On the design, autotuning, and optimization of GPU kernels for kinetic network simulations using fast explicit integration and GPU batched computation

Mike Guidry[1] and Azzam Haidar[1]

In collaboration with

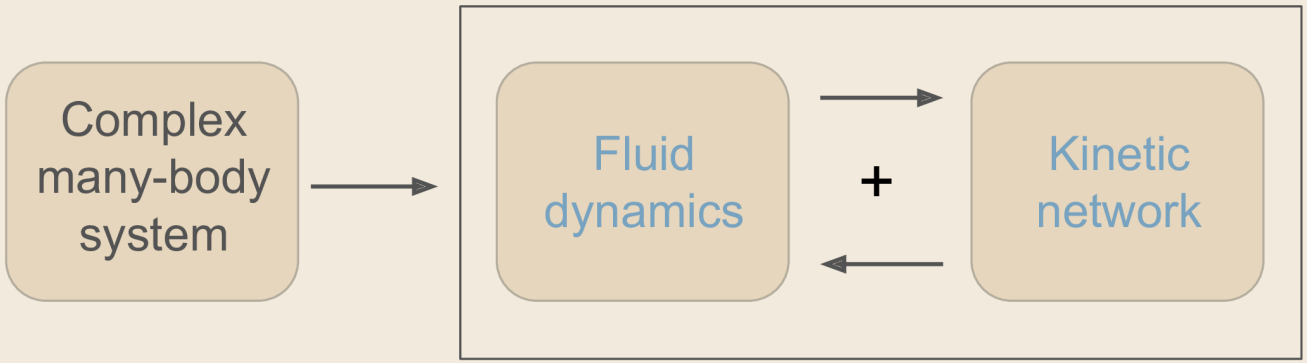
Ben Brock[1], Daniel Shyles[1], Stan Tomov[1],
Jay Billings[2], and Andrew Belt[1]

(1) *University of Tennessee*
(2) *Oak Ridge National Laboratory*

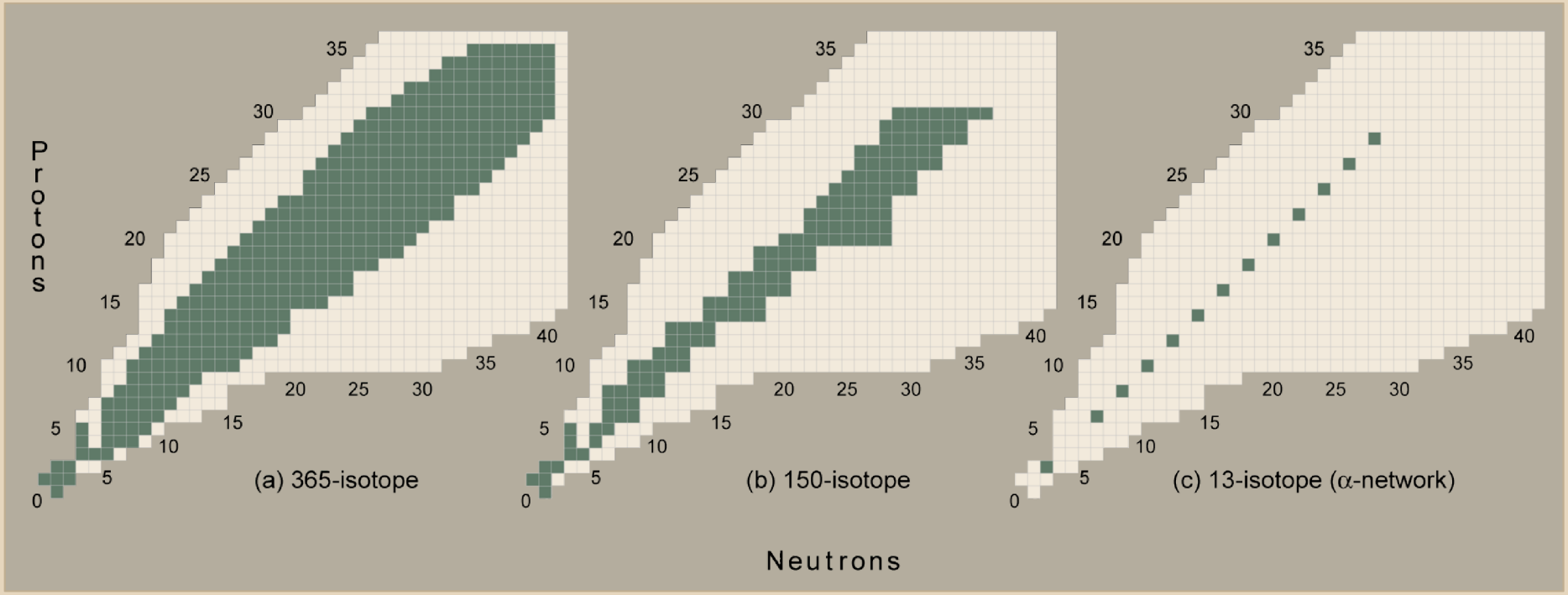
guidry@utk.edu
haidar@icl.utk.edu

Fluid Dynamics Plus Kinetics Approximation

Many physical systems can be modeled by a fluid dynamics plus kinetics approximation.



Realistic Networks for the Type Ia Problem



Coupling Realistic Thermonuclear Networks to Hydrodynamics

To incorporate realistic networks in astrophysical simulations we must improve (substantially) the speed and efficiency for computing kinetic networks coupled to fluid dynamics.

There are two general approaches that we might take:

- Improve the algorithms used to solve the kinetic networks.
- Improve the hardware on which the algorithms are executed.

This presentation is about using both to affect a dramatic improvement in the speed and efficiency for solving this problem.

Integrating Stiff Equations Numerically

Explicit numerical integration:

To advance the solution from time t_n to t_{n+1}, only information already available at t_n is required.

Implicit numerical integration:

To advance the solution from time t_n to t_{n+1}, information at the new point t_{n+1} is required, implying an *iterative solution*.

Thus, for numerical integration

- Explicit methods are *inherently simple, but potentially unstable.*
- Implicit methods are *inherently complicated, but stable.*

Methods to Integrate Stiff Equations

- There are two general approaches that we might use to deal with stiffness.
  - ***The traditional way:*** Integrate equations implicitly, which is stable but requires an iterative solution with matrix inversions at each step (expensive for large networks).
  - ***A new way:*** Replace equations with some that are more stable and integrate them explicitly.
- If we could stabilize explicit integration we could do each timestep more quickly in large networks.

Fundamental Sources of Stiffness

Negative populations

Macroscopic equilibration

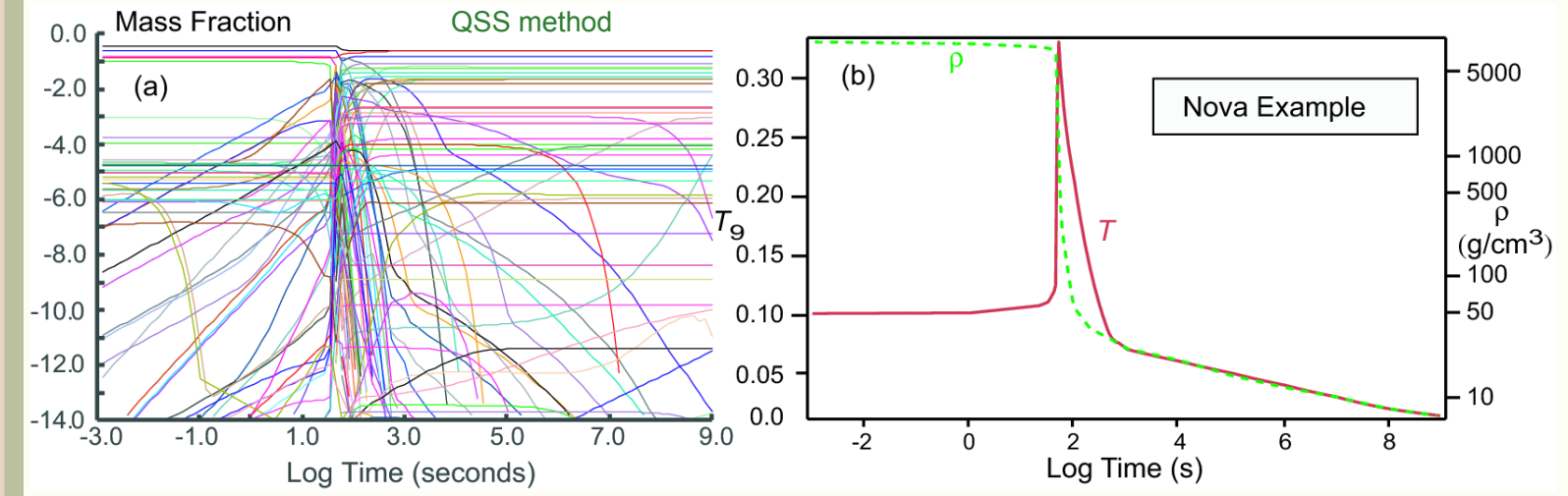
$$\frac{dy_i}{dt} = F_i^+ - F_i^-$$
$$= (f_1^+ + f_2^+ + \ldots)_i - (f_1^- + f_2^- + \ldots)_i$$
$$= (f_1^+ - f_1^-)_i + (f_2^+ - f_2^-)_i + \ldots = \sum_j (f_j^+ - f_j^-)_i$$

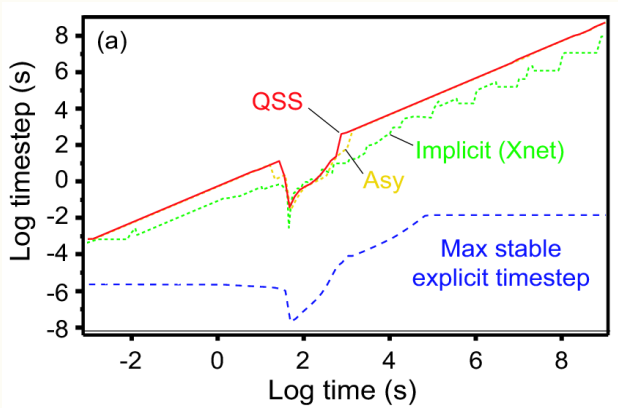
Microscopic equilibration

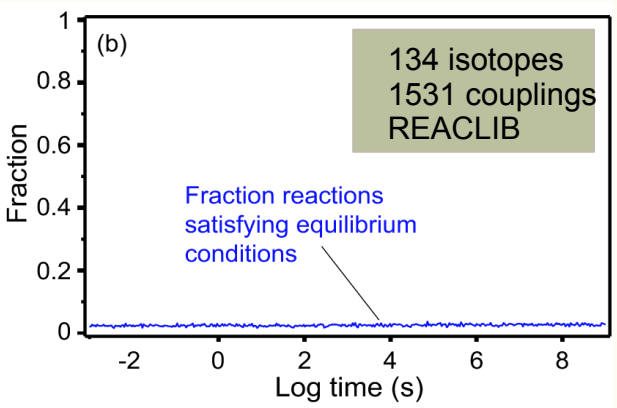
The key to stabilizing explicit integration is to understand the three basic sources of stiffness for a typical reaction network:

- *Negative populations,*
- *Macroscopic equilibration*
- *Microscopic equilibration.*

Example: Explicit Integration for a Nova Simulation

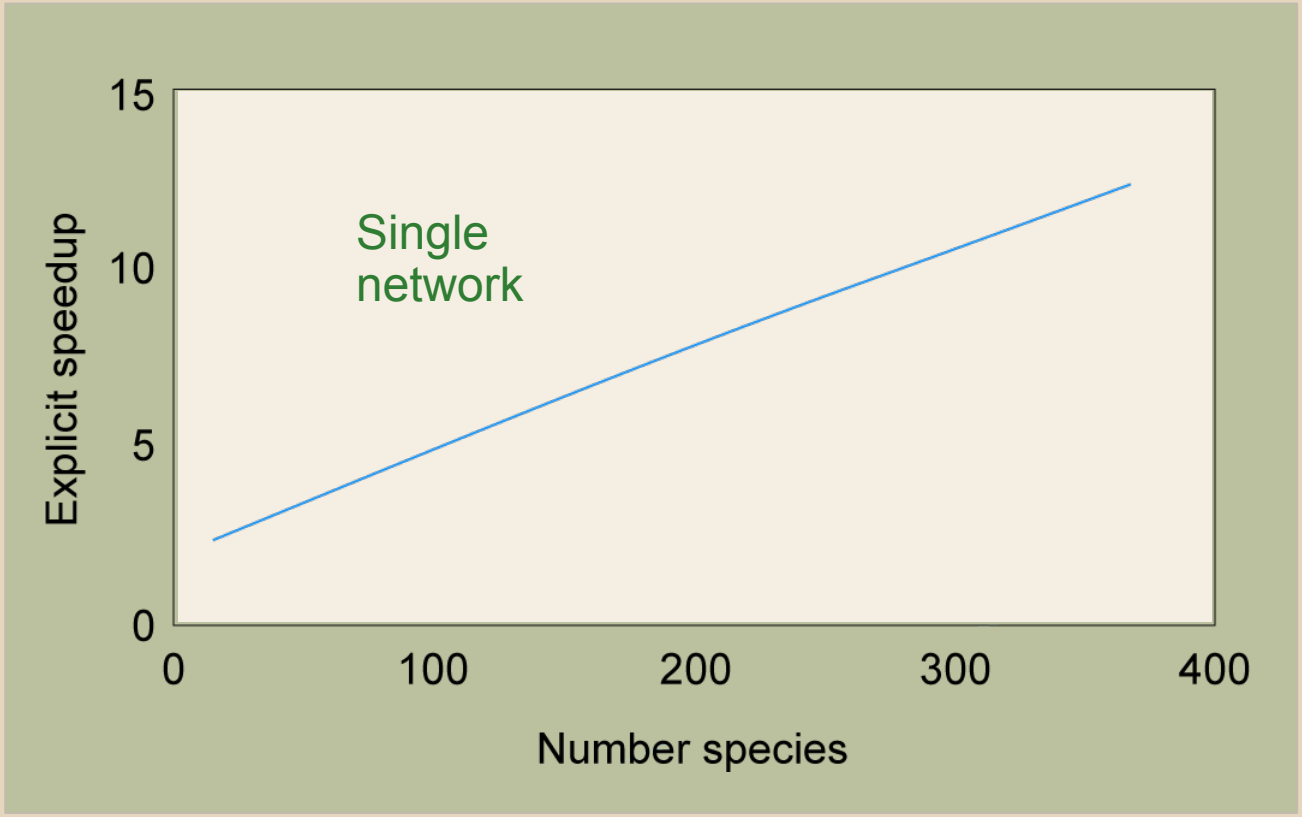






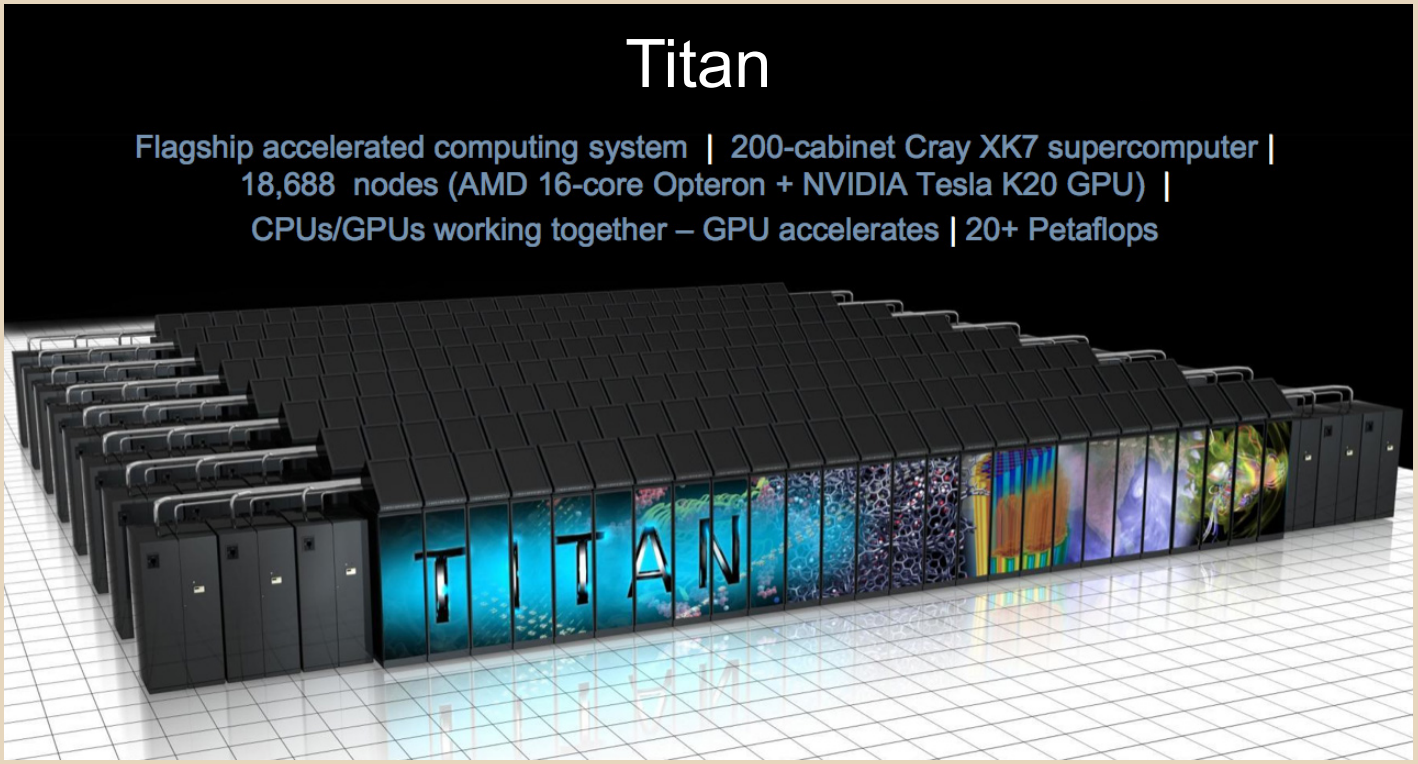
Method	Steps	Speed
Implicit	1332	1
Asy	935	10
QSS	777	12

Summary of Results: Explicit vs Implicit Speedup for a Single Network



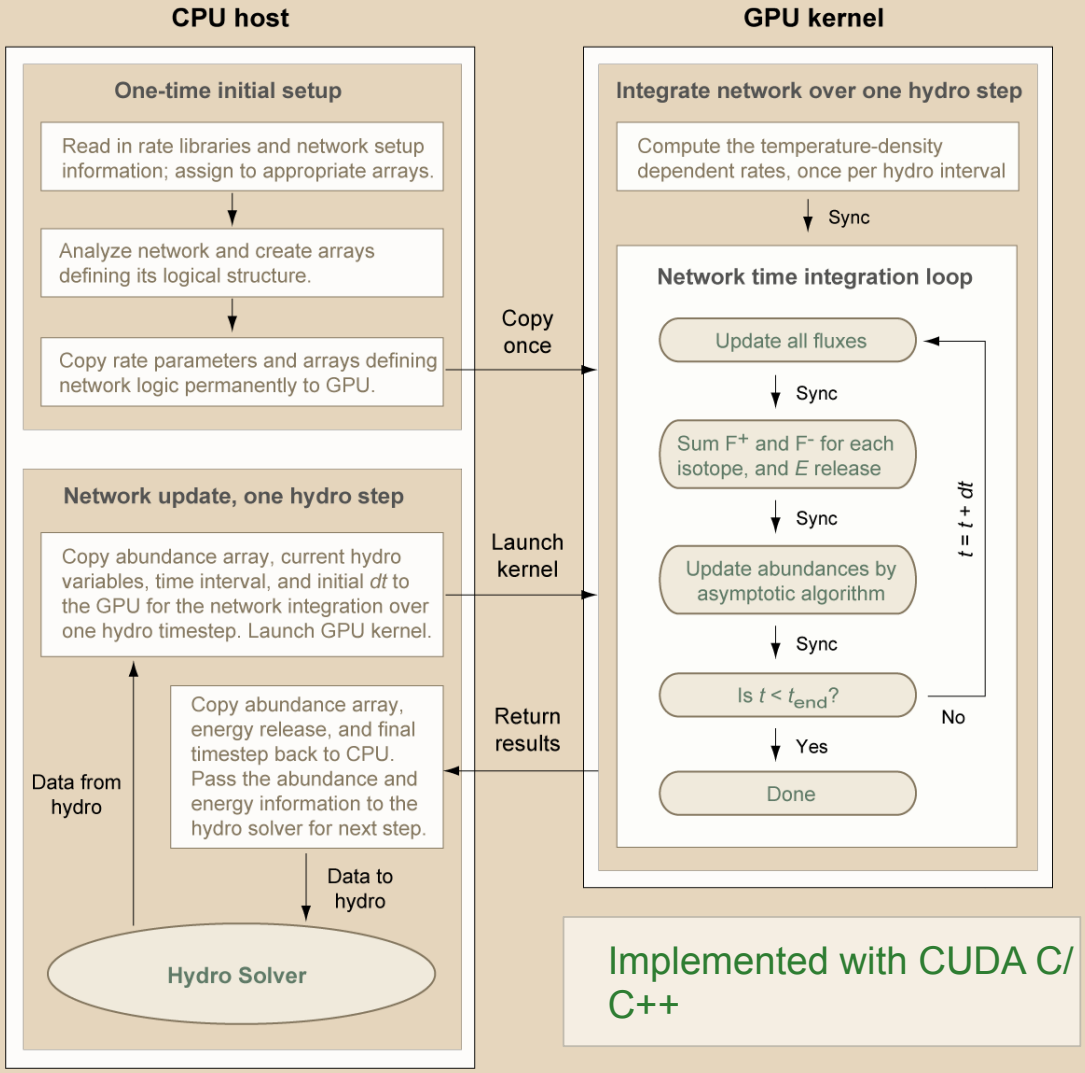
Thus our new algorithms can give a speed increase of about an order of magnitude for networks with several hundred species. Now let us consider the *role of modern hardware in this problem.*

