Architecture-aware Algorithms and Software for Peta and Exascale Computing

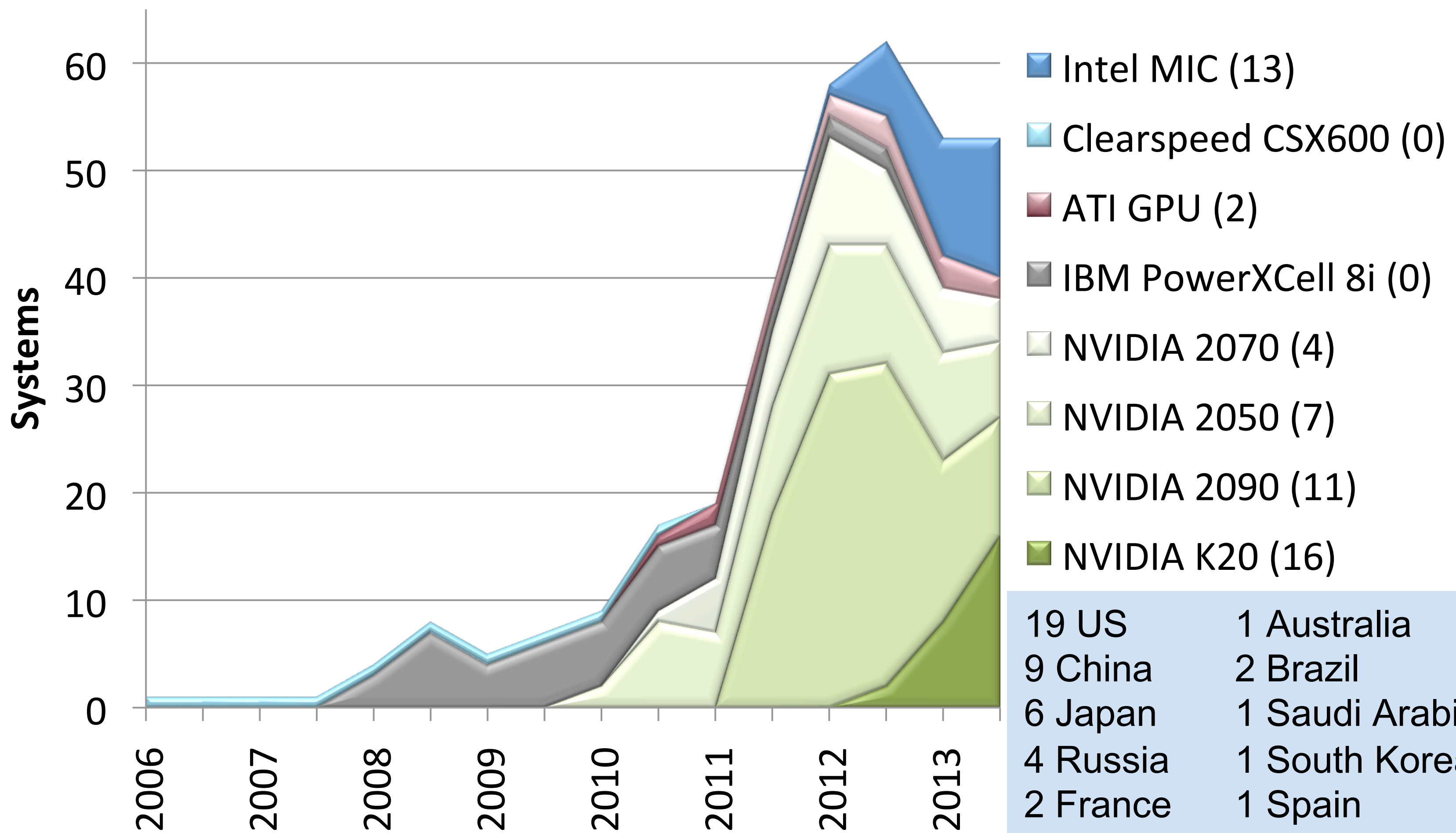
Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory
University of Manchester

