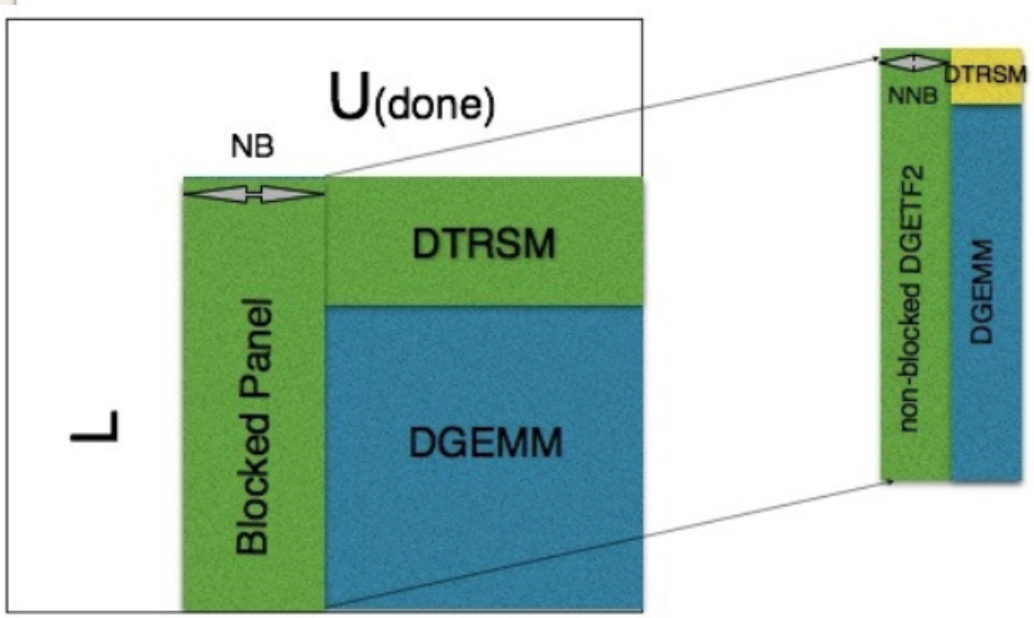
LU Panel Factorization Details - DGETF2



1) Find the max absolute value for the current column below the diagonal – the "pivots"
2) Swap the row of size **nb** `DSWAP`
3) Scale the column below the diagonal by the inverse of the pivots `DSCAL`
4) Update the panel to the right of the current column `DGER`