Computing Power for Scientific Applications

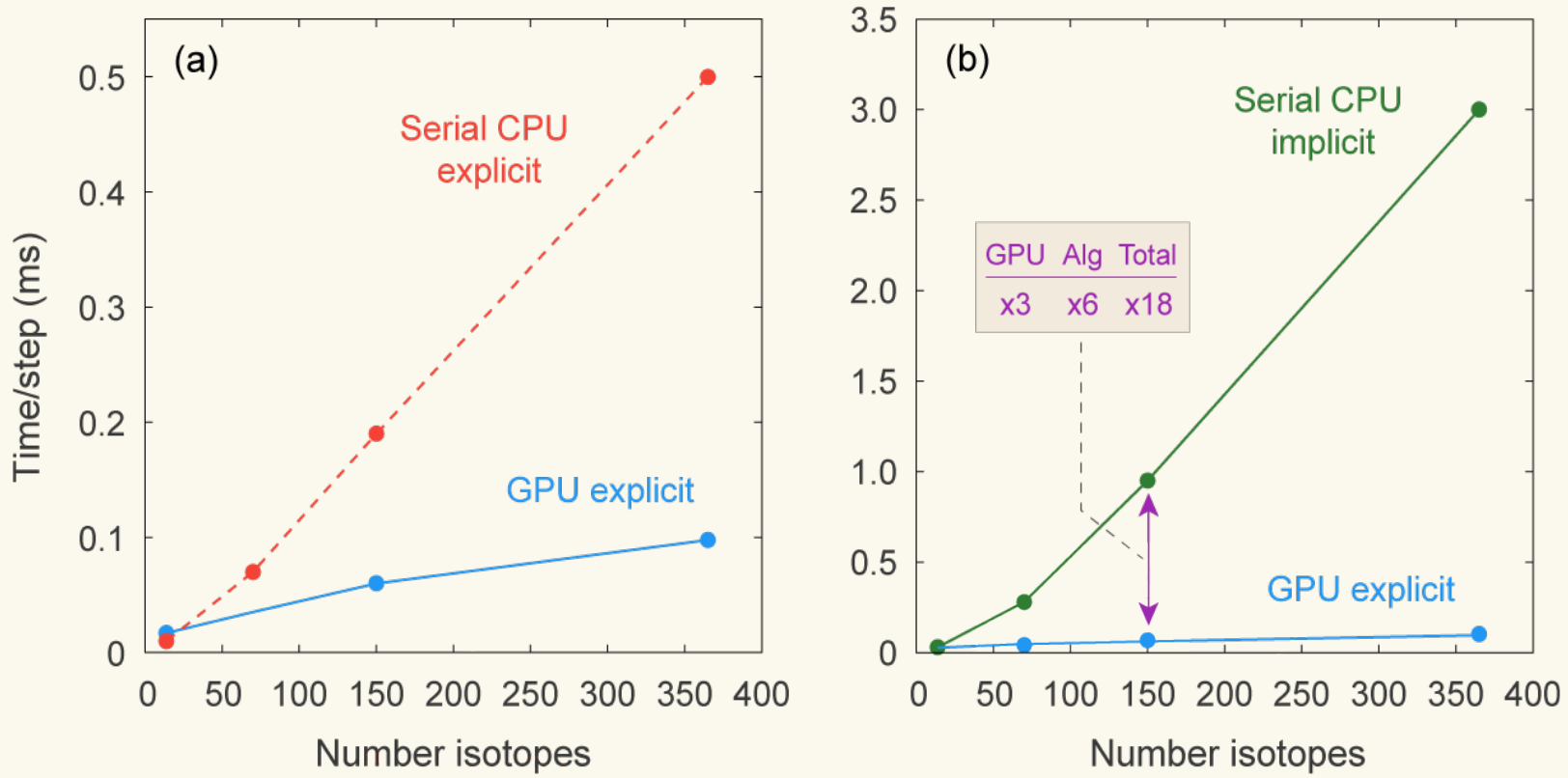


Total of 299,008 CPU cores and 18,868 GPUs. Capable of 27×10^{15} floating point operations per second (27 petaflops).

GPU Acceleration for the Network

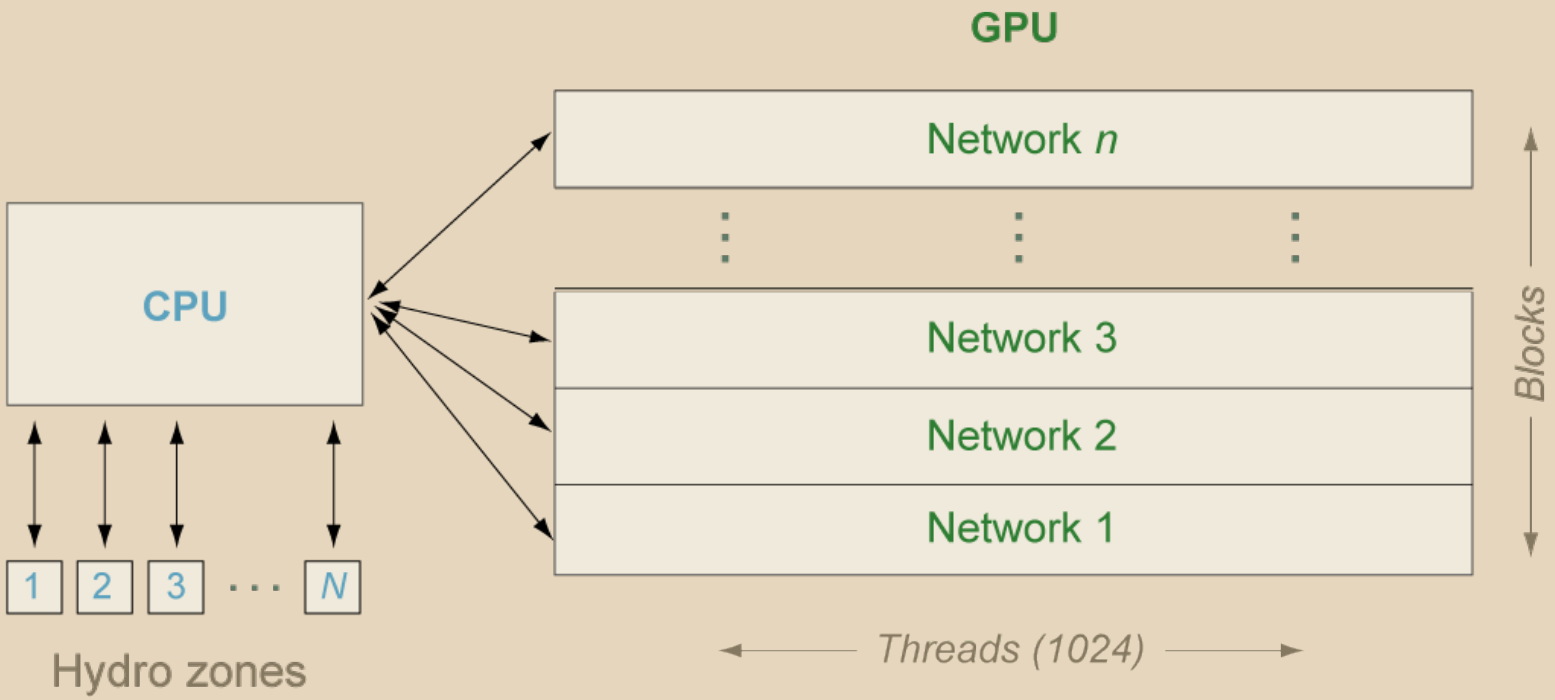


Scaling for a Single Network



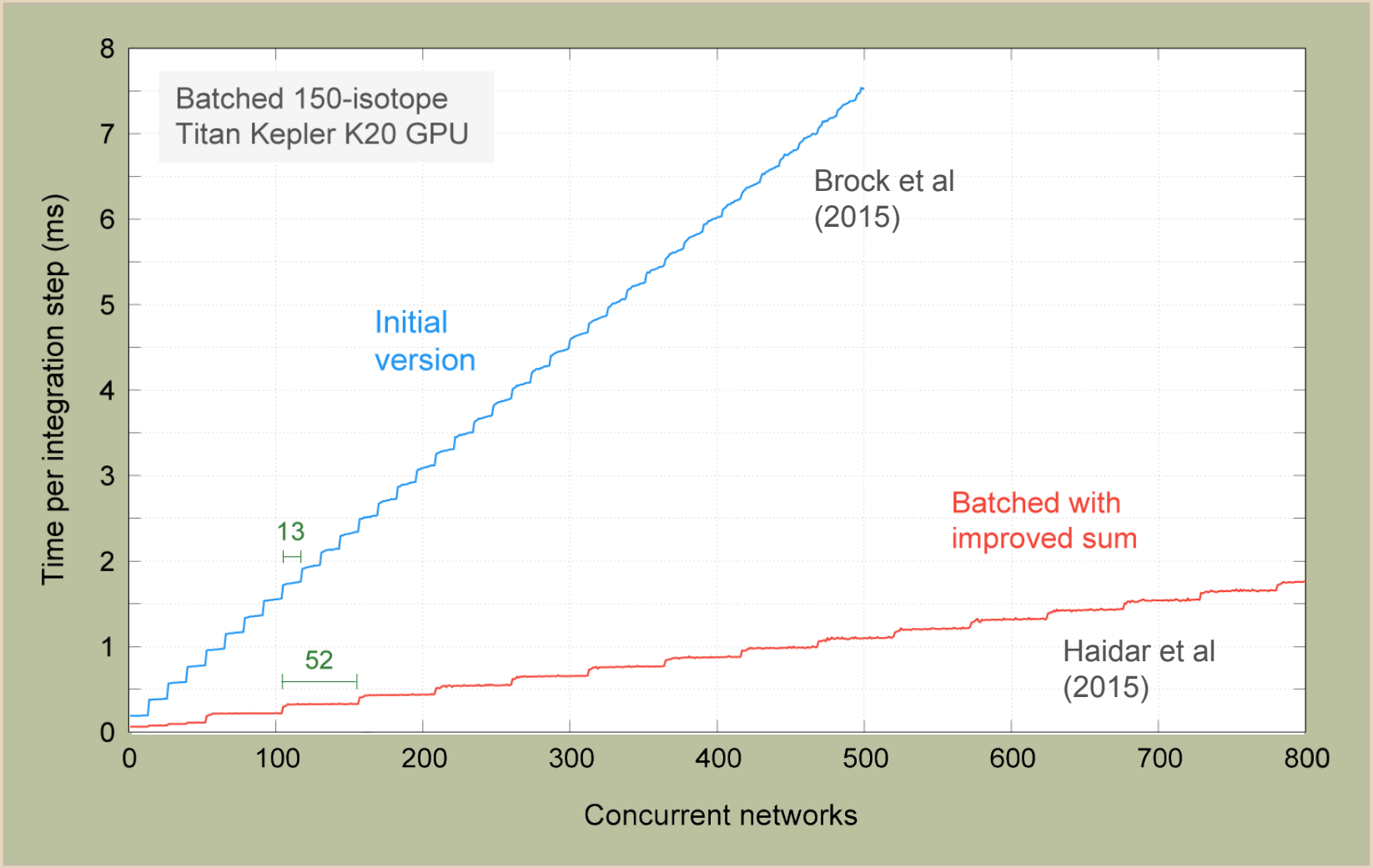
This is impressive speedup but *a single network utilizes only a small fraction of available GPU threads*. Greater efficiency requires that we *give the GPU more work.*

Stacking Multiple Networks on a GPU

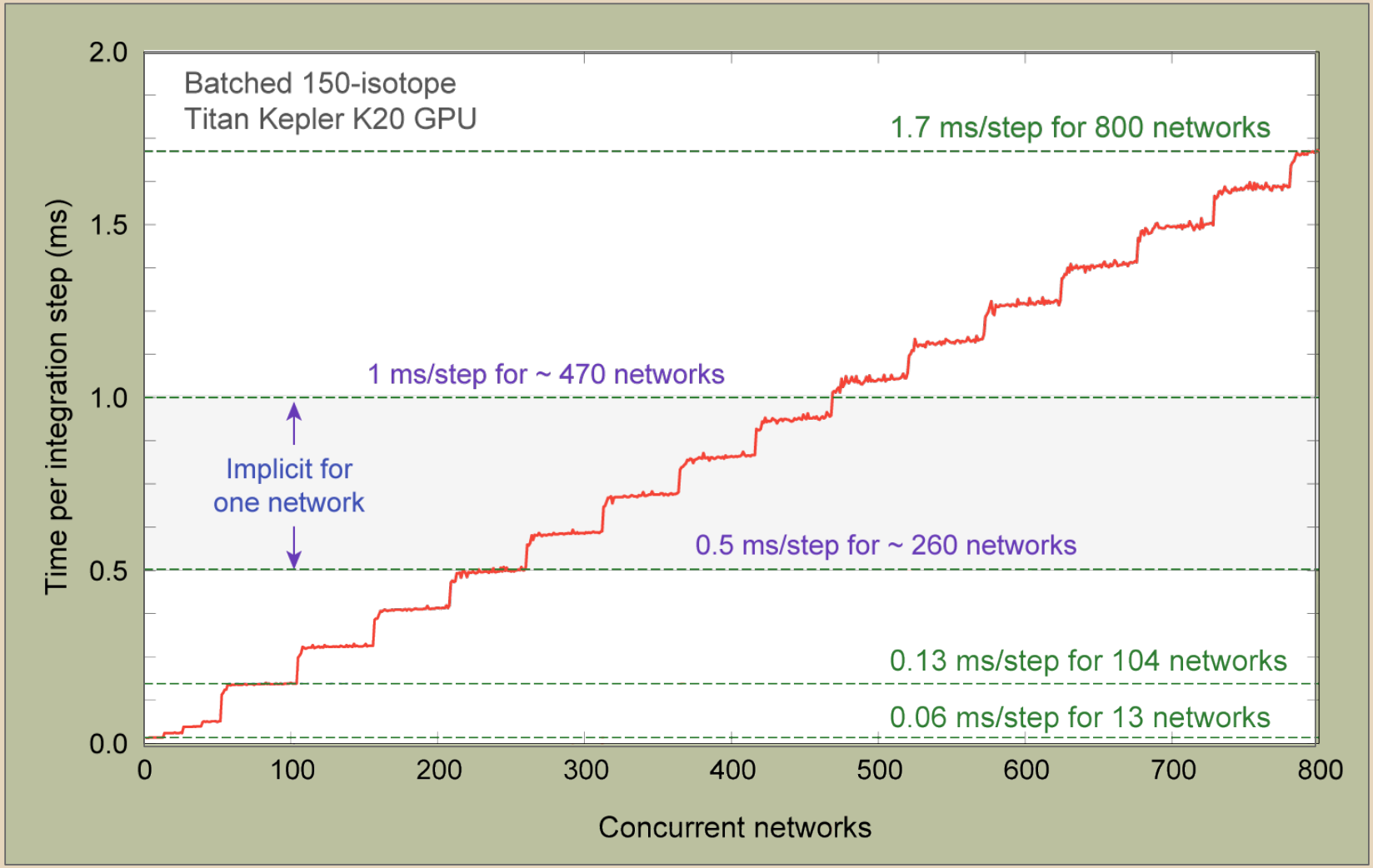


Thus, not only might it be possible to run one network of realistic size faster than is now feasible, it may be possible to run many such networks faster than it is now possible to run one such network.