November 2013: The TOP10

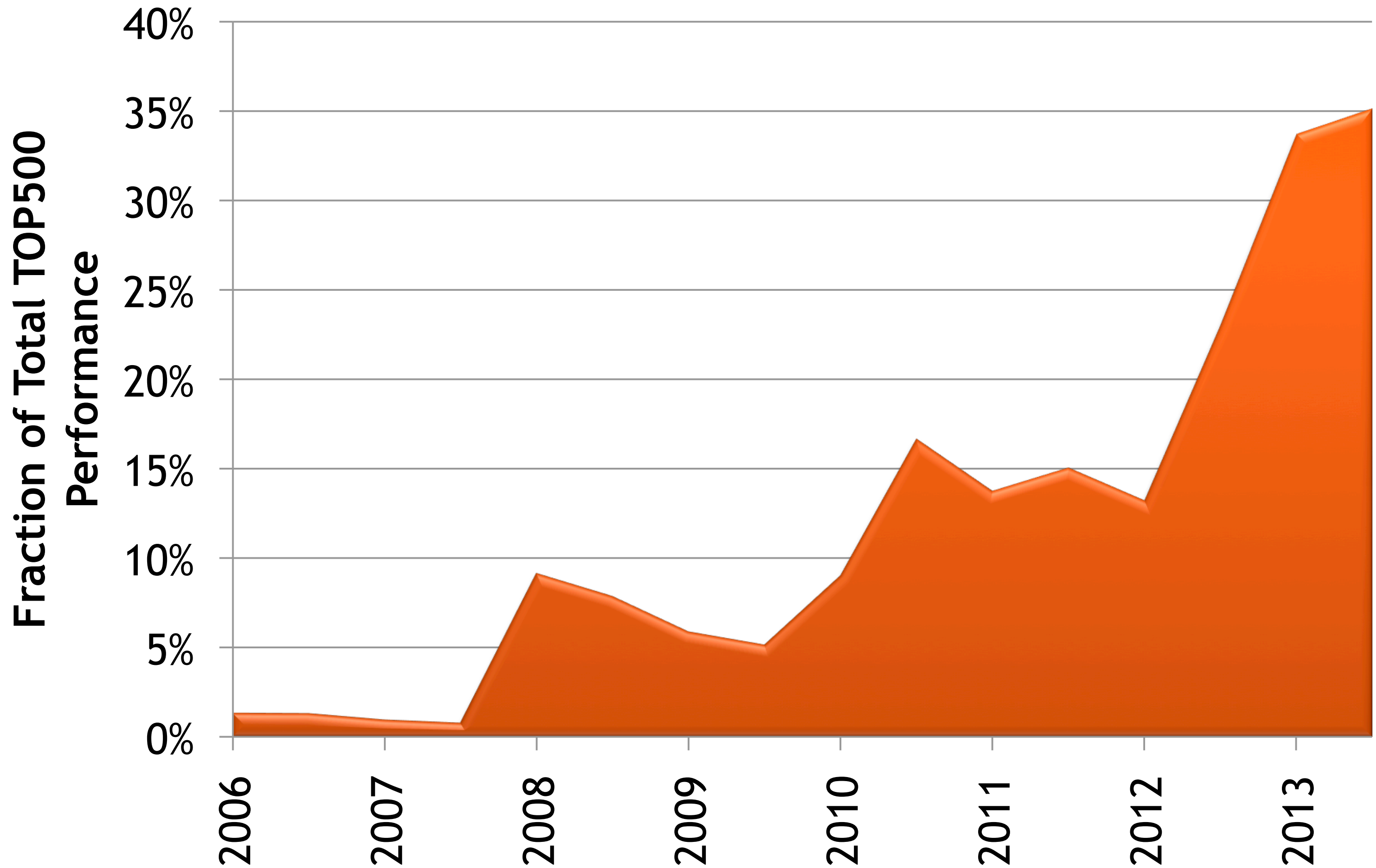
Rank	Site	Computer	Country	Cores	Rmax [Pflops]	% of Peak	Power [MW]	MFlops /Watt
1	National University of Defense Technology	Tianhe-2 NUDT, Xeon 12C 2.2GHz + IntelXeon Phi (57c) + Custom	China	3,120,000	33.9	62	17.8	1905
2	DOE / OS Oak Ridge Nat Lab	Titan, Cray XK7 (16C) + Nvidia Kepler GPU (14c) + Custom	USA	560,640	17.6	65	8.3	2120
3	DOE / NNSA L Livermore Nat Lab	Sequoia, BlueGene/Q (16c) + custom	USA	1,572,864	17.2	85	7.9	2063
4	RIKEN Advanced Inst for Comp Sci	K computer Fujitsu SPARC64 VIIIfx (8c) + Custom	Japan	705,024	10.5	93	12.7	827
5	DOE / OS Argonne Nat Lab	Mira, BlueGene/Q (16c) + Custom	USA	786,432	8.16	85	3.95	2066
6	Swiss CSCS	Piz Daint, Cray XC30, Xeon 8C + Nvidia Kepler (14c) + Custom	Swiss	115,984	6.27	81	2.3	2726
7	Texas Advanced Computing Center	Stampede, Dell Intel (8c) + Intel Xeon Phi (61c) + IB	USA	204,900	2.66	61	3.3	806
8	Forschungszentrum Juelich (FZJ)	JuQUEEN, BlueGene/Q, Power BQC 16C 1.6GHz+Custom	Germany	458,752	5.01	85	2.30	2178
9	DOE / NNSA L Livermore Nat Lab	Vulcan, BlueGene/Q, Power BQC 16C 1.6GHz+Custom	USA	393,216	4.29	85	1.97	2177
10	Leibniz Rechenzentrum	SuperMUC, Intel (8c) + IB	Germany	147,456	2.90	91*	3.42	848
500	Banking	HP	USA	22,212	.118	50		