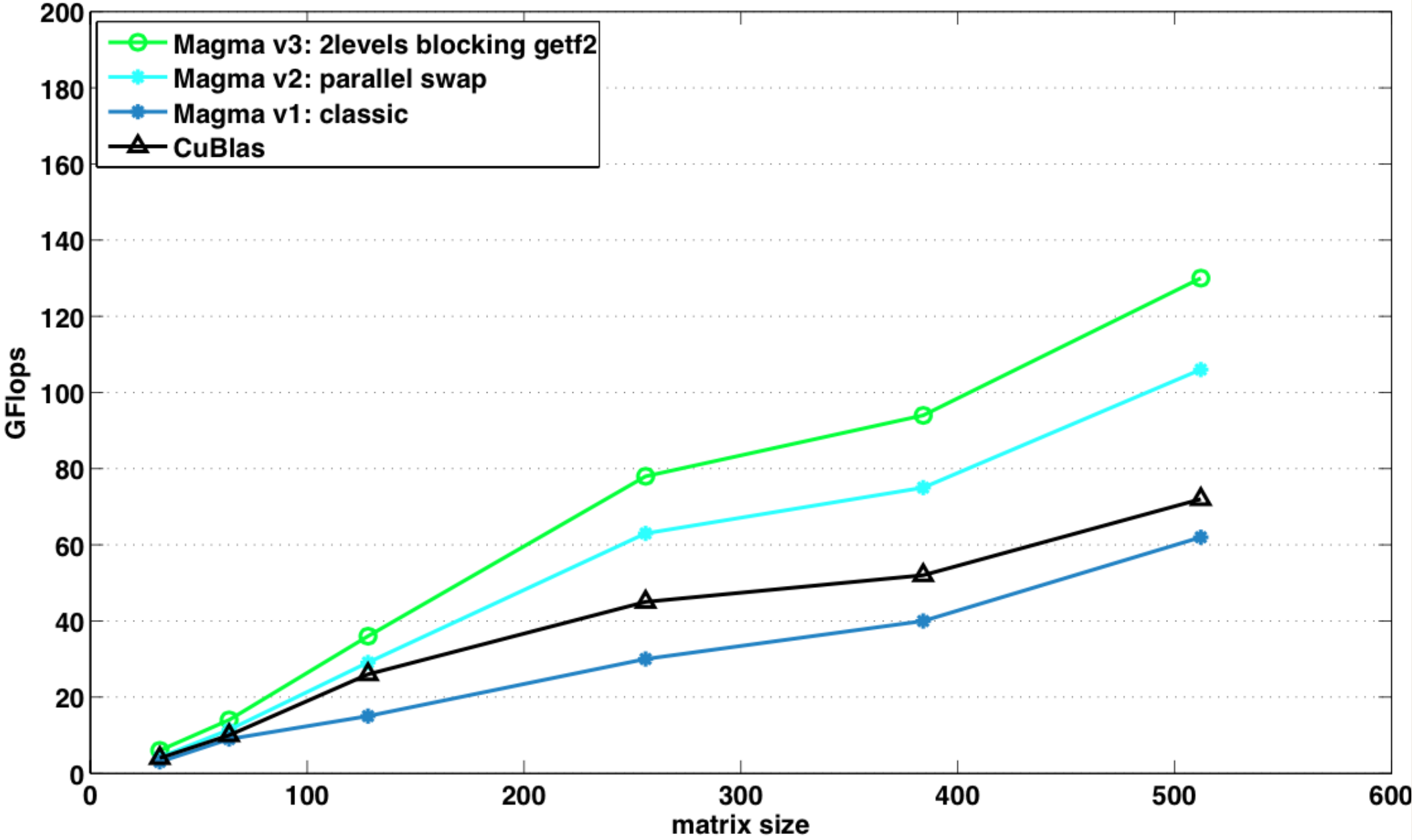
`DGETF2` consumes 30% of the time

Nested Blocking for LU Panel Factorization

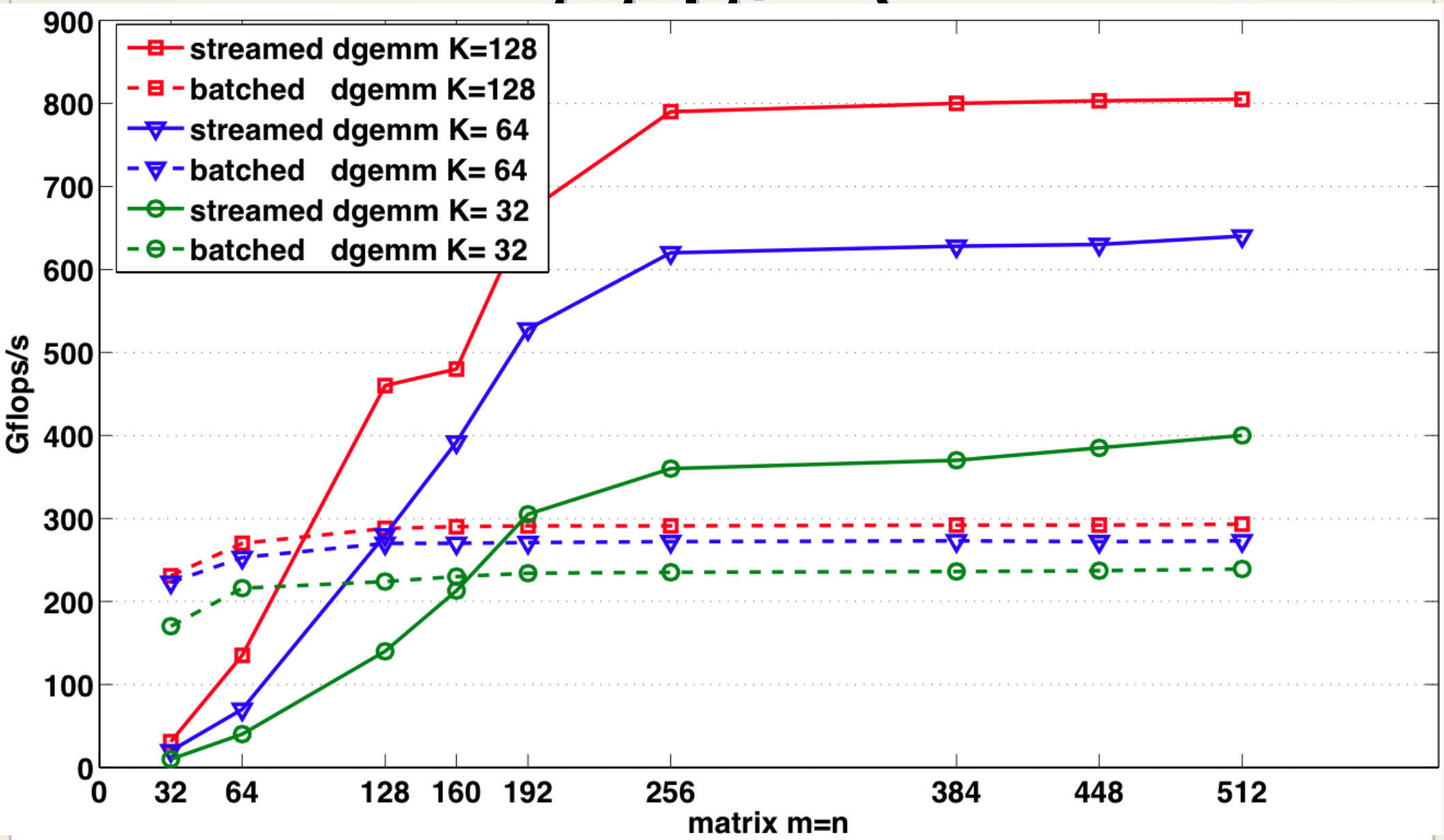


- Use nested blocking to factorize the panel
- Allows to replace the `DGER` kernel by `DGEMM` kernel.
- Reduces panel time from 30% to 8% of the total time.