Timing: Concurrent Network Launches



Timing: Concurrent Network Launches



References

Algebraic Stabilization of Explicit Numerical Integration for Extremely Stiff Reaction Networks, Mike Guidry, J. Comp. Phys. 231, 5266-5288 (2012). [ArXiv:1112.4778]

Explicit Integration of Extremely-Stiff Reaction Networks: Quasi-Steady-State Methods, M. W. Guidry and J. A. Harris, Comput. Sci. Disc. 6, 015002 (2013) [ArXiv:1112.4750]

Explicit Integration of Extremely-Stiff Reaction Networks: Partial Equilibrium Methods, M. W. Guidry, J. J. Billings, and W. R. Hix, Comput. Sci. Disc. 6, 015003 (2013) [arXiv: 1112.4738]

Explicit Integration of Extremely-Stiff Reaction Networks: Asymptotic Methods, M. W. Guidry, R. Budiardja, E. Feger, J. J. Billings, W. R. Hix, O. E. B. Messer, K. J. Roche, E. McMahon, and M. He, Comput. Sci. Disc. 6, 015001 (2013) [ArXiv: 1112.4716]

Explicit Integration with GPU Acceleration for Large Kinetic Networks, Ben Brock, Andrew Belt, Jay Billings, and Mike Guidry, submitted to J. Comp. Phys. [arXiv:1409.5826]

arXiv = http://arxiv.org/

Collaborators

- Ben Brock, *University of Tennessee*
- Daniel Shyles, *University of Tennessee*
- Azzam Haidar, *University of Tennessee*
- Stan Tomov, *University of Tennessee*
- Jay Billings, *Oak Ridge National Laboratory*
- Andrew Belt, *University of Tennessee*
- Mike Guidry, *University of Tennessee*

On the design, autotuning, and optimization of GPU kernels for kinetic network simulations using fast explicit integration and GPU batched computation

Mike Guidry[1] & Azzam Haidar[1]
In collaboration with
Ben Brock[1], Daniel Shyles[1], Stan Tomov[1], Jay Billings[2], Andrew Belt[1]

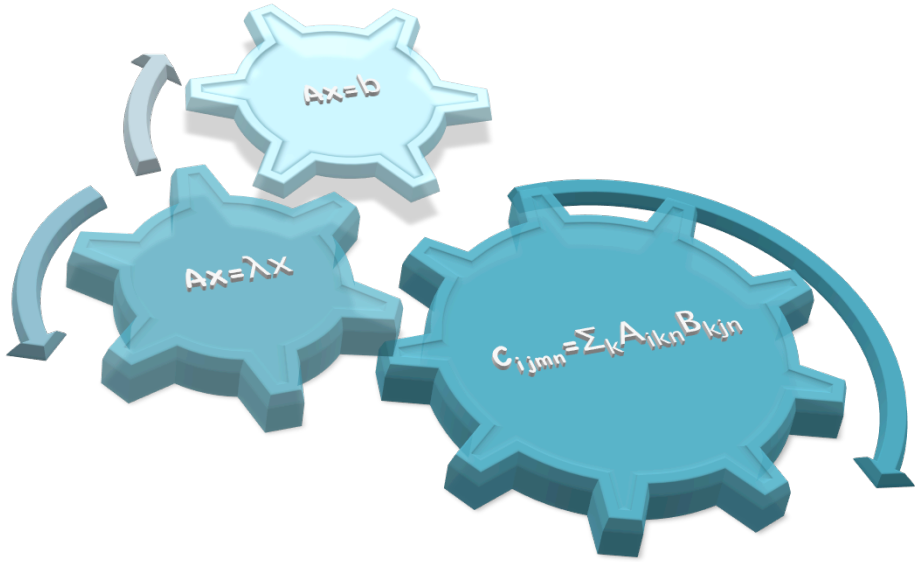
(1) *University of Tennessee*
(2) *Oak Ridge National Laboratory*

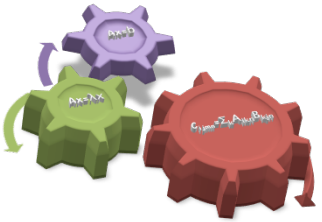
guidry@utk.edu
haidar@icl.utk.edu

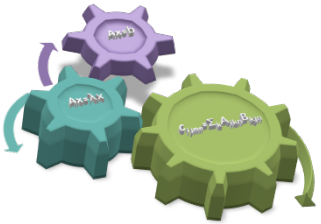
ICL INNOVATIVE COMPUTING LABORATORY

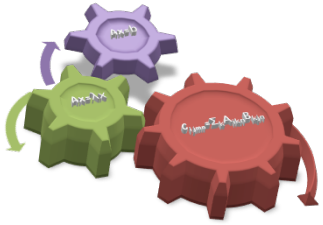
THE UNIVERSITY OF TENNESSEE KNOXVILLE

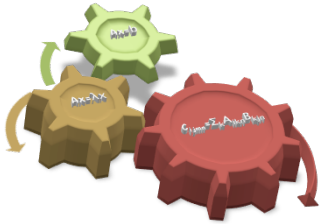
MAGMA Batched Computations

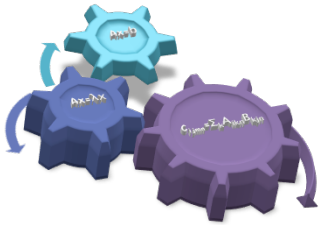












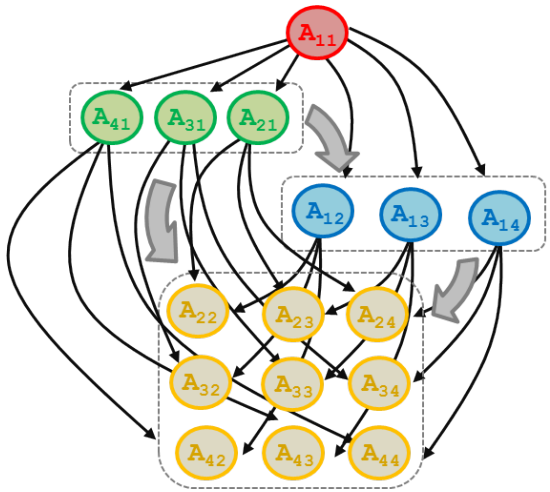
Motivation

- Many dense and sparse direct solvers need HP, energy-efficient LA functionalities on many small independent dense matrices
  - Tiled linear algebra algorithms
  - Multifrontal methods
  - Preconditioners (using DLA) in sparse iterative solvers, many applications, …

Sparse / Dense Matrix System

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

DAG-based factorization

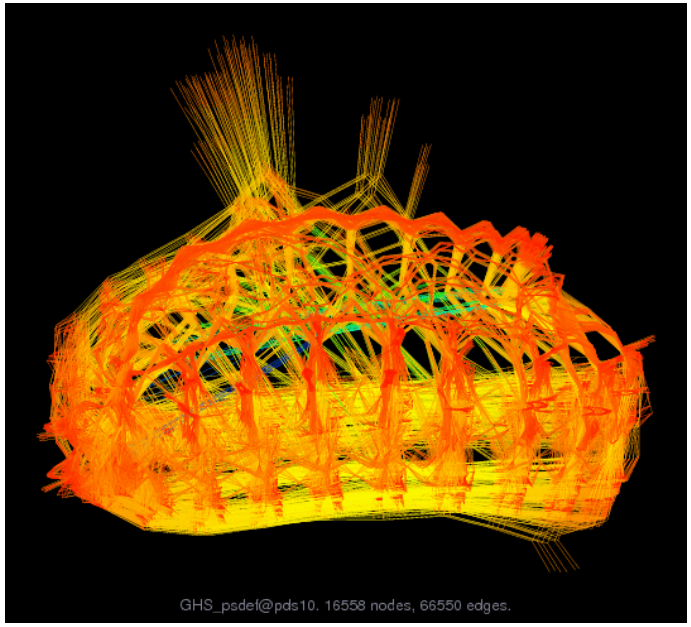


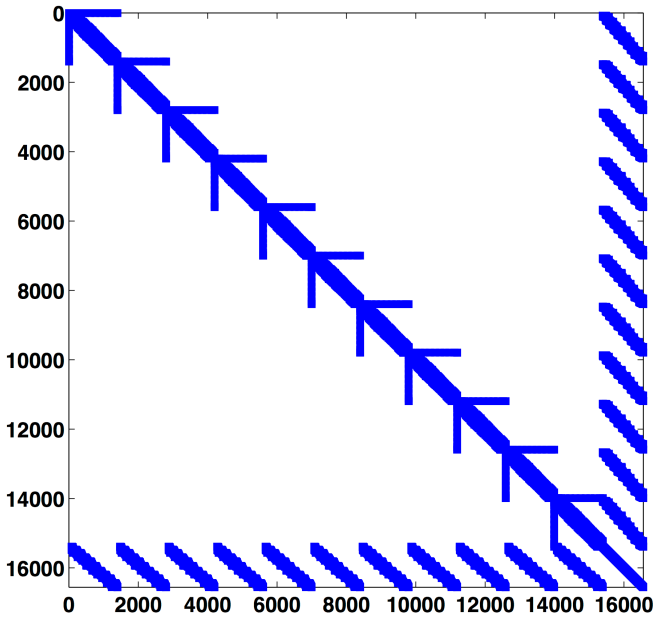
Batched LA

- **LU, QR,** or **Cholesky** on small diagonal matrices
- **TRSM**s, **QR**s, or **LU**s
- **TRSM**s, **TRMM**s
- **Updates (Schur complement)** **GEMM**s, **SYRK**s, **TRMM**s

And many other BLAS/LAPACK, e.g., for application specific solvers, preconditioners, and matrices

MAGMA Batched Computations





Motivation: factorization of thousands of small matrices

- Astrophysics
- Structural mechanics
- High order FEM
- Sparse direct solver
- Tensor contraction
- Machine Learning
- Data Mining
- Hydrodynamics
- Image processing
- Ranking and recommender systems, etc

MAGMA Batched Computations

We present here a feasibility design study, the idea is to target the new high-end technologies.

Observations and current situation:

- There is a lack of linear algebra software for small problems especially for GPU, Xeon Phi, etc
- CPU: this can be done easily using existing software infrastructure
- GPU: are efficient for large data parallel computations, and therefore have often been used in combination with CPUs, where the GPU handle the compute bound operations while the CPU handles the small and difficult tasks to be parallelized
- What programming model is best for small problems?

MAGMA Batched Computations

We present here a feasibility design study, the idea is to target the new high-end technologies.

Our goals:

- to deliver a high- performance numerical library for batched computations tuned for the modern processor architecture and that outperform multicore CPUs in both performance and energy efficiency.
- is to consider both, the higher ratio of execution and the memory model of the new emerging accelerators and coprocessors.
- define modular interfaces that allow code replacement techniques. This will provide the developers of applications, compilers, and runtime systems with the option of expressing computation as a loop, or a single call to a routine from the new batch operation standard.

MAGMA Batched Computations

We present here a feasibility design study, the idea is to target the new high-end technologies.

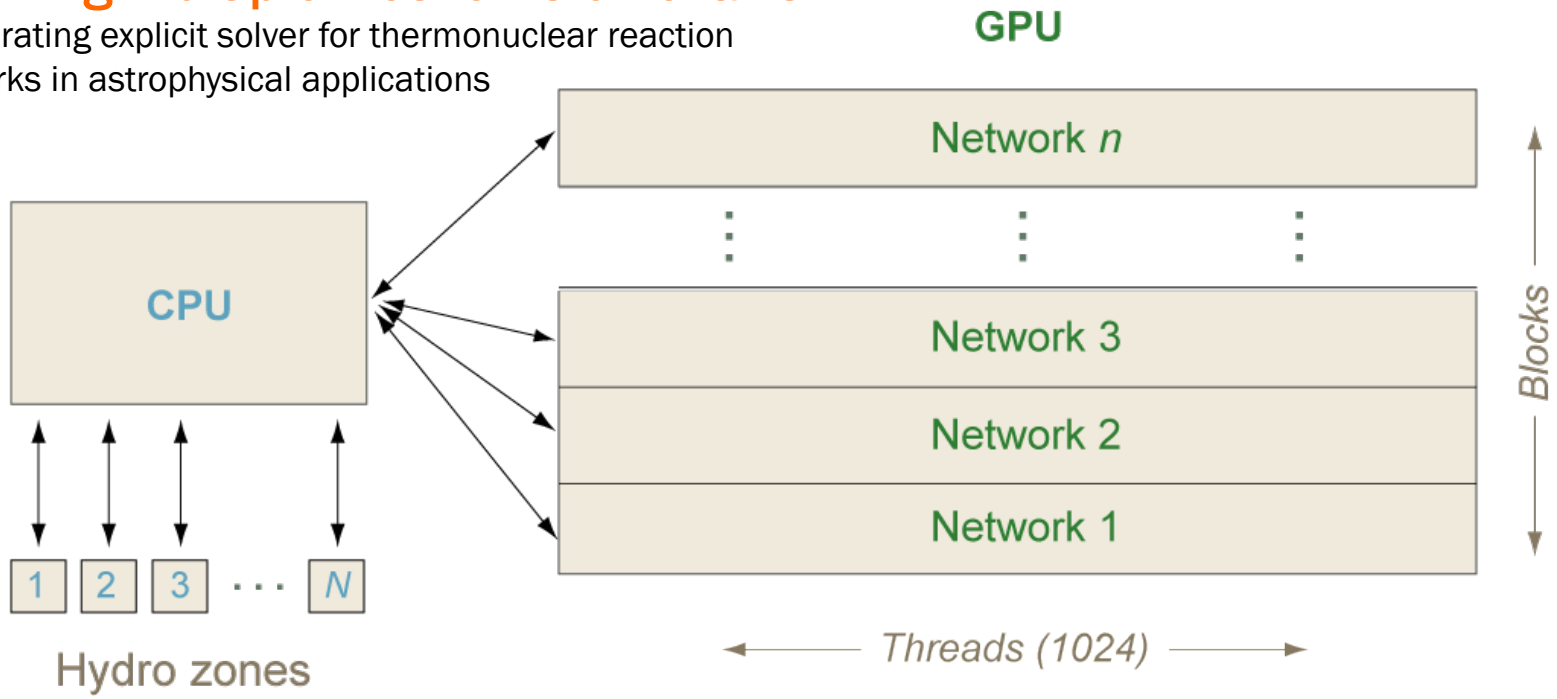
2 examples:

- Accelerating large kinetic networks simulation
- A linear algebra algorithm (LU decomposition)

Kinetic Networks simulation

Stacking Multiple Networks on a GPU

Accelerating explicit solver for thermonuclear reaction networks in astrophysical applications

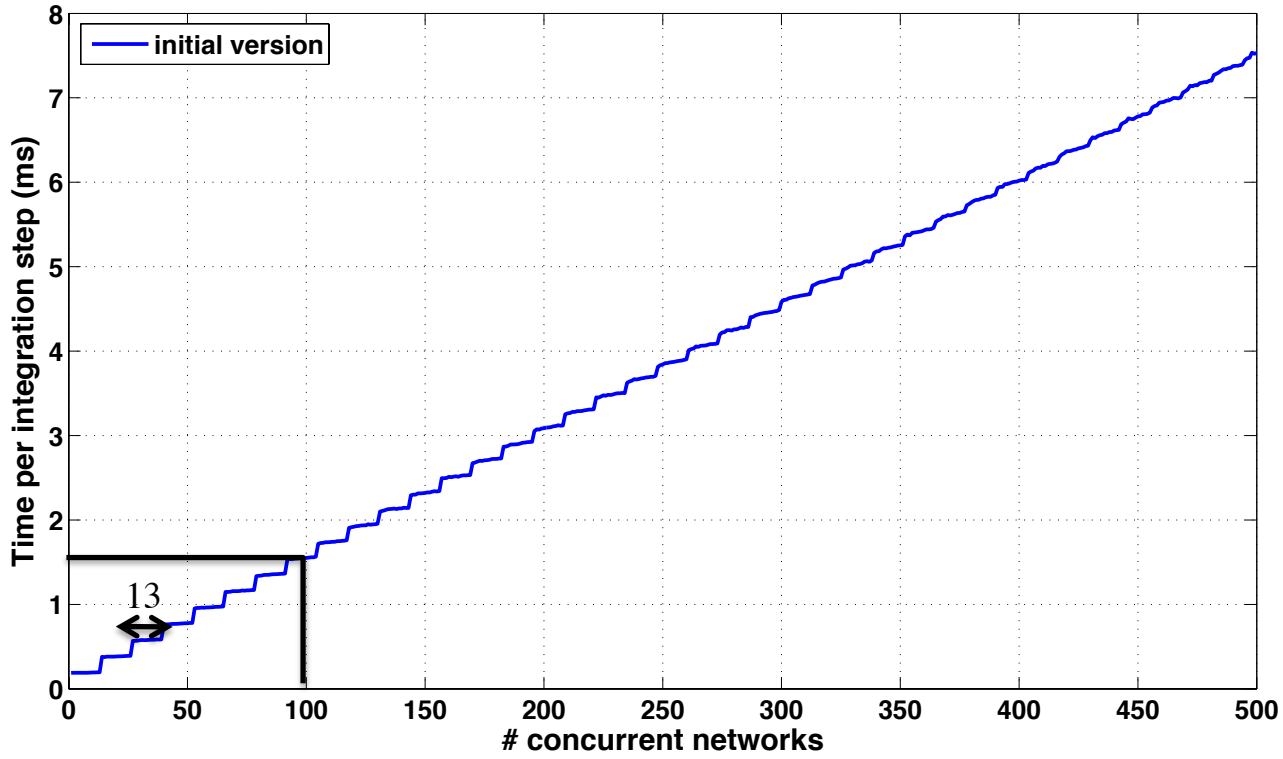


Thus, not only might it be possible to run one network of realistic size faster than is now feasible, it may be possible to run many such networks faster than it is now possible to run one such network.

Kinetic Networks simulation

Stacking Multiple Networks on a GPU

- Develop and optimize the one network kernel
- Use CUDA streams to parallelize on the GPU and run multiple networks simulations



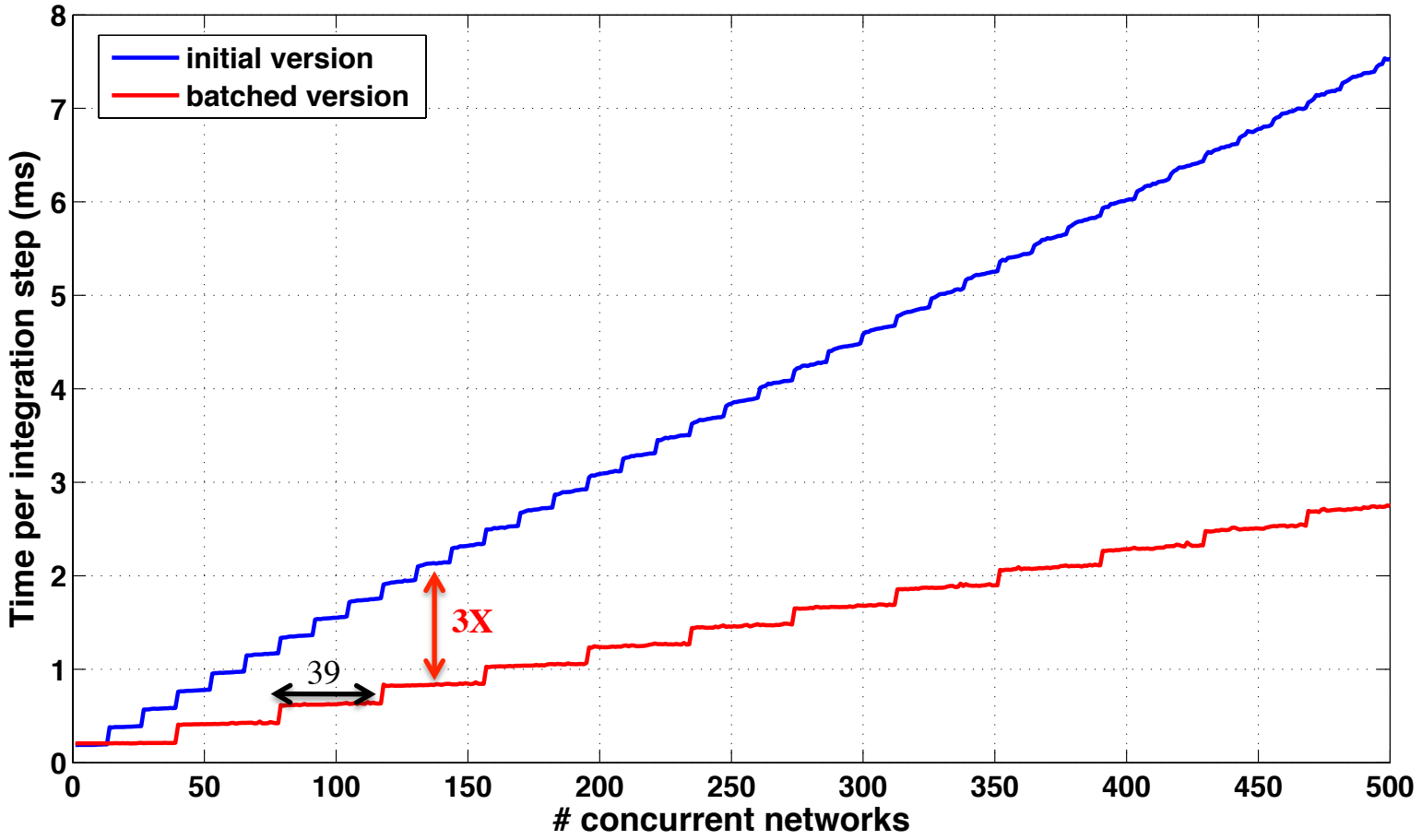
Kinetic Networks simulation

Accelerating explicit solver for thermonuclear reaction networks in astrophysical applications

Introducing batched design:

- Simulates evolution of the nuclear kinetics where for any single time step on a single zone there is need to solve a small computation
- Number of zones can grow with domain size and dimension to tens of thousands
- Zones can be solved independently (batched fashion)
- Redesigning some block of the code, minimizing shared memory requirements and reordering some computation in order to fit our batched design.