Accelerators (53 systems)

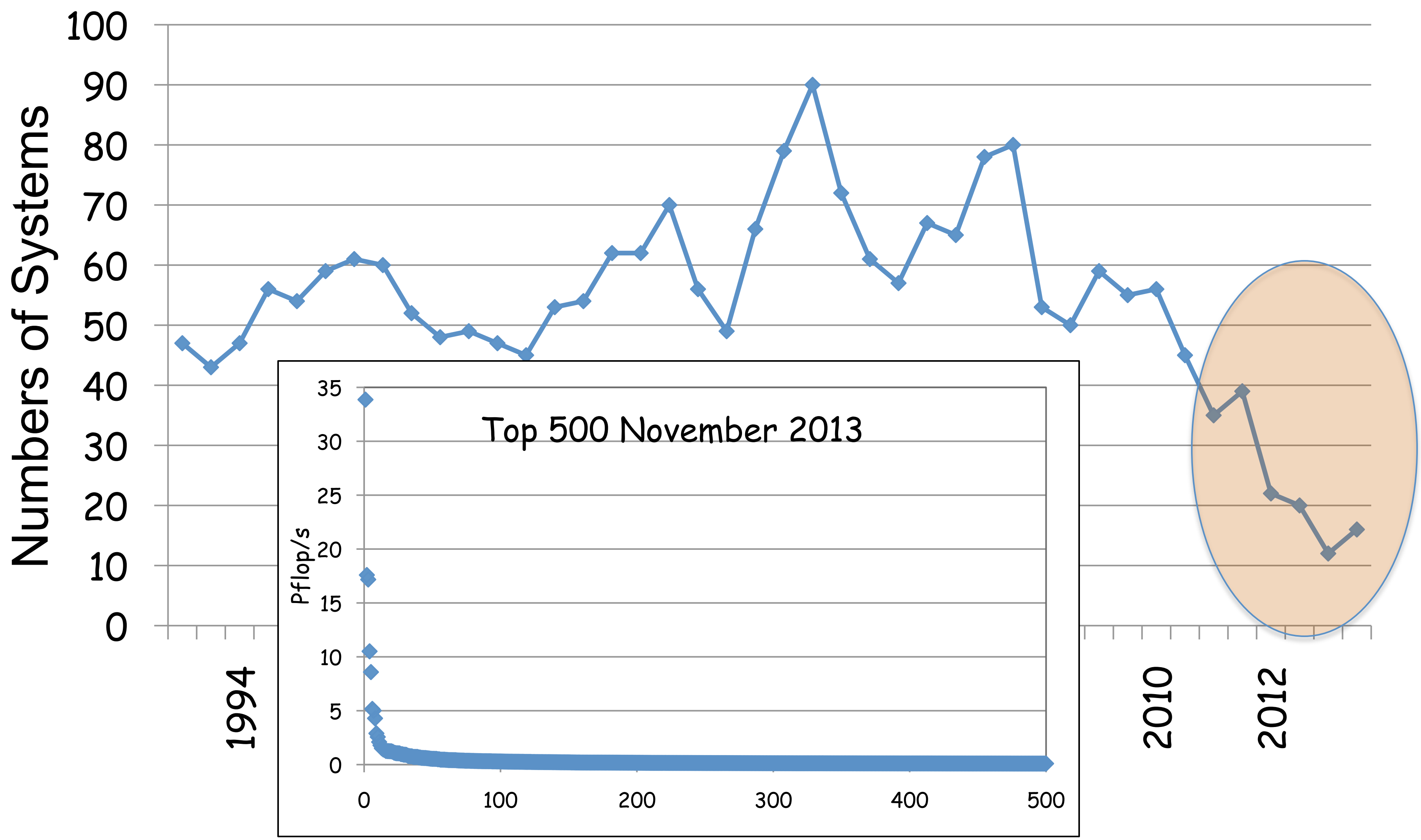


19 US	1 Australia
9 China	2 Brazil
6 Japan	1 Saudi Arabia
4 Russia	1 South Korea
2 France	1 Spain
2 Germany	2 Switzerland
2 India	1 UK
1 Italy	
1 Poland	

Top500 Performance Share of Accelerators