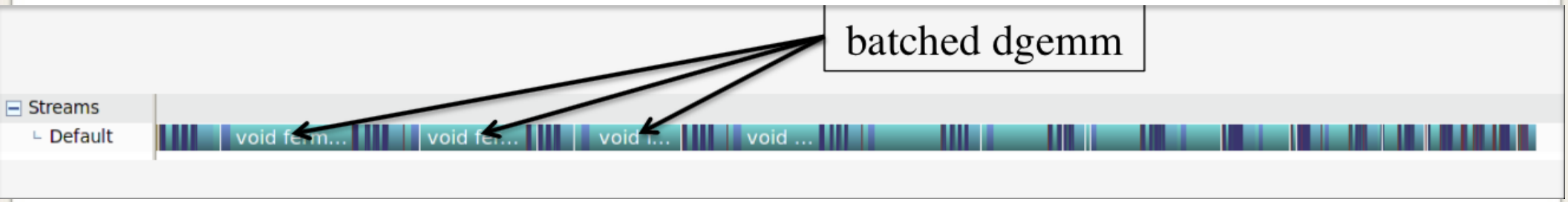
Performance: LU x 2000

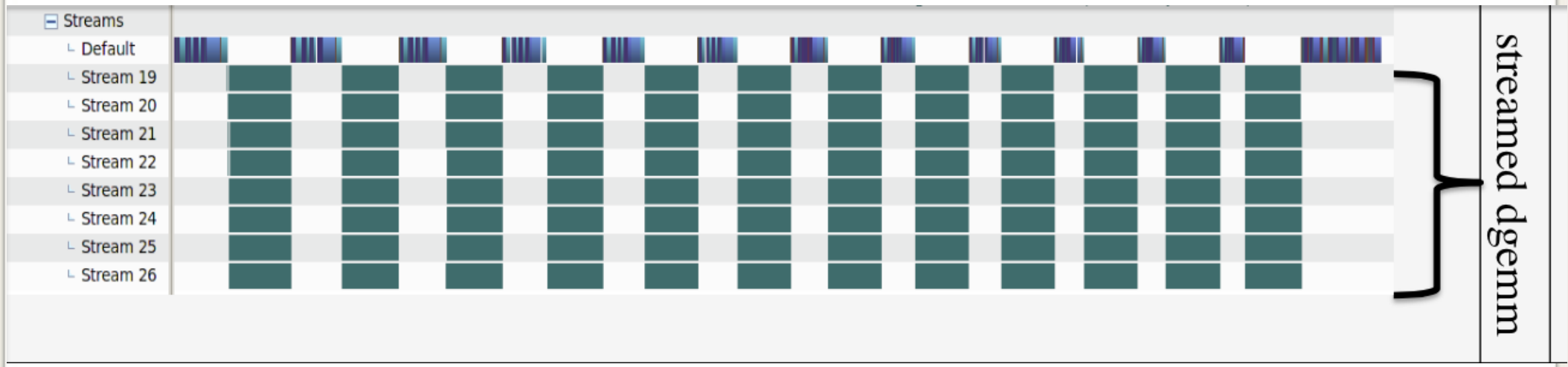


Performance of DGEMM Variants (60% of

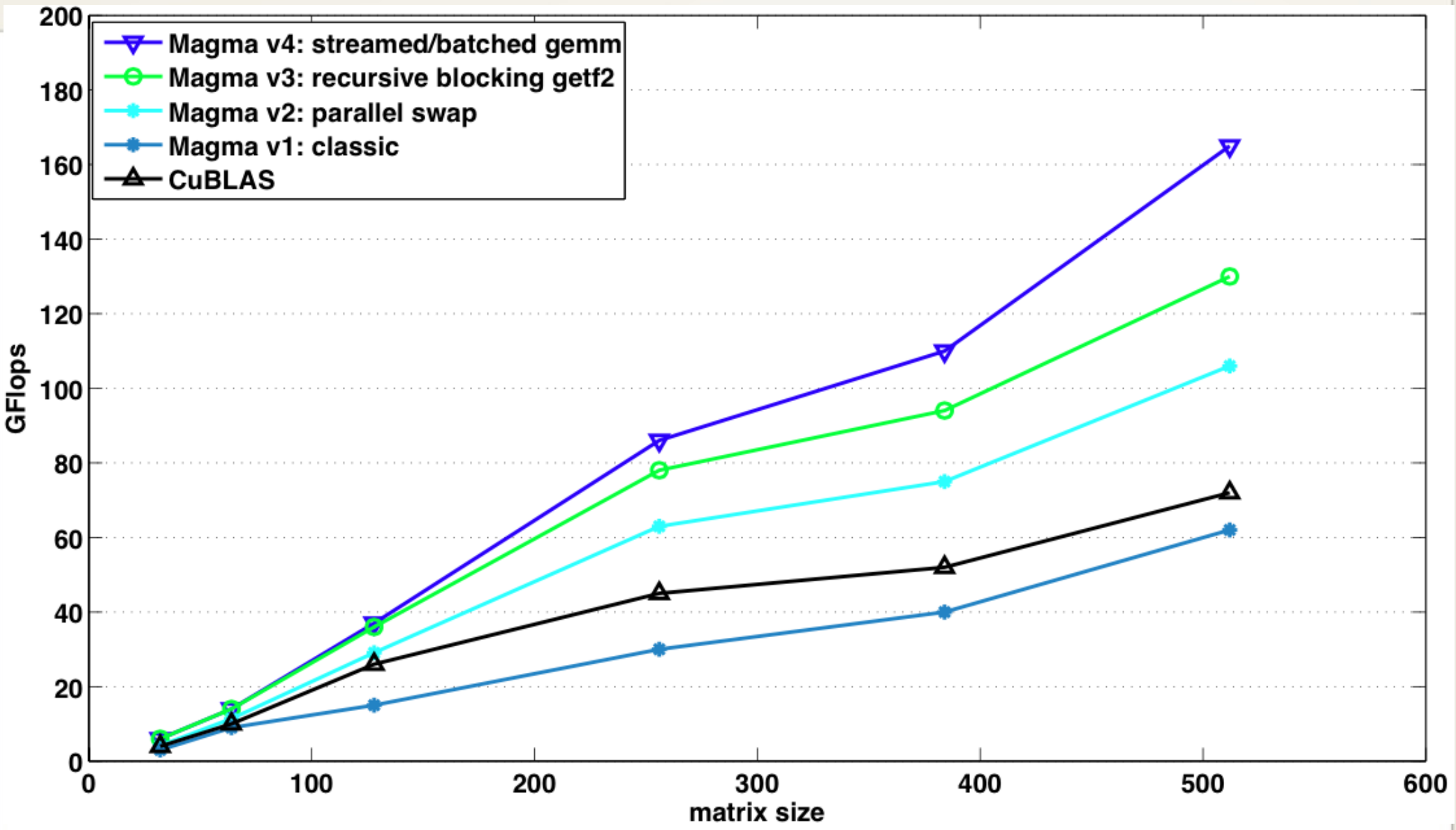


Execution Trace of DGEMM Variants

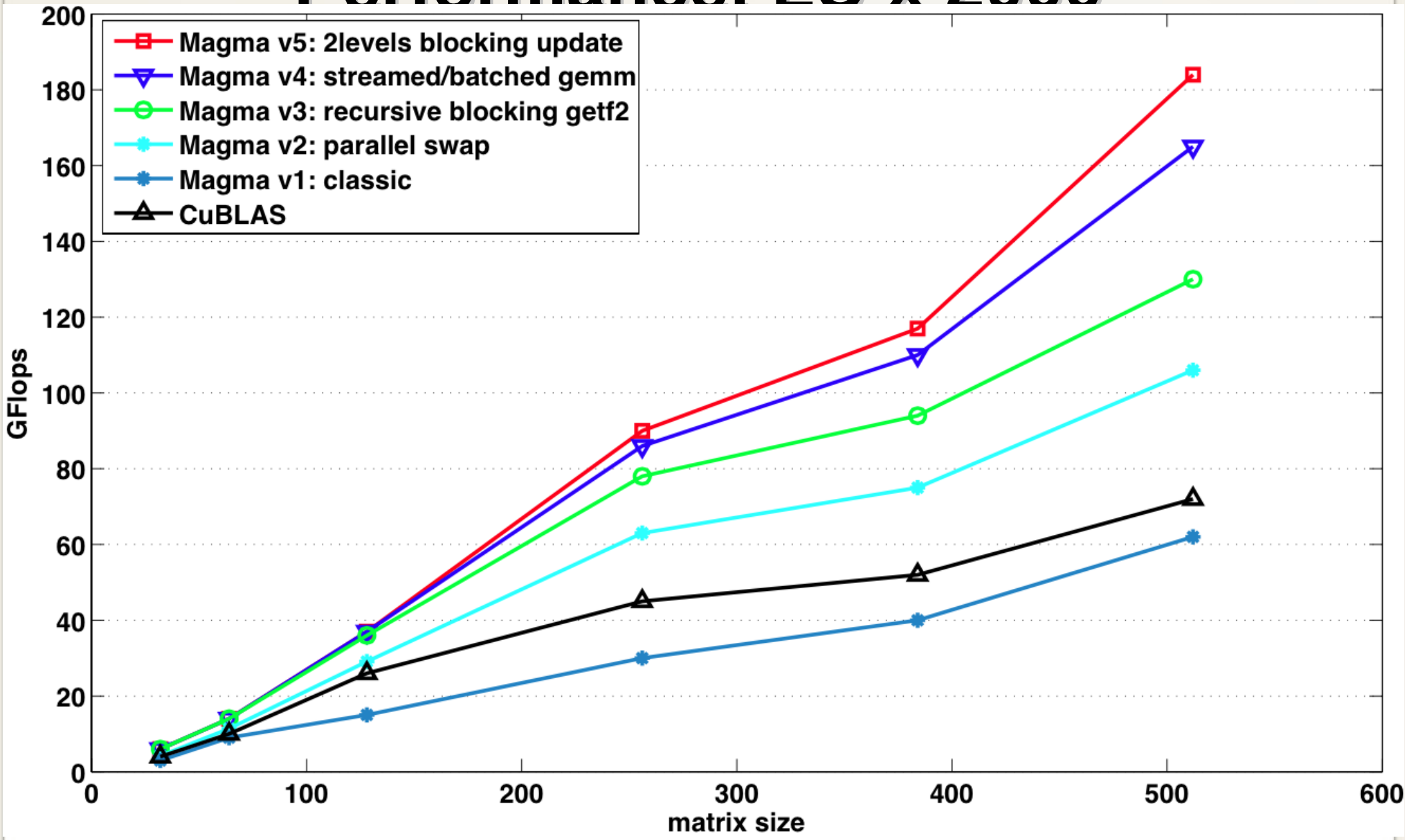




Performance: LU x 2000



Performance: LU x 2000



Conclusions and Future Work

- One sided factorizations need reworking to work in batch scenario
  - Our `DGETRF` and `DPOTRF` are much better now than anything available
  - Similar optimizations work across both
  - Improving `DGEQRF` is next on the agenda
- Two sided factorizations are an important next target
  - These includes: symmetric and non-symmetric eigenvalues as well as SVD
  - Factorizations: tri-diagonal, bi-diagonal, Hessenberg
    - Routine names: `DSYTRD`, `DGEBRD`, `DGEHRD`
  - Much more complicated dependence structure and memory access patterns
  - But applications demand it: recommender systems, electronic structure calculation, …
- GPU-only implementations for self-hosted and weak CPU chips