Kinetic Networks simulation

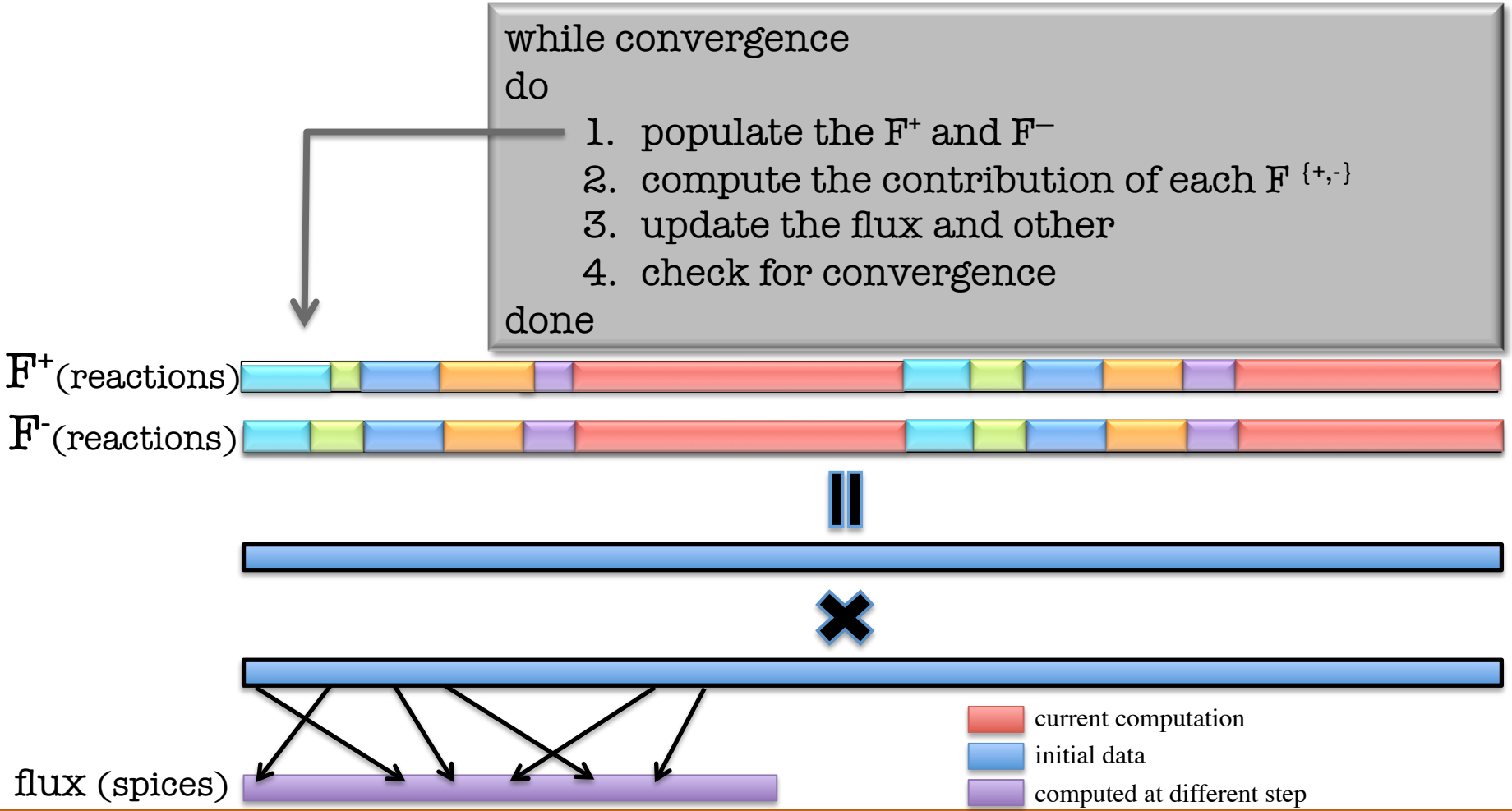


Kinetic Networks simulation

- Observations
  - Batched is faster and able to run about 39 kernels at once instead of 13 kernels for stream
  - 3X speedup observed
  - The calculation of a single zone can be viewed as a loop of 32100 computation
- Bottlenecks
  - The amount of shared memory is considered large for the “batched design”
  - The algorithmic throughput/data structure is not good for the “batched design” number of threads/block, the data layout
- Proposition
  - Analyze all the steps of the algorithm and try to improve it

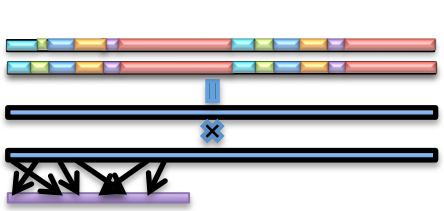
Kinetic Networks simulation

- Observations



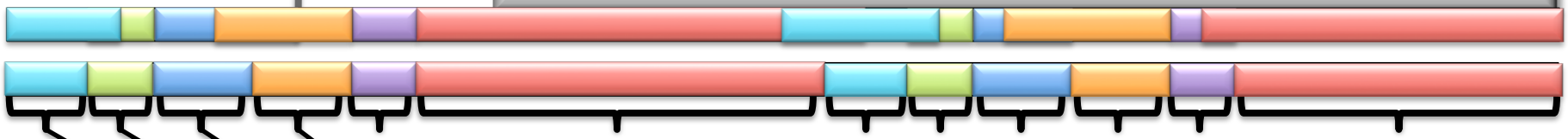
Kinetic Networks simulation

- Observations



while convergence
do
1. populate the F^{+} and F^{-}
2. compute the contribution of each $F^{\{+,-\}}$
3. update the flux and other
4. check for convergence

done



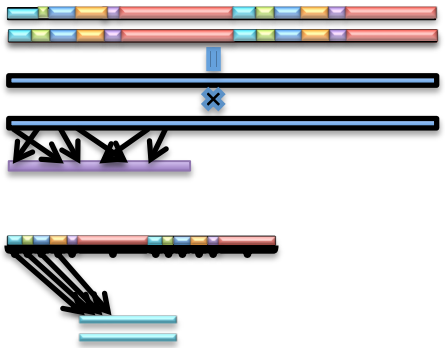
SUM

F^{+} contributions (spices)

F^{-} contributions (spices)

Kinetic Networks simulation

- Observations



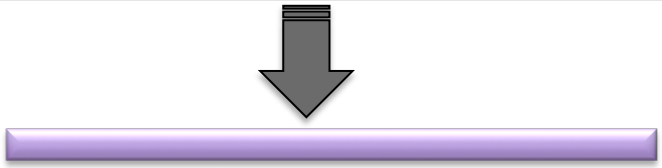
while convergence
do
1. populate the F^{+} and F^{-}
2. compute the contribution of each $F^{\{+,-\}}$
3. update the flux and other
4. check for convergence

done

F^{+} contributions (spices)

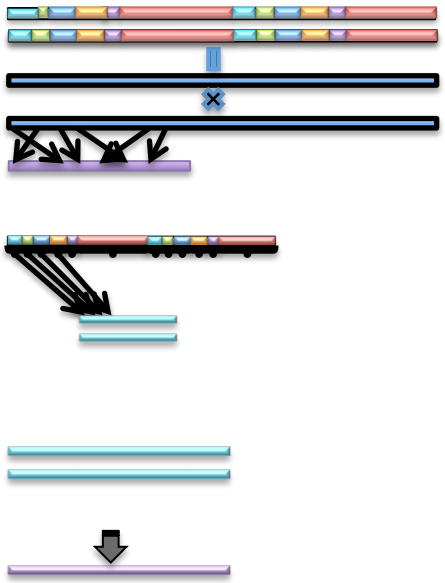
F^{-} contributions (spices)

Explicit method calculation and some magic that read data from the contribution of F^{+} and F^{-} and update the flux vector



Kinetic Networks simulation

- Observations



while convergence
do
1. populate the F^{+} and F^{-} — 10 %
2. compute the contribution of each $F^{\{+,-\}}$ — 80 %
3. update the flux and other
4. check for convergence — 10 %

done

Kinetic Networks simulation

- Observations
  - The main expensive component is the SUM of variable size
  - Data is coalescent (that's true) but is not stored in cacheline
  - Small sum cannot be computed in parallel so sequential so threads do not read coalescent data anymore
  - For the current example there is:
    - 6 large sum of size <512
    - 293 sum of size <32
  - Large sum consume about 70% of the time small sum is about 20%

Kinetic Networks simulation

- Proposition 1
  - Improve the large sum by making another kernel that works using 512 threads
- Observation 1
  - Improvements of about 20% on the large sum has been observed

Kinetic Networks simulation

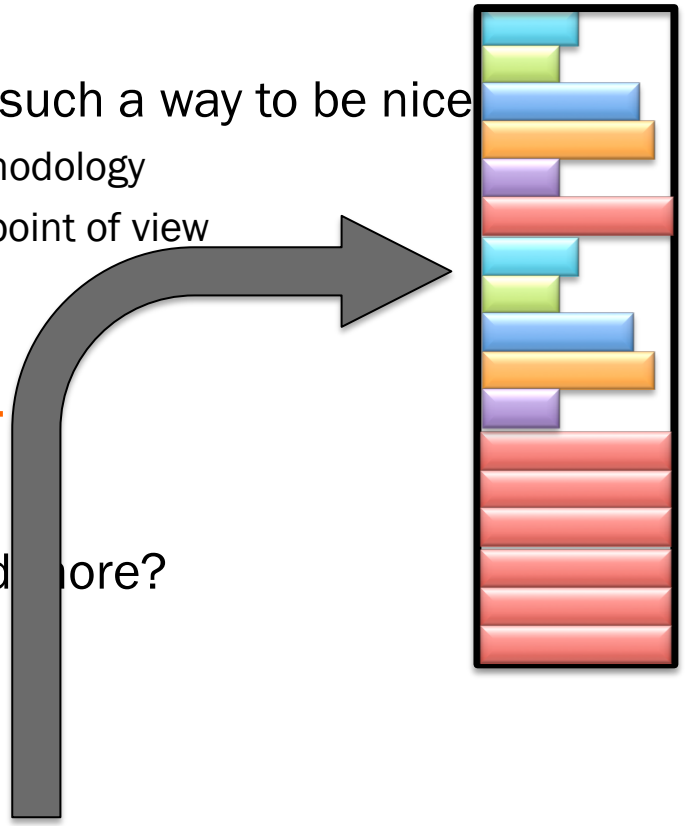
- Proposition 2
  - Try to parallelize the small sum with specific kernels
- Observation 2
  - Do not improve at all, it slow down because of extra cost of reordering and shared memory requirements

Kinetic Networks simulation

- Proposition 3
  - Split the data over two arrays for small and large and use parallel sum since the F^+ and the F^- can proceed in parallel
- Observation 3
  - Very complicated, the sum becomes 3X faster but the populate and becomes the slowest because of non coalescent data (now) and tracking which data is on the small array or large array
  - ➔ need to change the data structure

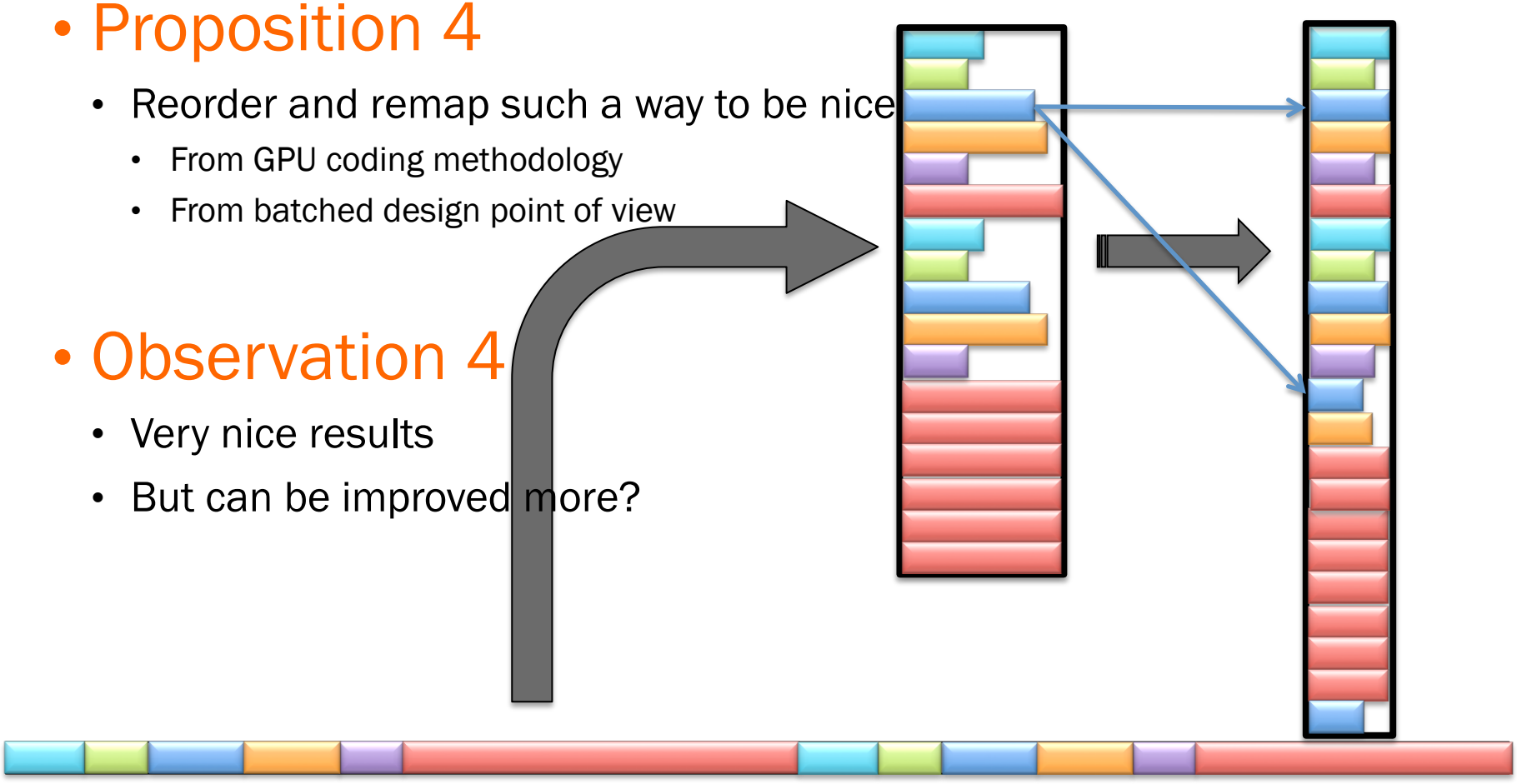
Kinetic Networks simulation

- Proposition 4
  - Reorder and remap such a way to be nice
    - From GPU coding methodology
    - From batched design point of view
- Observation 4
  - Very nice results
  - But can be improved more?

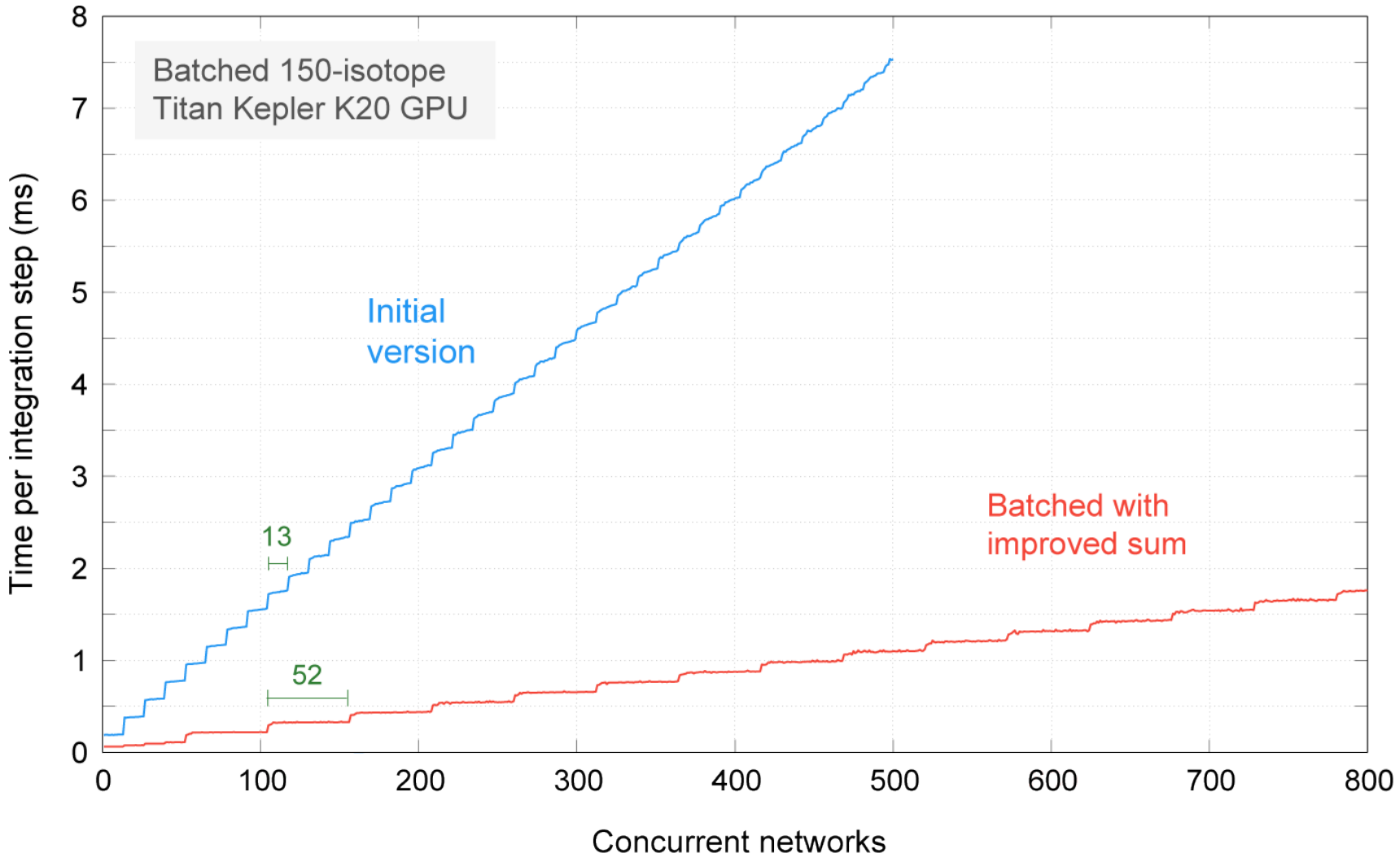


Kinetic Networks simulation

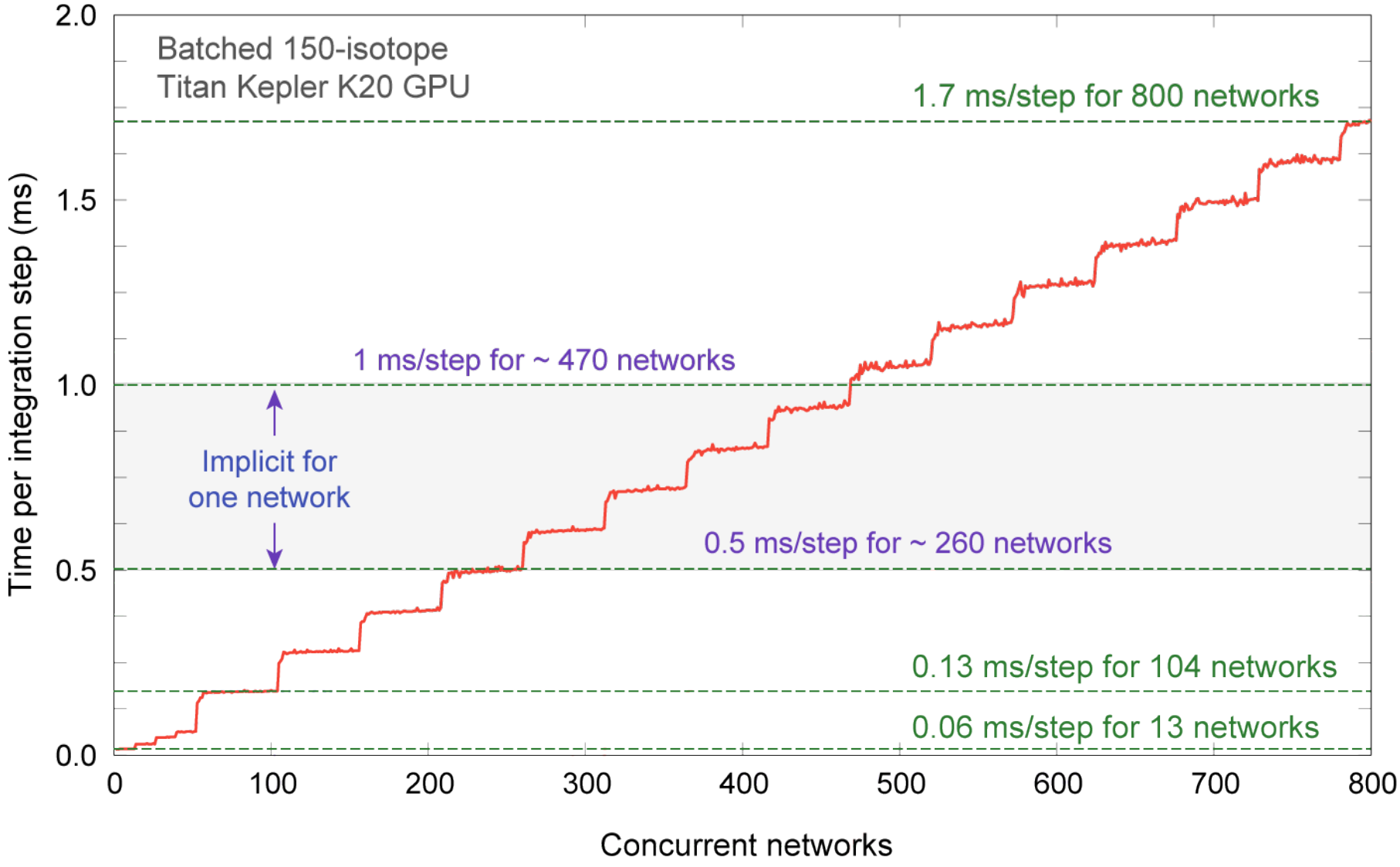
- Proposition 4
  - Reorder and remap such a way to be nice
    - From GPU coding methodology
    - From batched design point of view
- Observation 4
  - Very nice results
  - But can be improved more?