For the Top 500: Rank at which Half of Total Performance is Accumulated



Commodity plus Accelerator Today

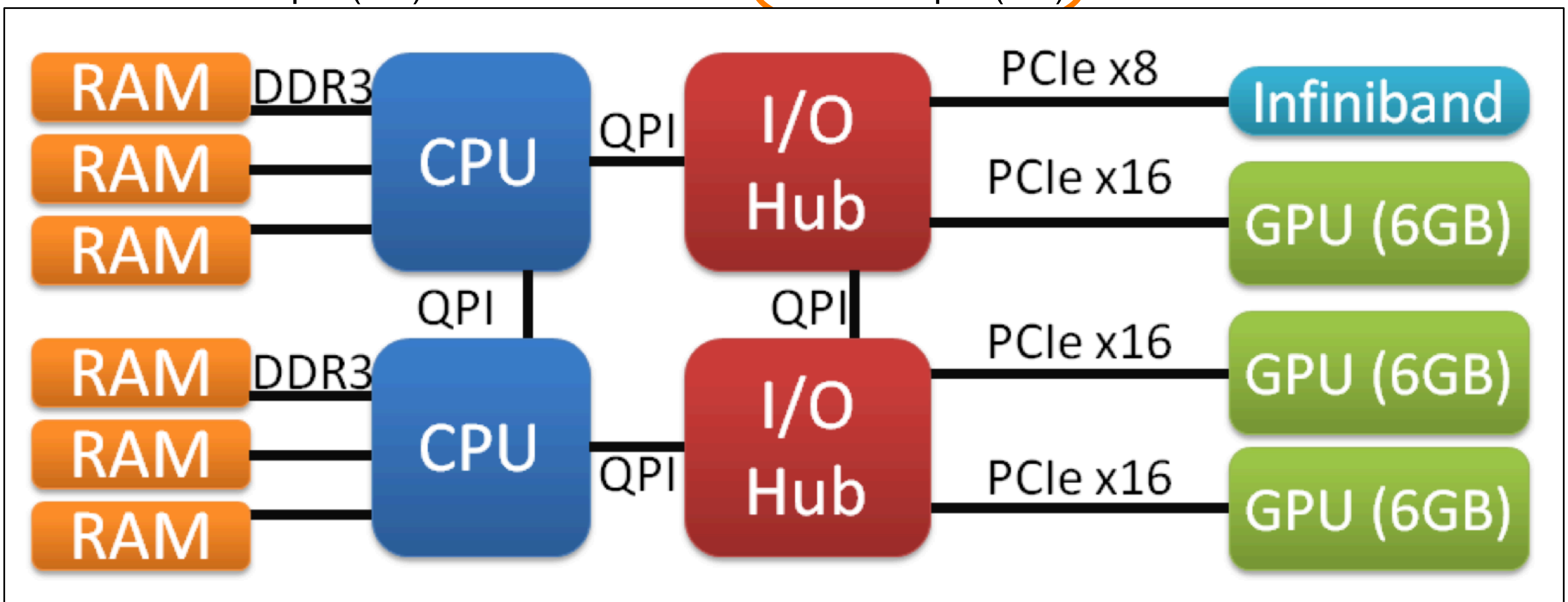
Commodity

Intel Xeon
8 cores
3 GHz
8*4 ops/cycle
96 Gflop/s (DP)

Accelerator (GPU)