Kinetic Networks simulation



Kinetic Networks simulation



MAGMA Batched Computations

We present here a feasibility design study, the idea is to target the new high-end technologies.

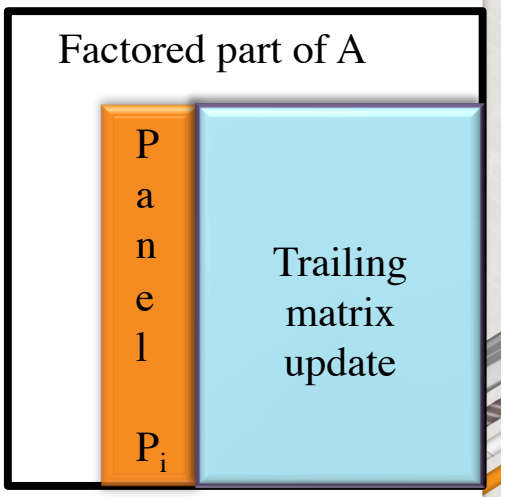
2 examples:

- Accelerating large kinetic networks simulation
- A linear algebra algorithm (LU decomposition)

MAGMA Batched Computations

Algorithmic basics:

- Linear solver *Ax=b* follow the Lapack style algorithmic design blocking algorithm
- Two distinctive phases
  - panel factorization: latency-bound workload
  - trailing matrix update: compute-bound operation



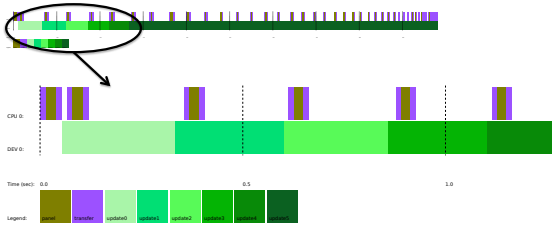
Hardware characteristics and limit:

- GPU memory is limited (48KB of shared per SMX, limited number of register)
- Prefer implementation that extensively uses large number of thread/block (a warp is 32 threads)
- Prefer coalescent memory access (32 threads can read in parallel 32 elements)

MAGMA Batched Computations

Classical strategies design

- For standard problems the strategy is to prioritize the data-intensive operations to be executed by the accelerator and keep the memory-bound ones for the CPUs since the hierarchical caches are more appropriate to handle it



Difficulties

- Cannot be used here since matrices are very small and communication becomes expensive

Proposition

- Go on and have a native GPU implementation

MAGMA Batched Computations

Classical strategies design

- For large problems performance is driven by the update operations,

Difficulties

- For batched small matrices it is more complicated and requires both phases to be efficient

Proposition

- Redesign both phases in a tuned efficient way

MAGMA Batched Computations

Classical strategies design

- A recommended way of writing efficient GPU kernels is to use the GPU's shared memory – load it with data and reuse that data in computations as much as possible.

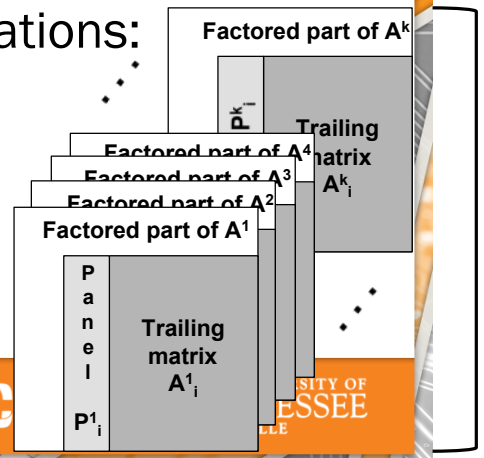
Difficulties

- Our study and experience shows that this procedure provides very good performance for classical GPU kernels but is not that appealing for batched algorithm for different reasons:

MAGMA Batched Computations

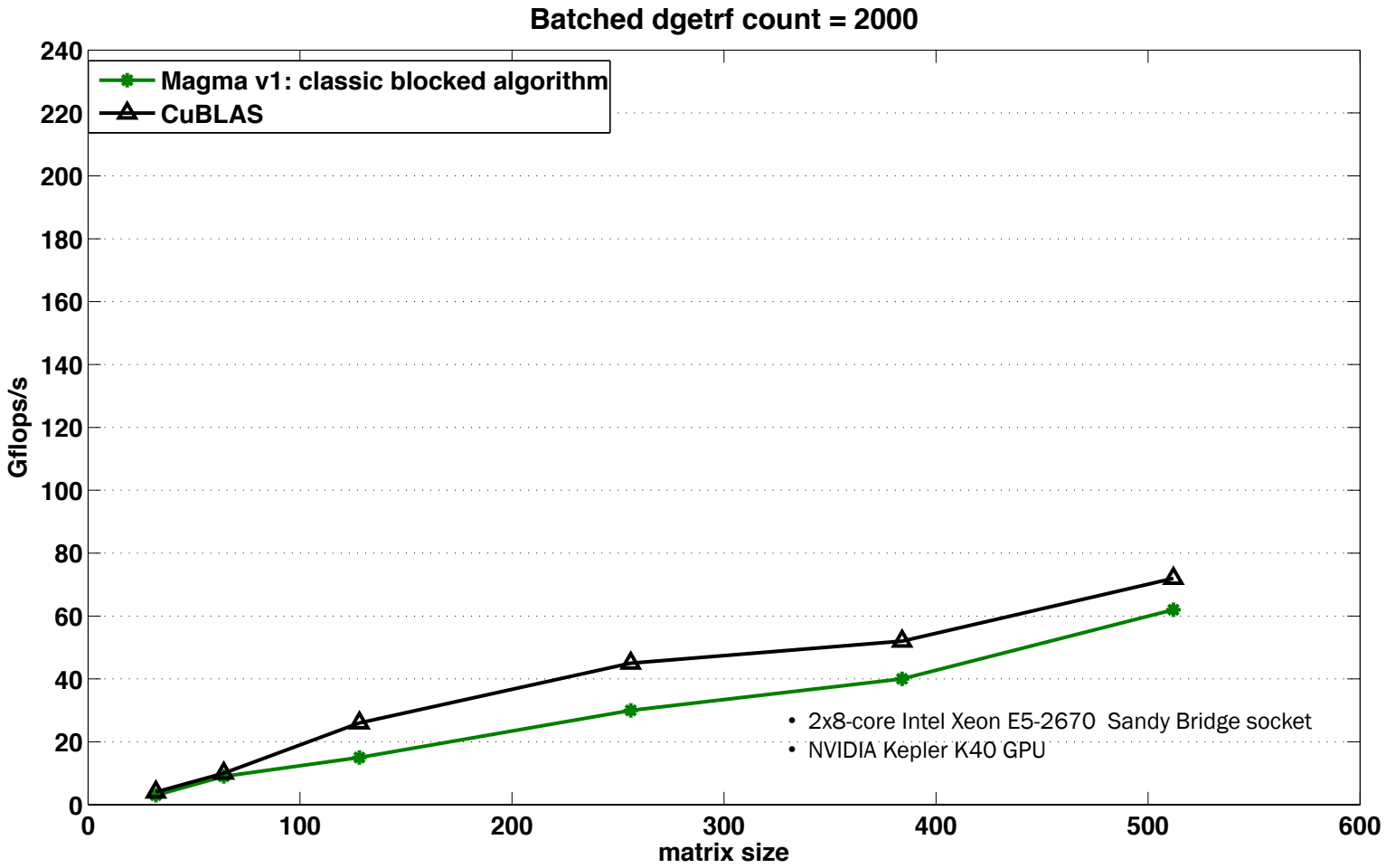
Difficulties

- Completely saturating the shared memory per SMX can decrease the performance of memory bound operations, since only one thread-block will be mapped to that SMX at a time (low occupancy)
- due to a limited parallelism in the panel computation, the number of threads used in the thread block will be limited, resulting in low occupancy, and subsequently poor core utilization
- Shared memory is small (48KB/SMX) to fit the whole panel
- The panel computation involves different type of operations:
  - Vectors column (find the max, scale, norm, reduction)
  - Row interchanges (swap)
  - Small number of vectors (apply)



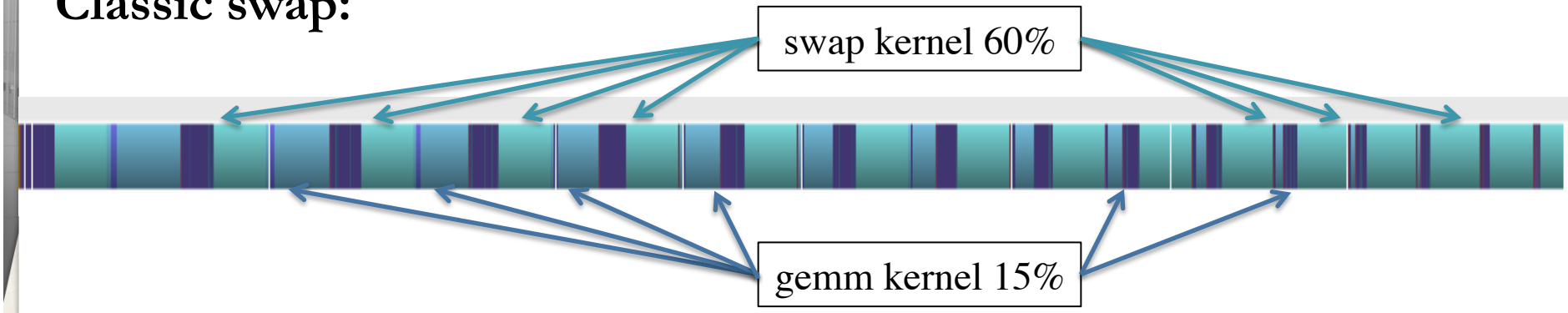
Proposition: custom design per operations type

MAGMA Batched Computations

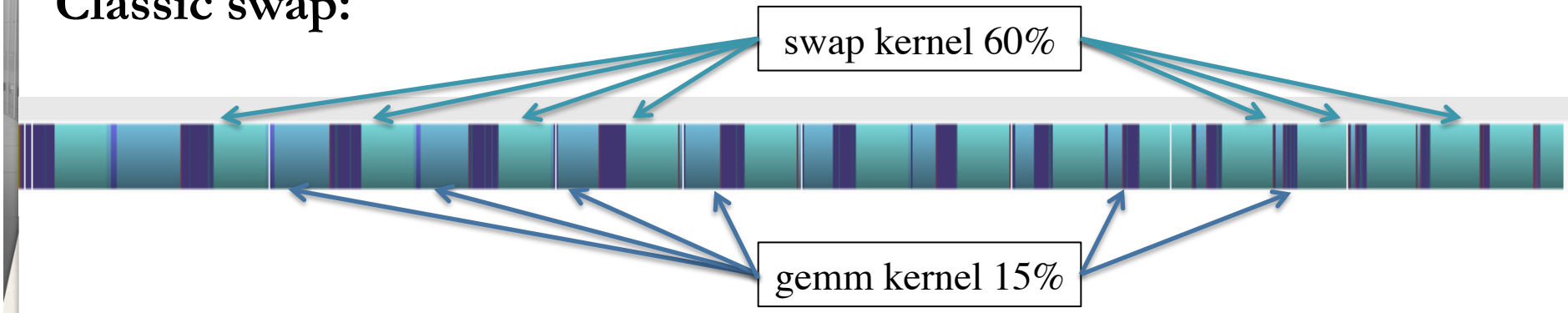


MAGMA Batched Computations

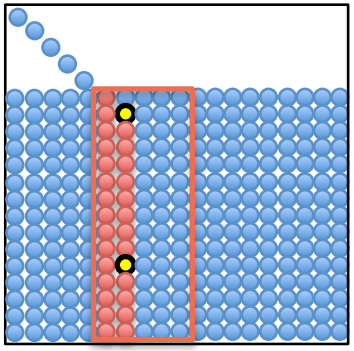
Classic swap:



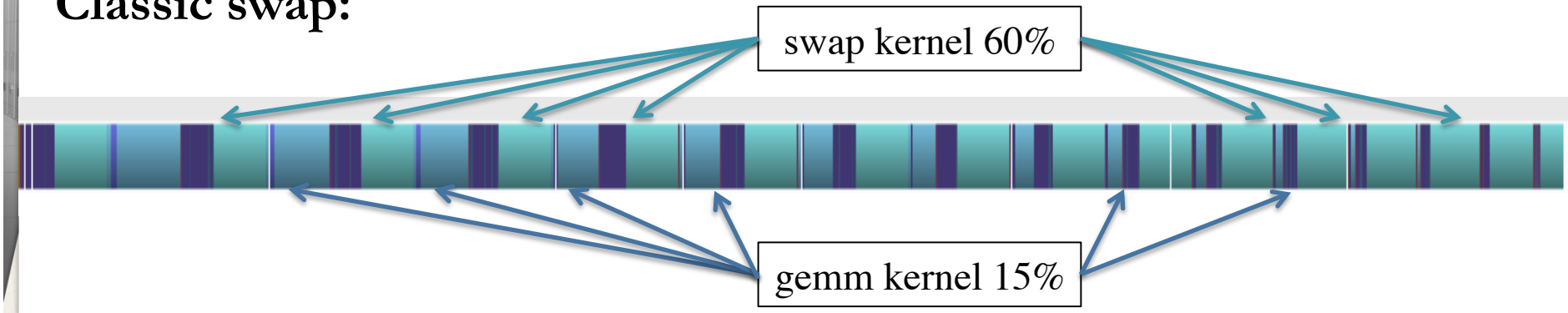
Classic swap:



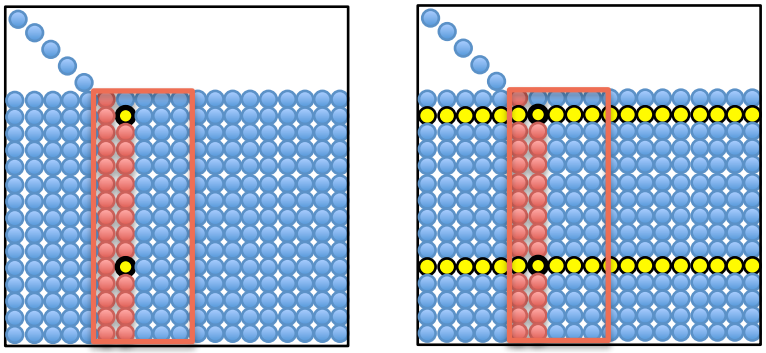
How does the swap work?



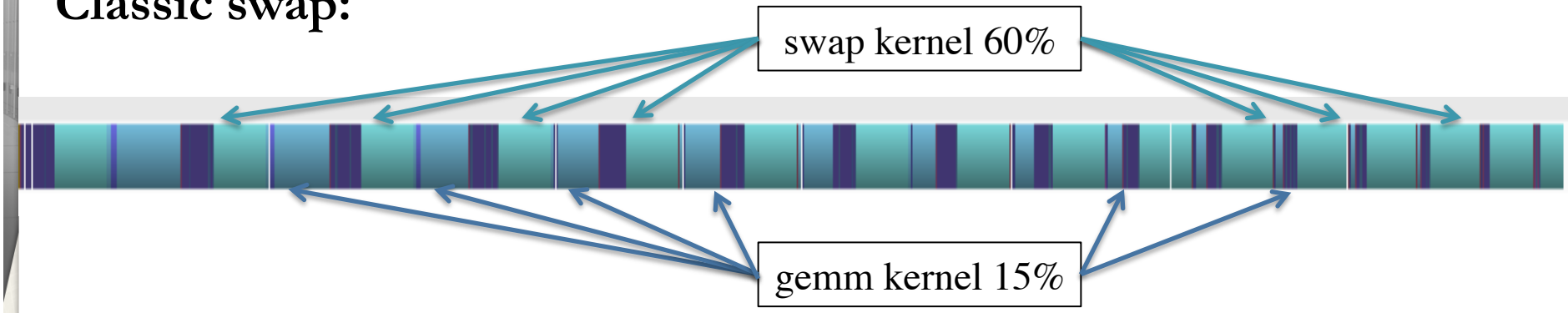
Classic swap:



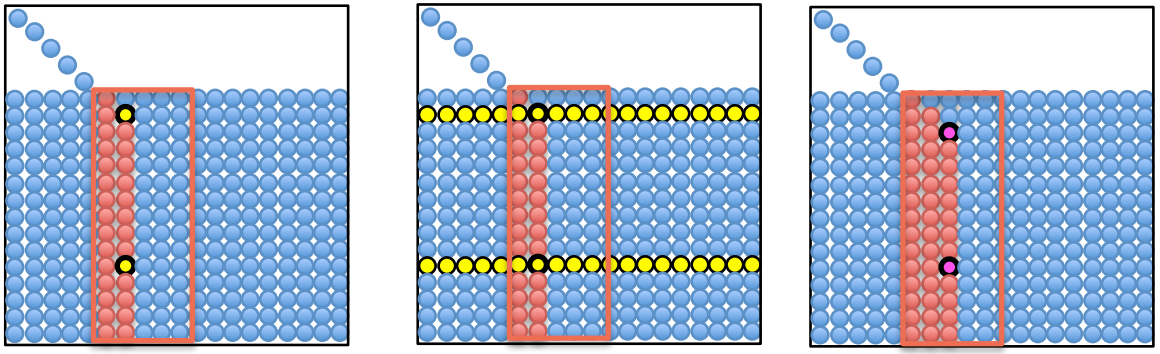
How does the swap work?



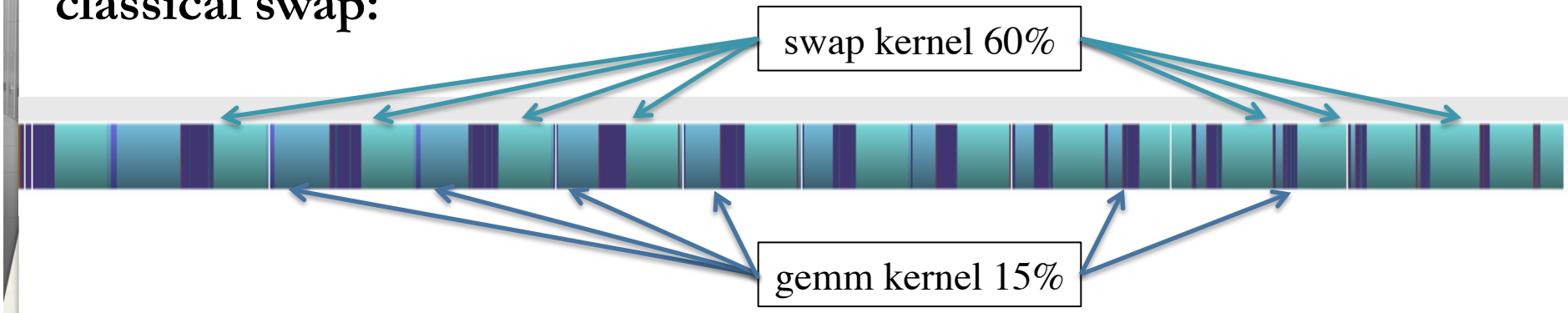
Classic swap:



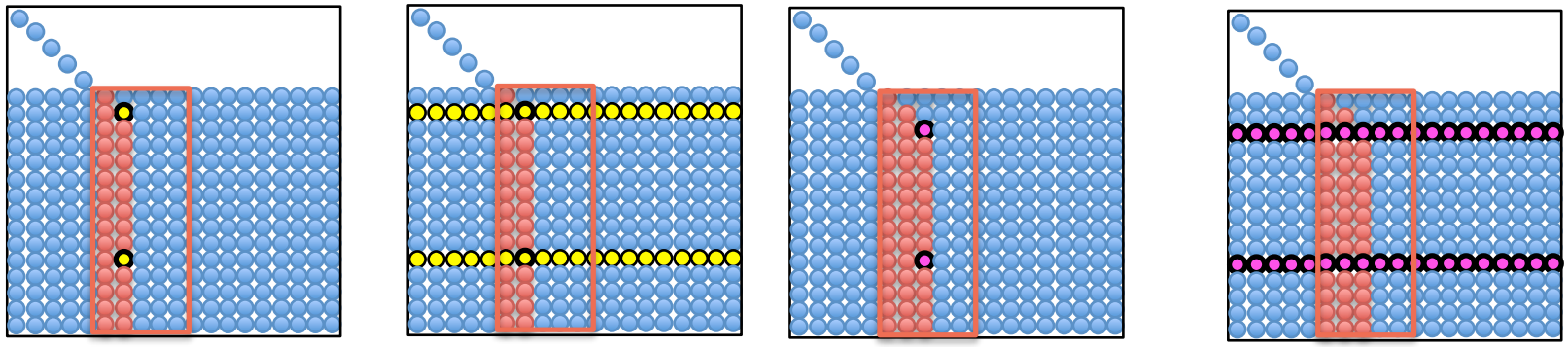
How does the swap work?



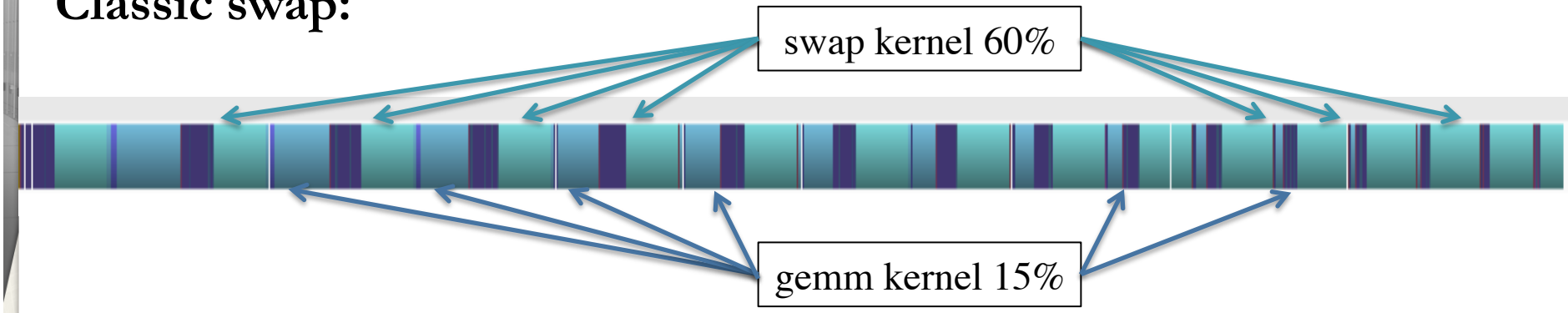
classical swap:



How does the swap work?



Classic swap:



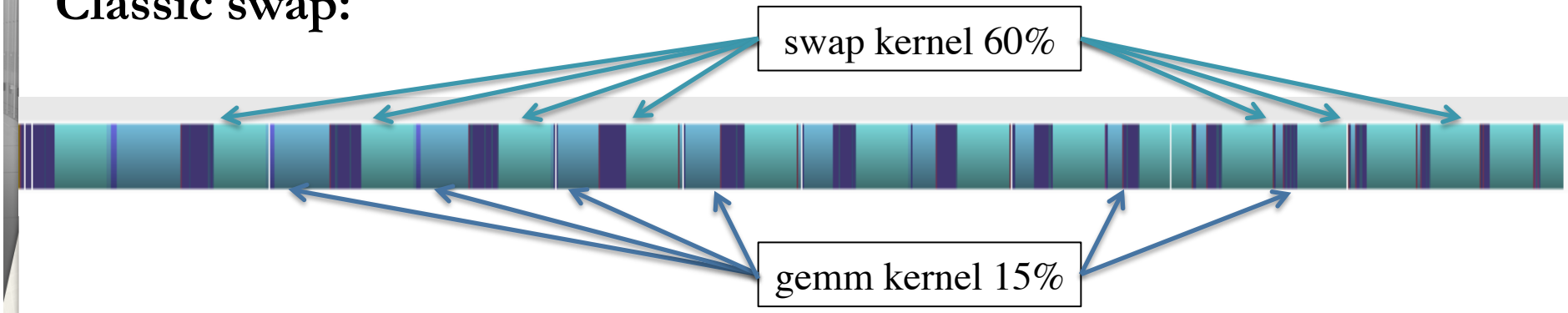
Bottlenecks:

- The swapping consists of *nb* successive interchanges of two rows of the matrices (serial).
- Data reading is not coalescent: a GPU warp cannot read 32 value at the same time unless matrix is stored in transpose form. However if matrix is stored in transpose form the swap is fast BUT the other components become very slow.

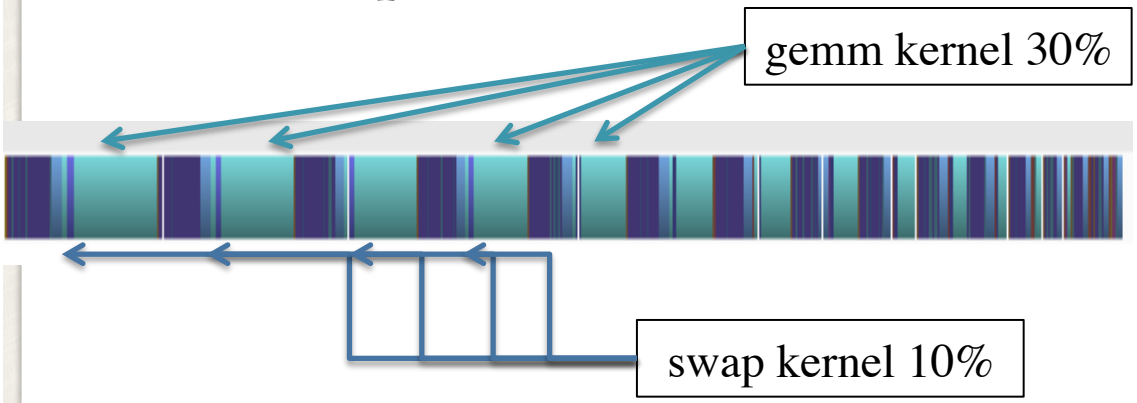
Proposition:

- We propose to modify the kernel to apply all ***nb*** row swaps in parallel
- This modification will also allow the coalescent write back of the top ***nb*** rows of the matrix
- Note that the top ***nb*** rows are those used by the ***dtrsm*** kernel that is applied right after the ***dlaswp***, so one optimization is to use shared memory to load a chunk of the ***nb*** rows

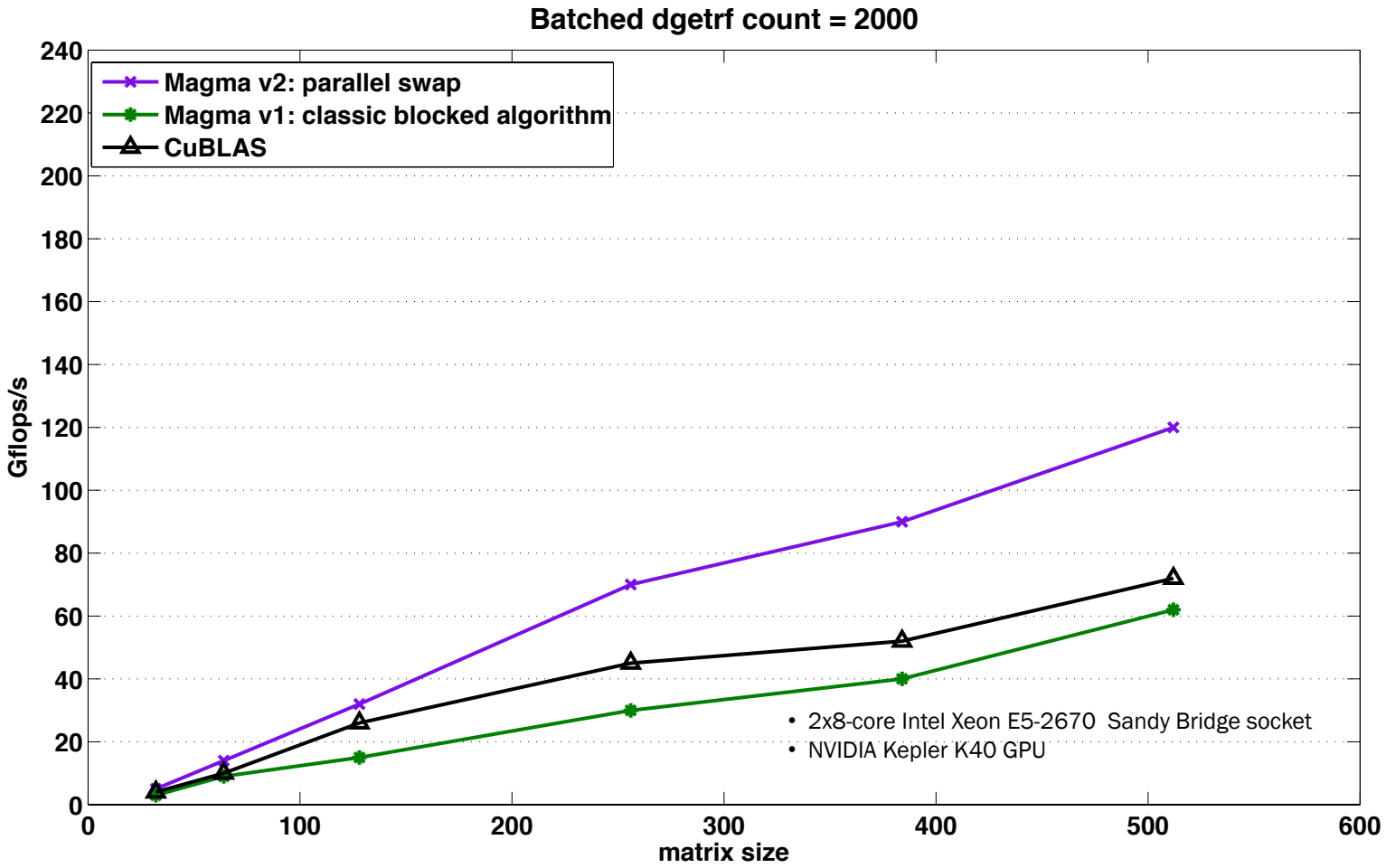
Classic swap:



Parallel swap:

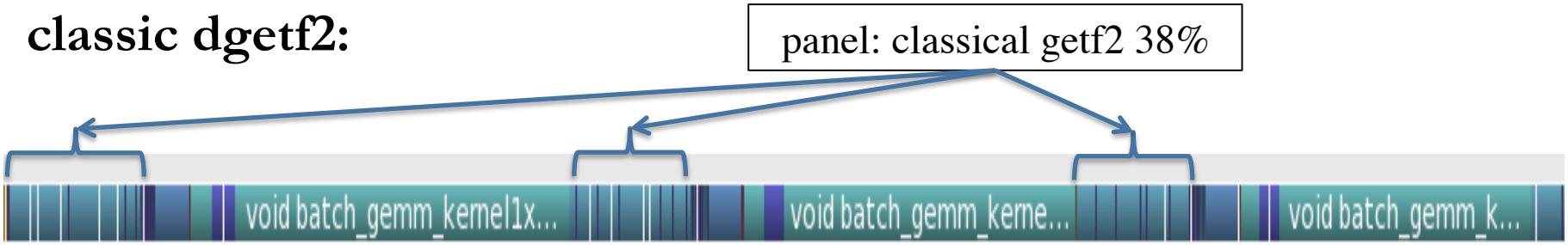


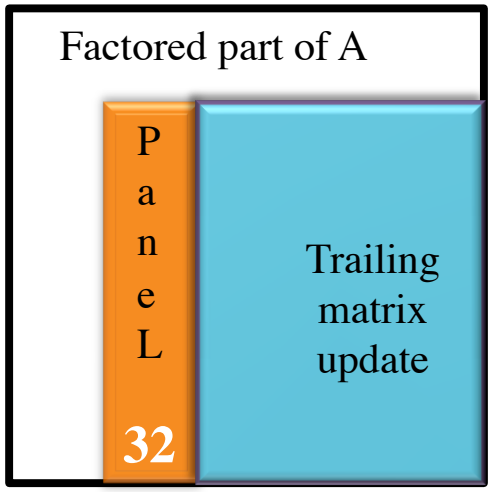
MAGMA Batched Computations



MAGMA Batched Computations

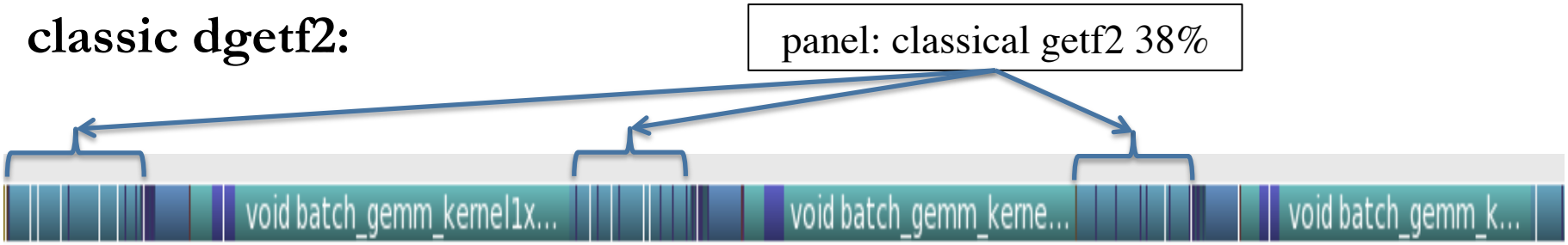
Panel factorization
classic dgetf2:

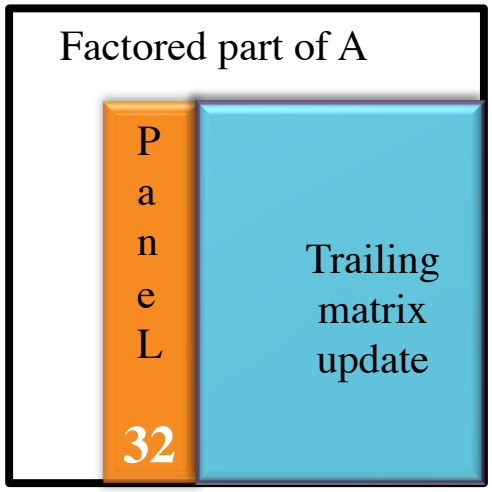




MAGMA Batched Computations

Panel factorization
classic dgetf2:





Bottlenecks:

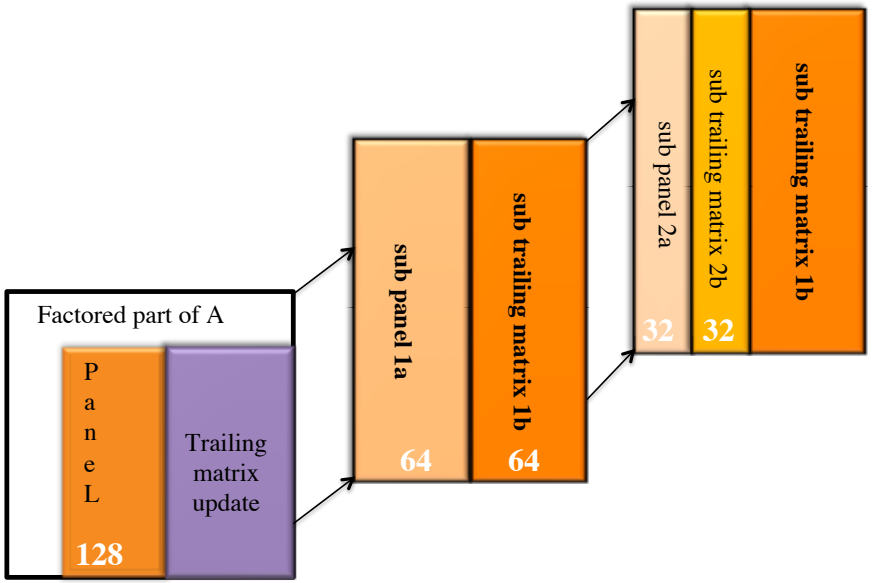
- *nb* large: panel get slower
  --> very bad performance.
- *nb* small: panel get faster but the update is not anymore efficient since dealing with gemm's of small sizes
  --> very bad performance.
- trade-off ? No effect, since we are talking about small size.

Proposition:

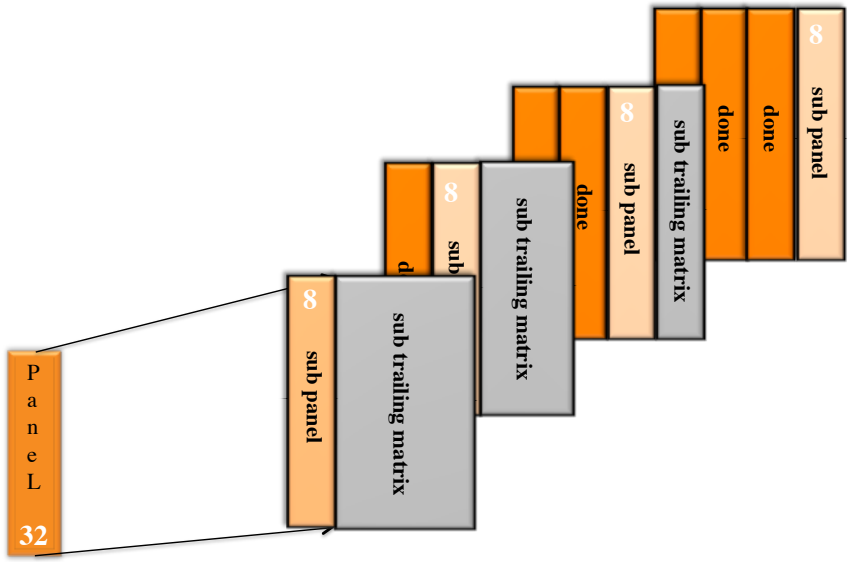
- We propose to develop two layers blocking: a recursive and nested blocking technique that block also the panel.

MAGMA Batched Computations

Two-layers blocking:



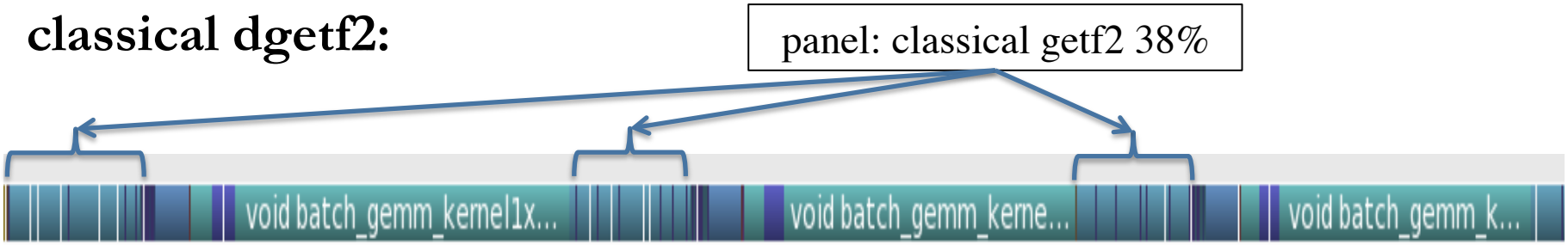
(a) Recursive nested blocking fashion.