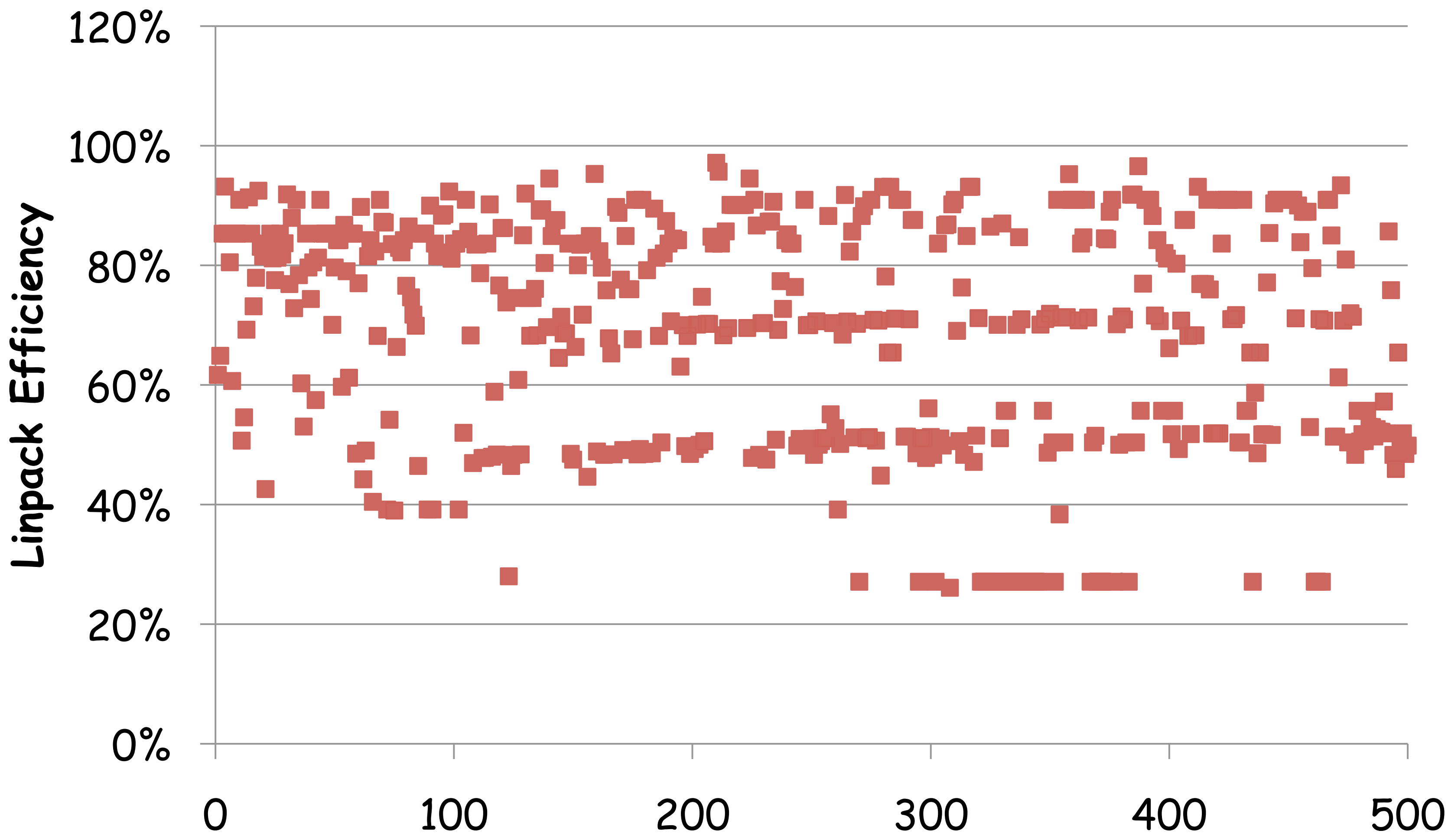
Nvidia K20X “Kepler”
2688 “Cuda cores”
.732 GHz
2688*2/3 ops/cycle
1.31 Tflop/s (DP)