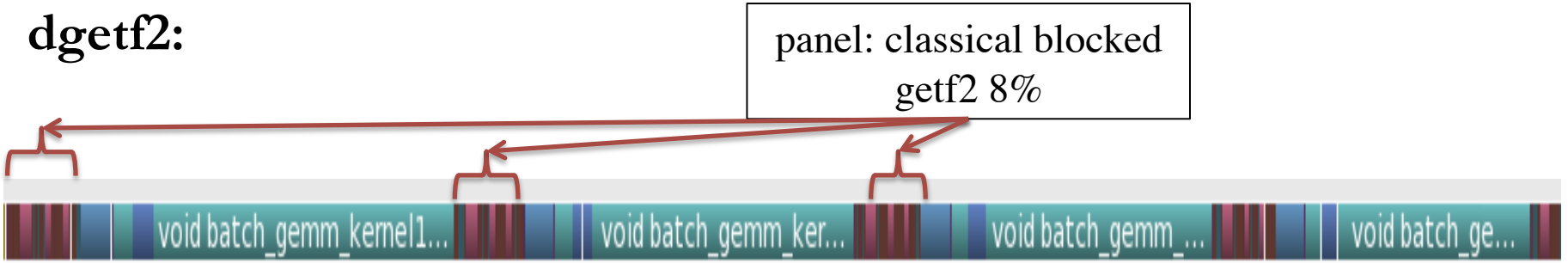
(b) Classical blocking fashion.

MAGMA Batched Computations

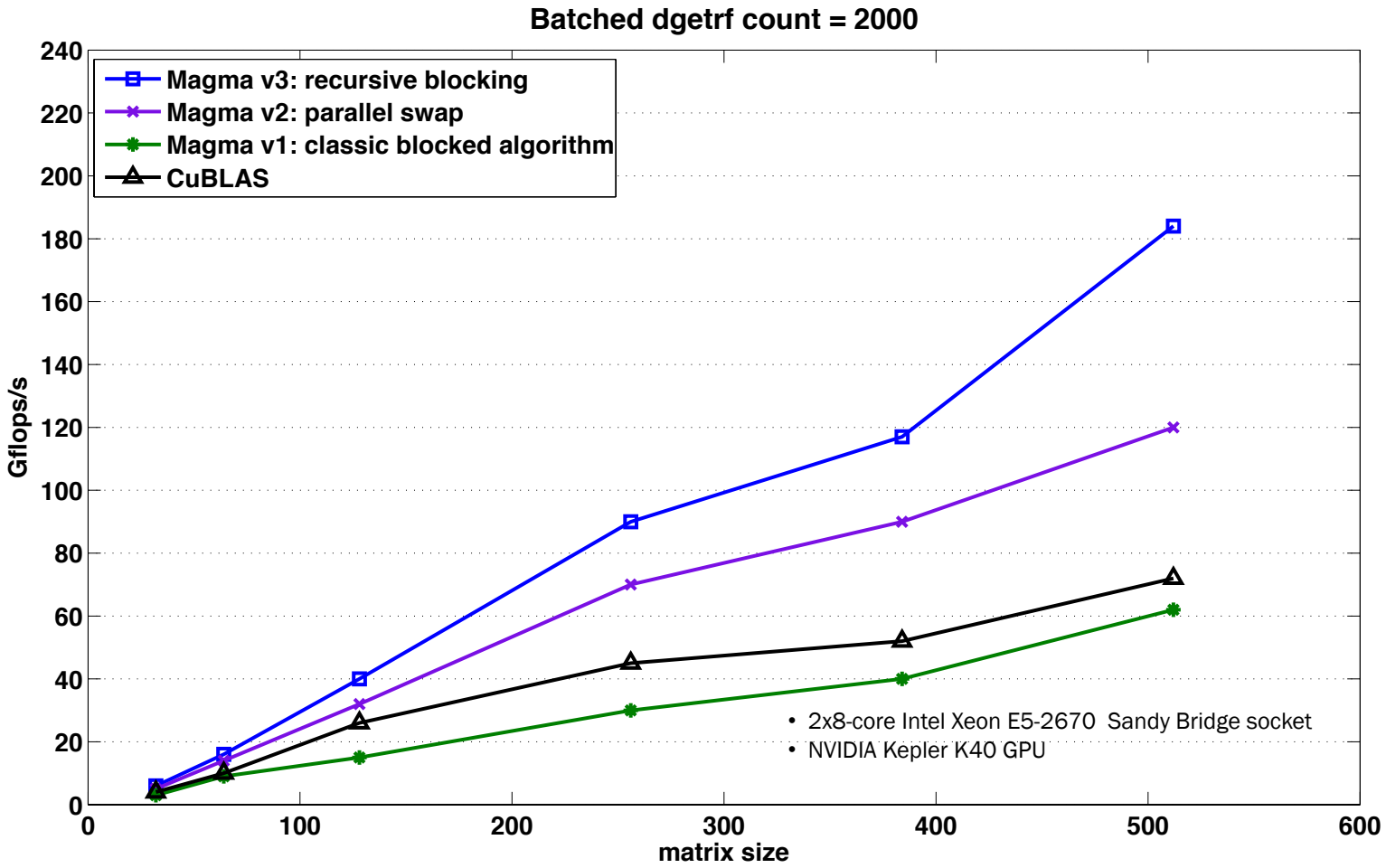
panel factorization
classical dgetf2:



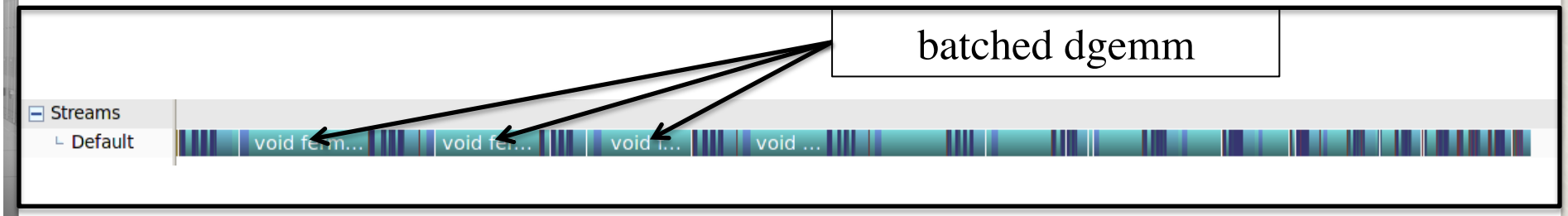
Recursive blocking of dgetf2:



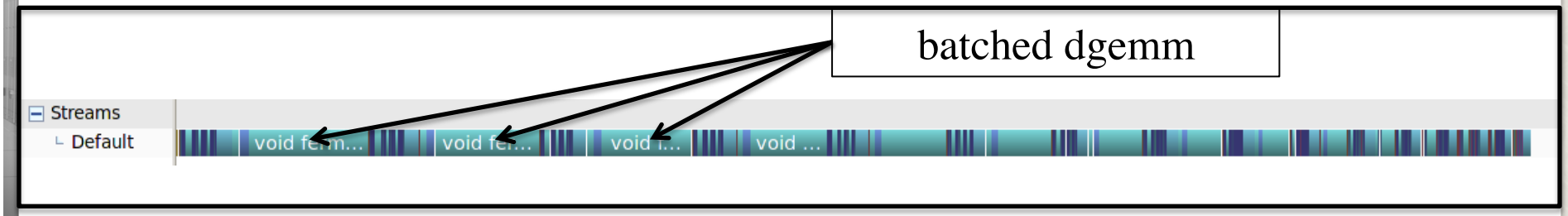
MAGMA Batched Computations

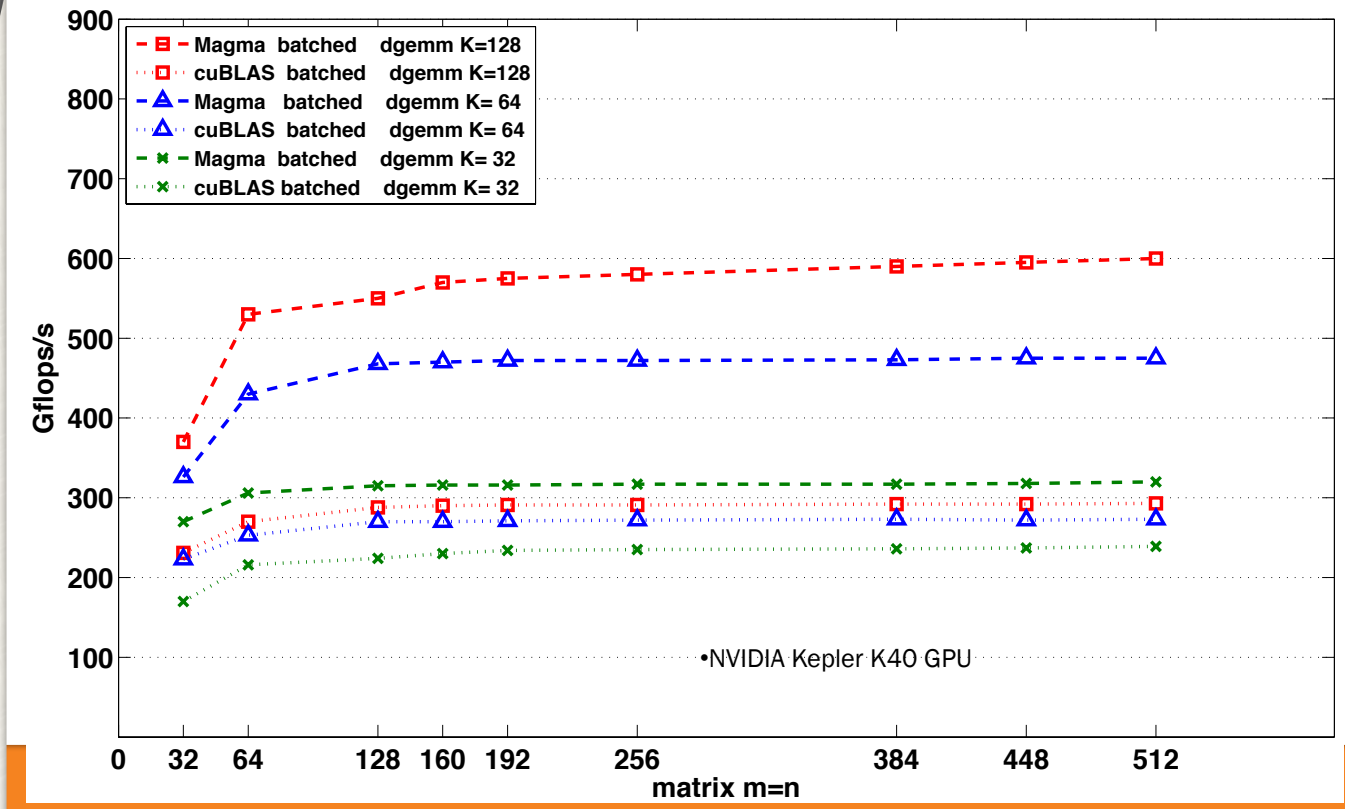


MAGMA Batched Computations

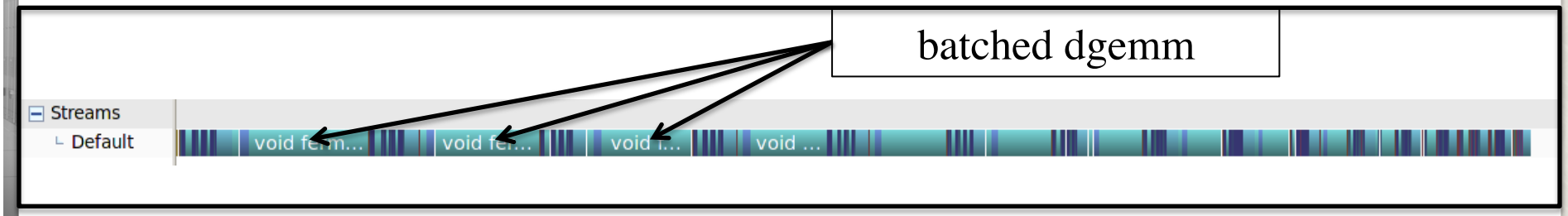


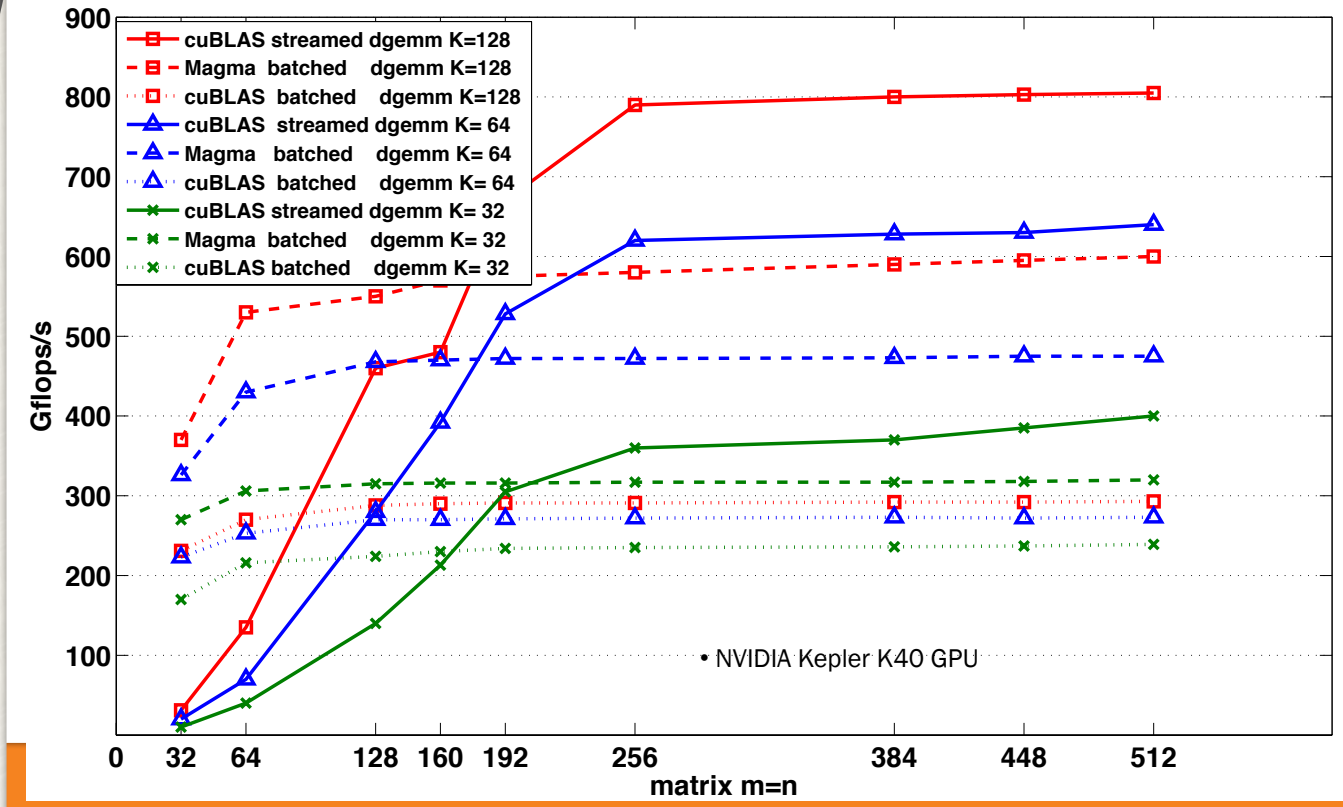
MAGMA Batched Computations





MAGMA Batched Computations

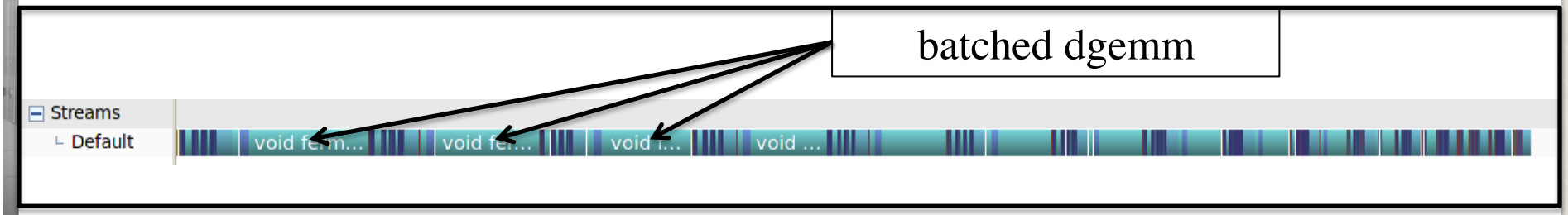


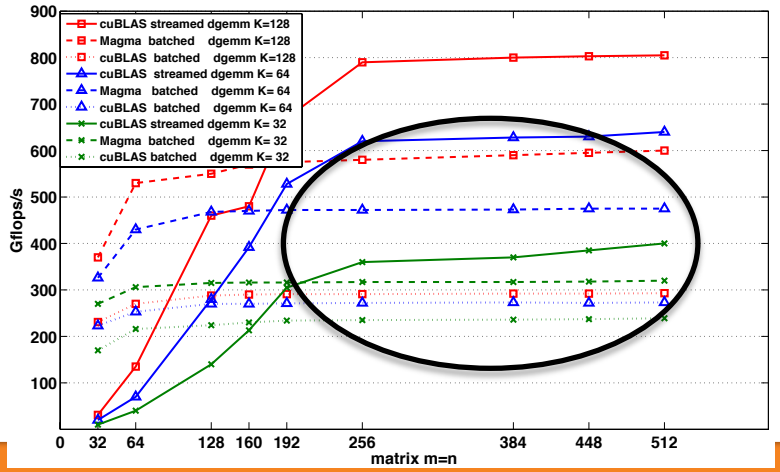


ICL

THE UNIVERSITY OF TENNESSEE KNOXVILLE

MAGMA Batched Computations





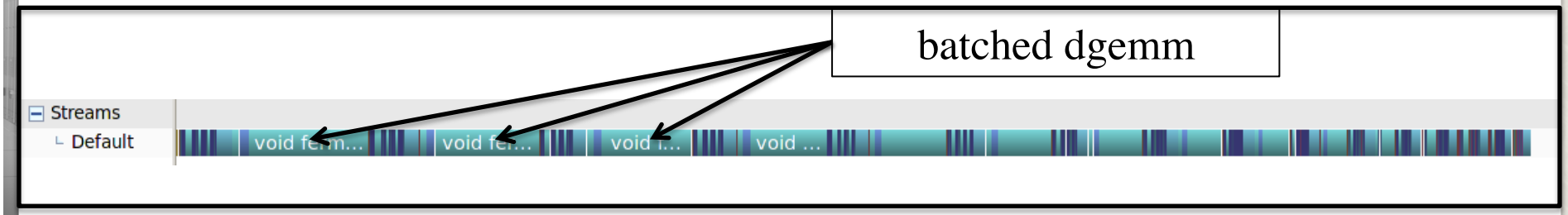
Bottlenecks:

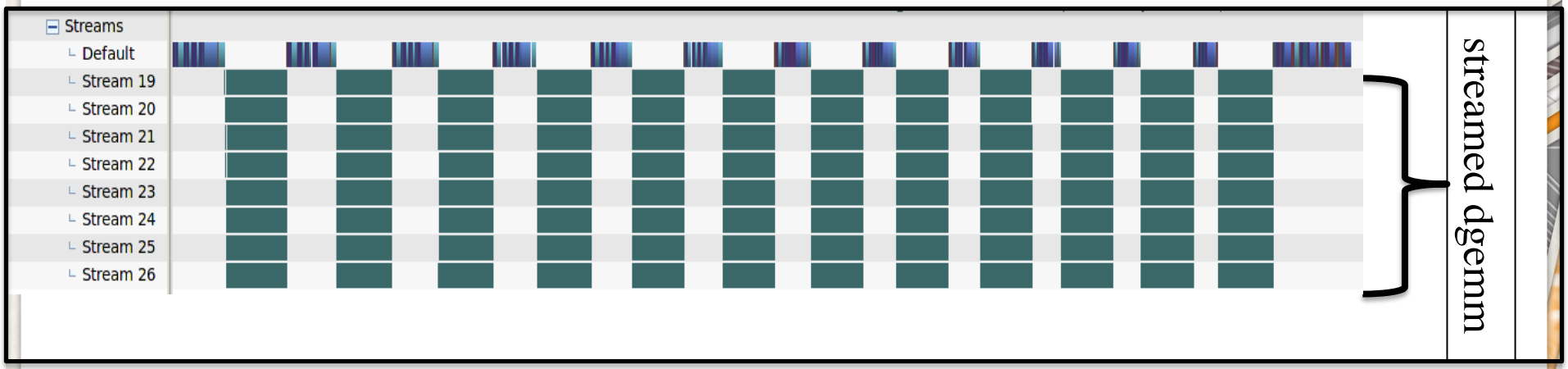
- Batched gemm kernel from cuBLAS and Magma are well suited for small matrix sizes (128) but stagnate for larger sizes (>128)

Proposition:

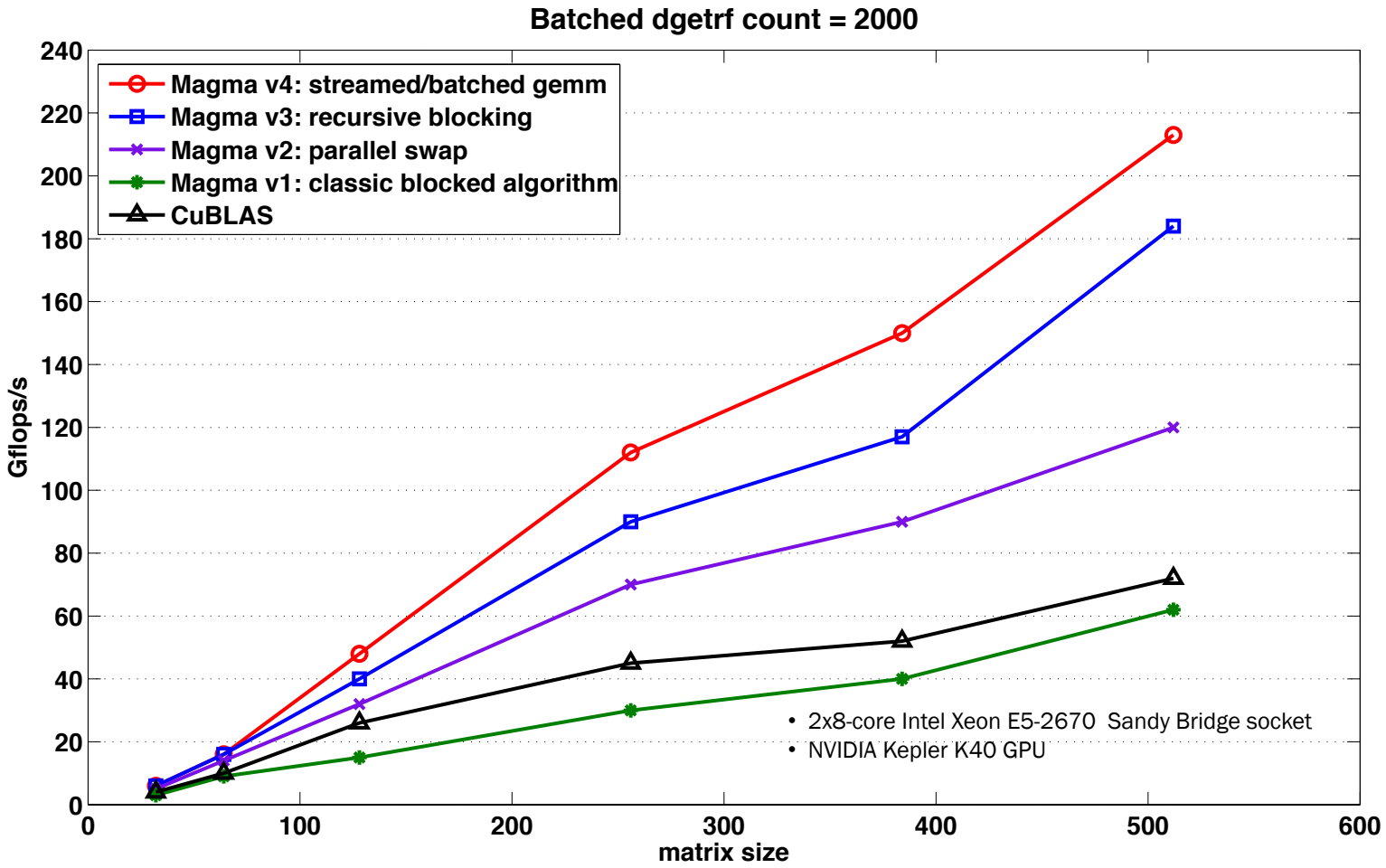
- Autotune Magma GEMM for small size and provide a low level API that can be used from inside the kernels as well as try to use streamed whenever appropriate

MAGMA Batched Computations





MAGMA Batched Computations



MAGMA Batched Computations

Comparison with CPU:

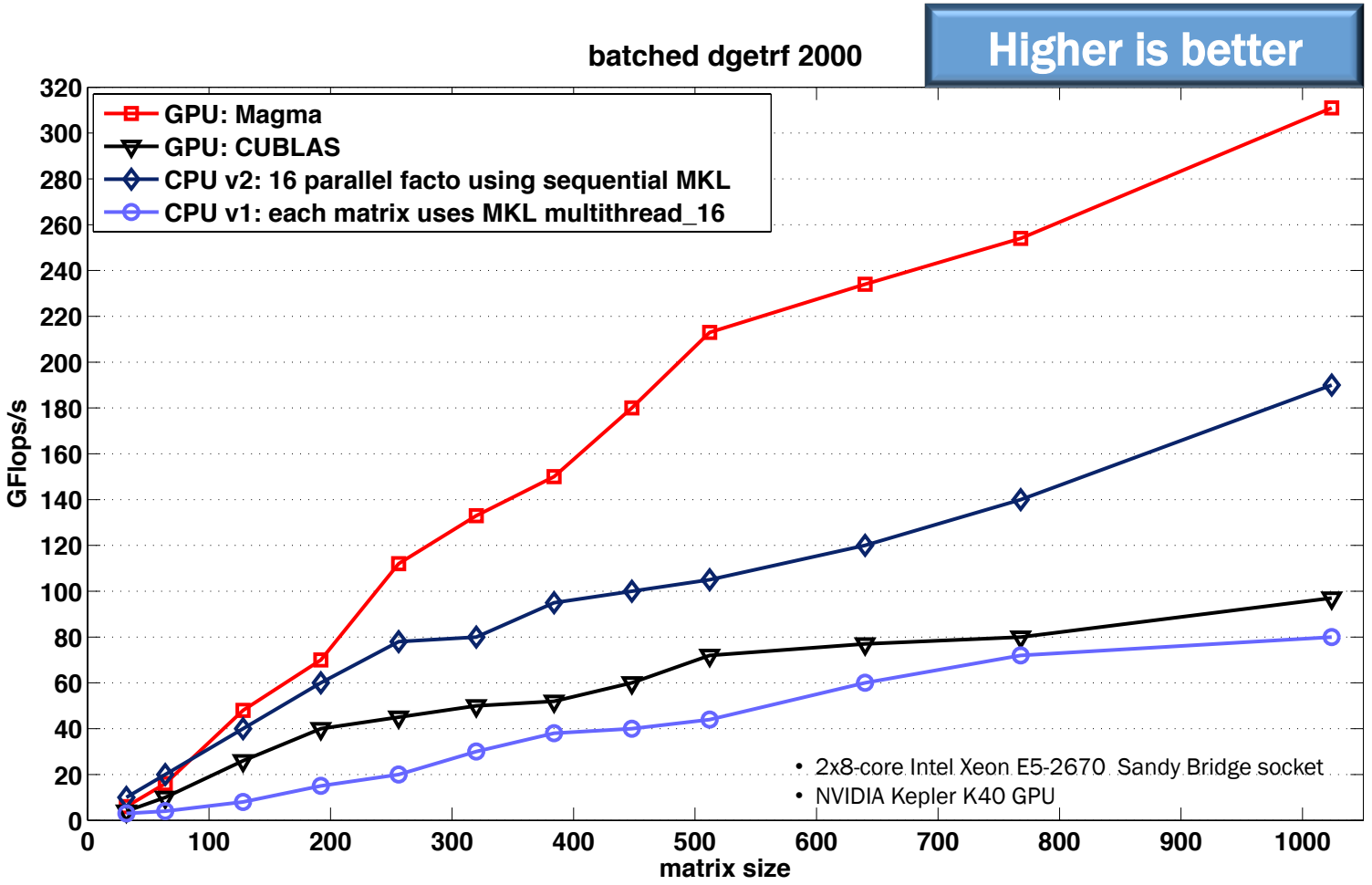
- **Version 1:** The simple CPU implementation is to go in a loop fashion to factorize matrix after matrix, where each factorization is using the multi-thread version of the MKL Library.

  Expected to have low in performance because each matrix is small – it does not exhibit parallelism and so the multithreaded code is not able to feed with work all 16 SB threads used.

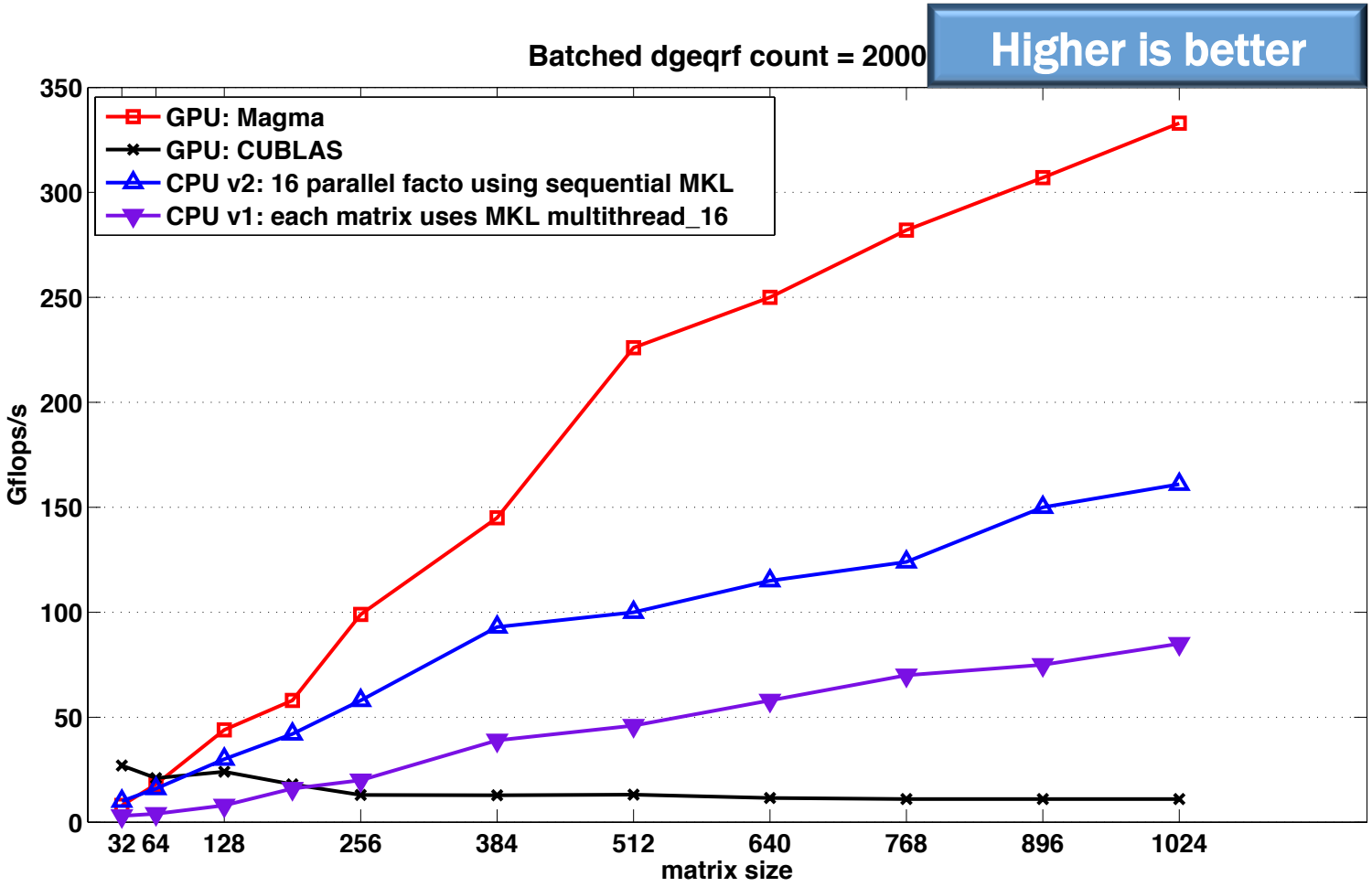
- **Version 2:** for that we proposed another version of the CPU implementation. Since the matrices are small (< 512) and at least 16 of them fit in the L3 cache level.

  One of the best technique is to use each thread to factorize independently a matrix. This way 16 factorizations are conducted independently in parallel.

MAGMA Batched Computations

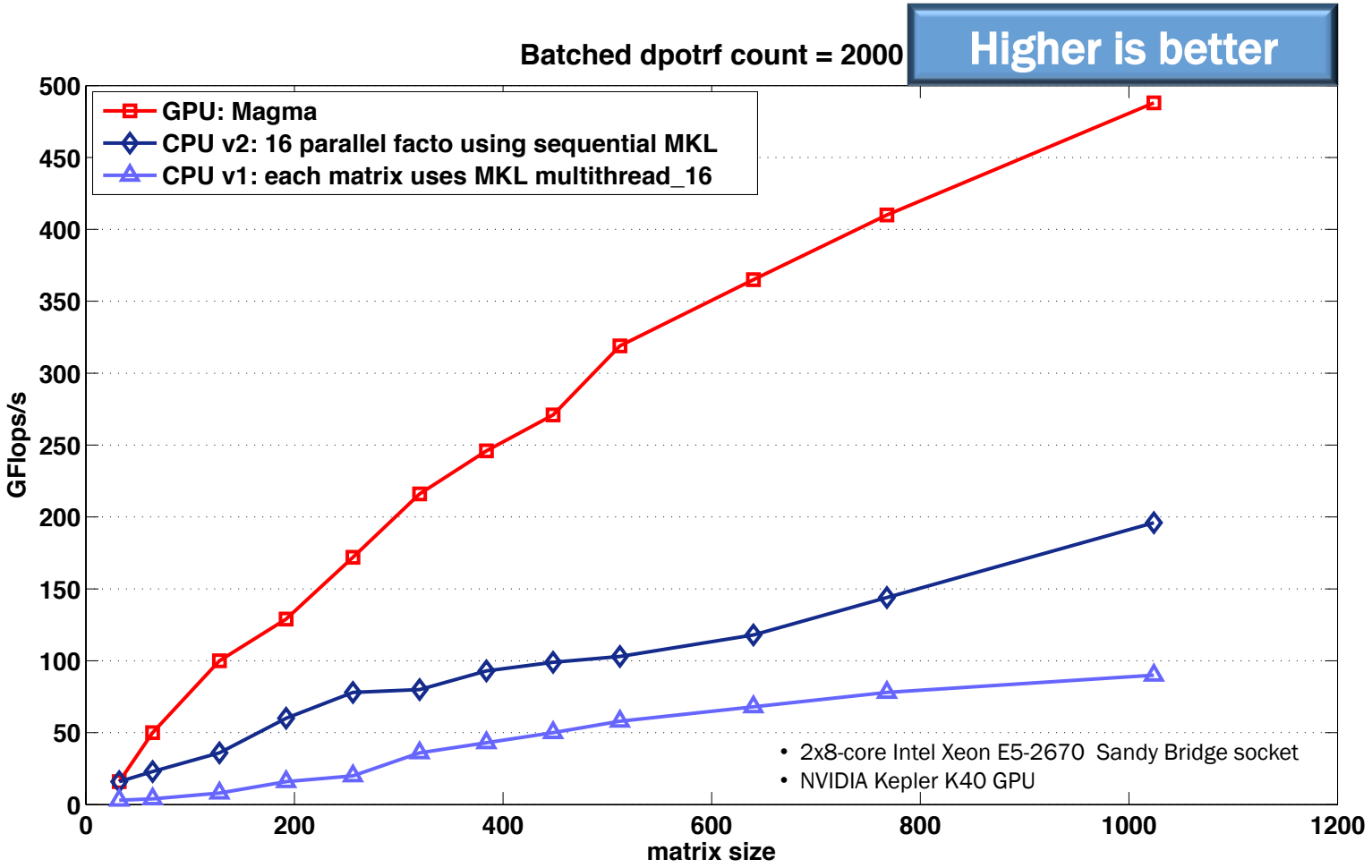


MAGMA Batched Computations



- 2x8-core Intel Xeon E5-2670 Sandy Bridge socket
- NVIDIA Kepler K40 GPU

MAGMA Batched Computations



MAGMA Batched Computations

Summary

- Batched computation can give a boost in performance for problem with very small sizes
- Traditional algorithmic design might not be the best direction
  - we need a new way of thinking
  - revisit and redesign algorithm to take advantage of the hardware specifics
- Performance model can help analyzing algorithm and their implementation, for example
  - An optimized GPU function cannot be efficient for all kind of computation, it depend on the context used for
  - Small computation are delicate and requires specific kernels (building block or fused).
  - Low level API is required to avoid overhead and context switching

Future Directions

- Extended functionality
  - Variable sizes
  - Dynamics scheduling
  - Sparse direct multifrontal solvers & preconditioners
  - Applications
- Further tuning
  - autotuning

Collaborators / Support

- MAGMA and Batched Magma [Matrix Algebra on GPU and Multicore Architectures] team http://icl.cs.utk.edu/magma/
- PLASMA [Parallel Linear Algebra for Scalable Multicore Architectures] team http://icl.cs.utk.edu/plasma
- Collaborating partners
  - University of Tennessee, Knoxville
  - University of California, Berkeley
  - University of Colorado, Denver
  - INRIA, France
  - KAUST, Saudi Arabia

References

• **A. Haidar, S. Tomov, P. Luszczek, and J. Dongarra.**
MAGMA Embedded: Towards a Dense Linear Algebra Library for Energy Efficient Extreme Computing
IEEE High Performance Extreme Computing Conference IEEE-HPEC 2015, Waltham, MA USA.
Best paper finalist decision in September 2015

• **A. Haidar, T. Dong, S. Tomov, P. Luszczek, and J. Dongarra.**
A Framework for Batched and GPU-resident Factorization Algorithms Applied to Block Householder Transformations
International SuperComputing Conference IEEE-ISC 2015, Frankfurt, Germany.
Springer, Lecture Notes in Computer Science Volume 9137, 2015, pp 31-47

• **K. Kabir, A. Haidar, S. Tomov, and J. Dongarra**
On the Design, Development and Analysis of Optimized Matrix-Vector Multiplication Routines for coprocessors.
International SuperComputing Conference IEEE-ISC 2015, Frankfurt, Germany
Springer, Lecture Notes in Computer Science Volume 9137, 2015, pp 58-73

• **A. Haidar, T. Dong, P. Luszczek, S. Tomov and J. Dongarra.**
Batched Matrix Computations on Hardware Accelerators Based on GPUs.
International Journal of High Performance Computing Applications 2014.

• **A. Haidar, T. Dong, S. Tomov, P. Luszczek, and J. Dongarra.**
Optimization for Performance and Energy for Batched Matrix Computations on GPUs
20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Workshop on General Purpose Processing Using GPUs. GPGPU/PPoPP 2015.

• **T. Dong, A. Haidar, S. Tomov, and J. Dongarra.**
A Fast Batched Cholesky Factorization on a GPU
ICPP 2014, The 43rd International Conference on Parallel Processing 2014.