192 Cuda cores/SMX
2688 “Cuda cores”

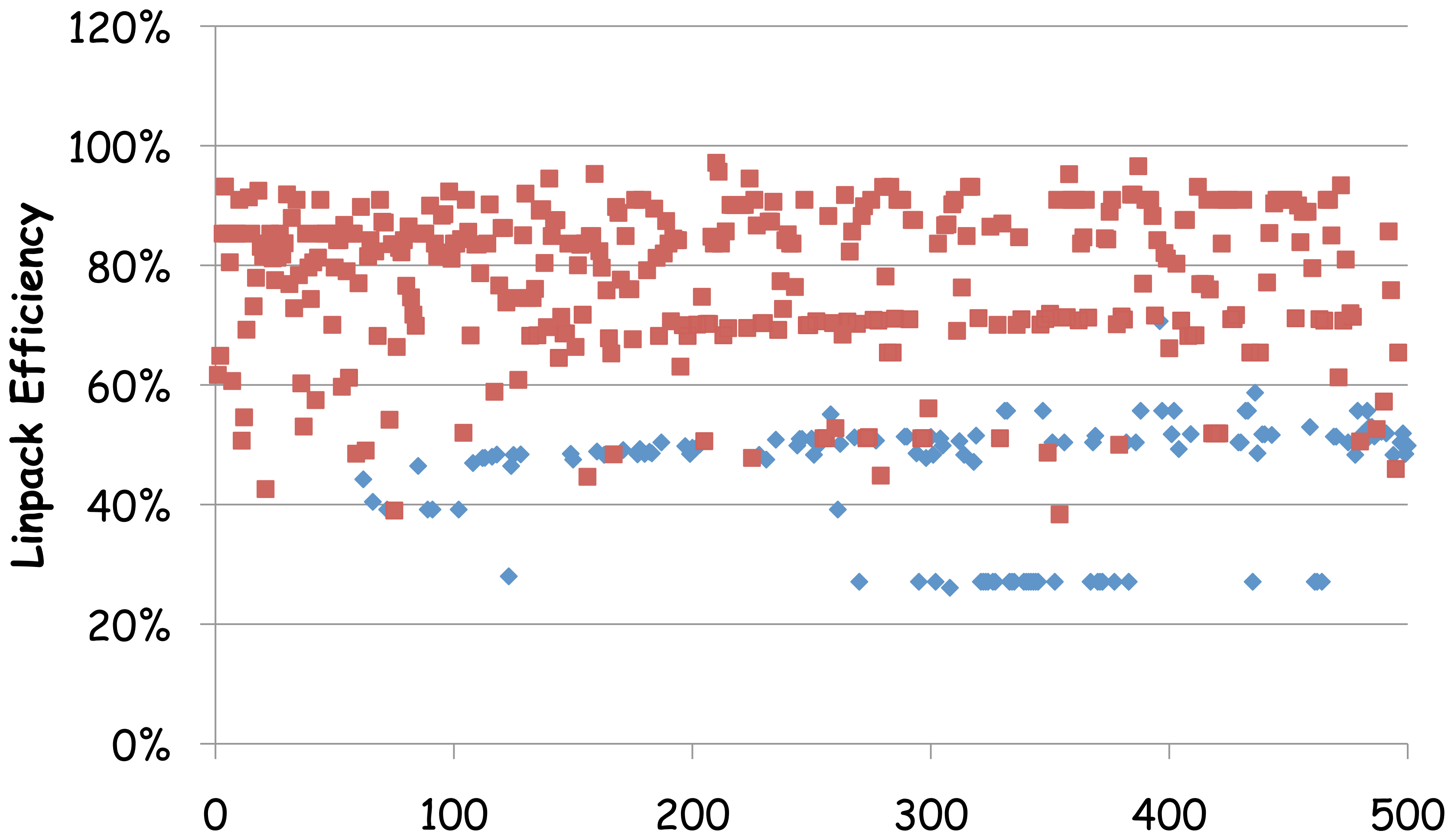


Interconnect
PCI-X 16 lane
64 Gb/s (8 GB/s)
1 GW/s

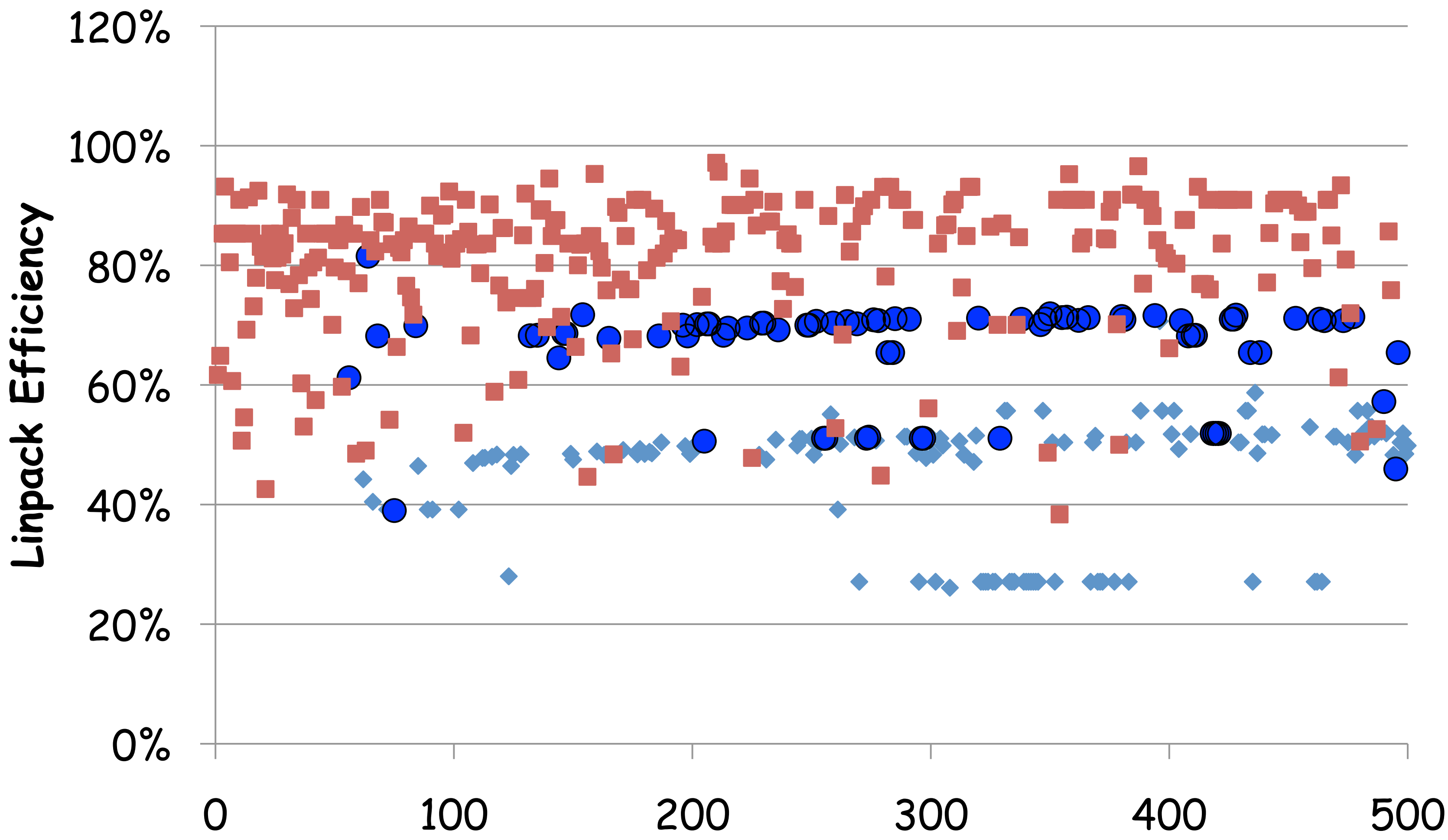
Linpack Efficiency



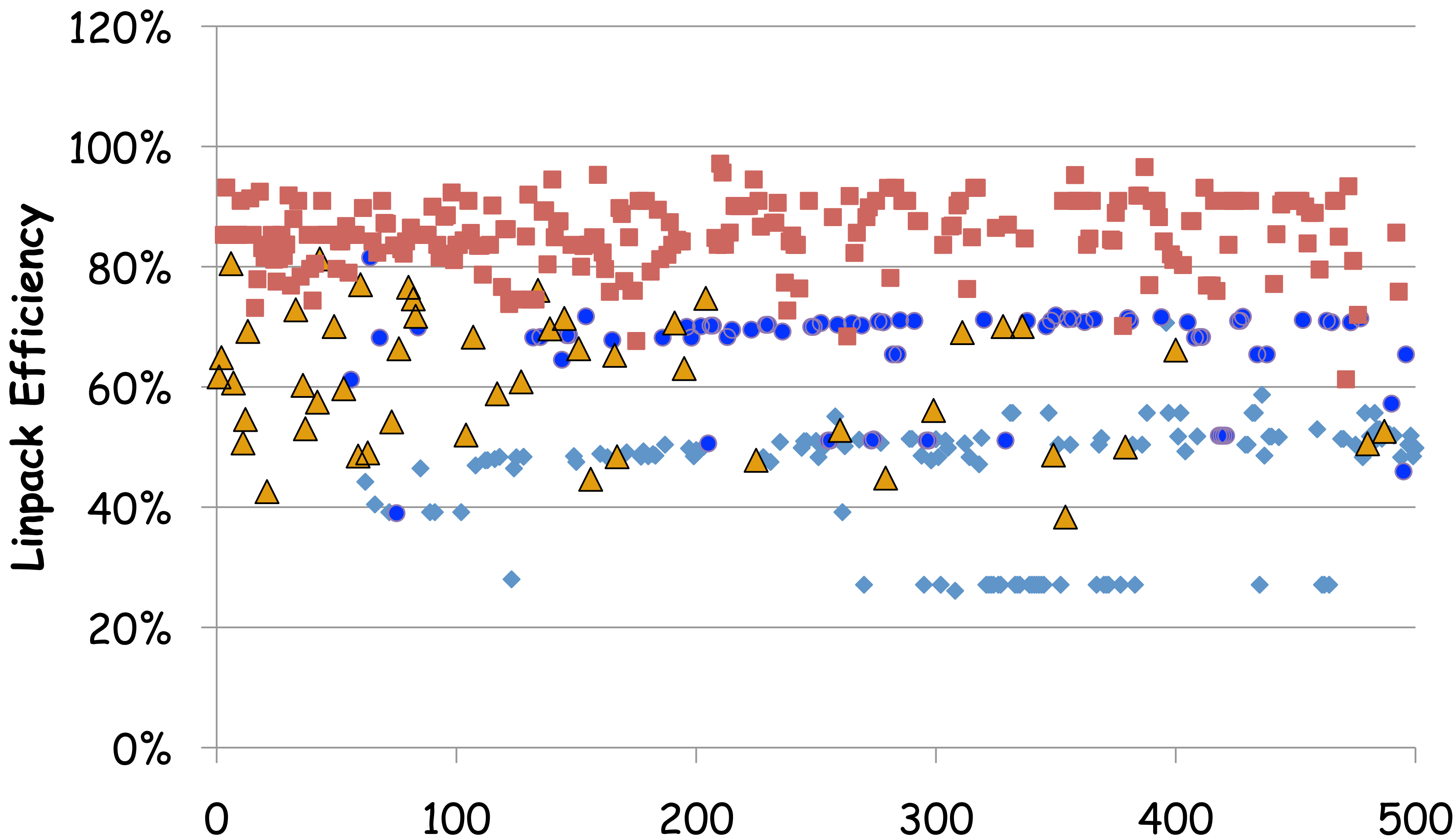
Linpack Efficiency



Linpack Efficiency



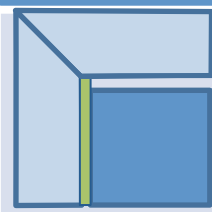
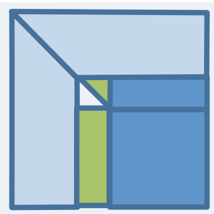
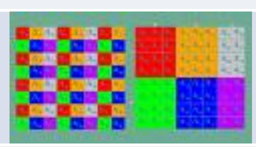
Linpack Efficiency

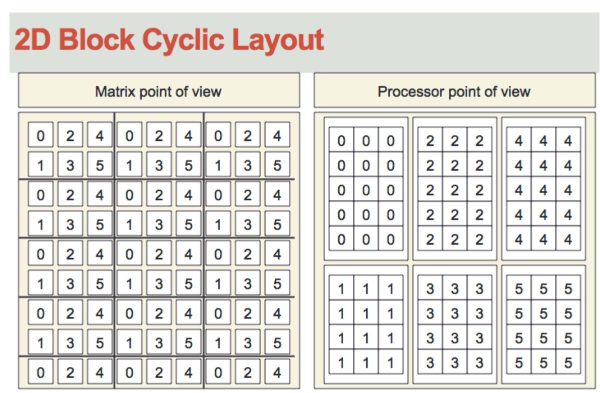


DLA Solvers

- We are interested in developing Dense Linear Algebra Solvers
- Retool LAPACK and ScaLAPACK for hybrid architectures

A New Generation of DLA Software

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing



MAGMA: LAPACK for GPUs

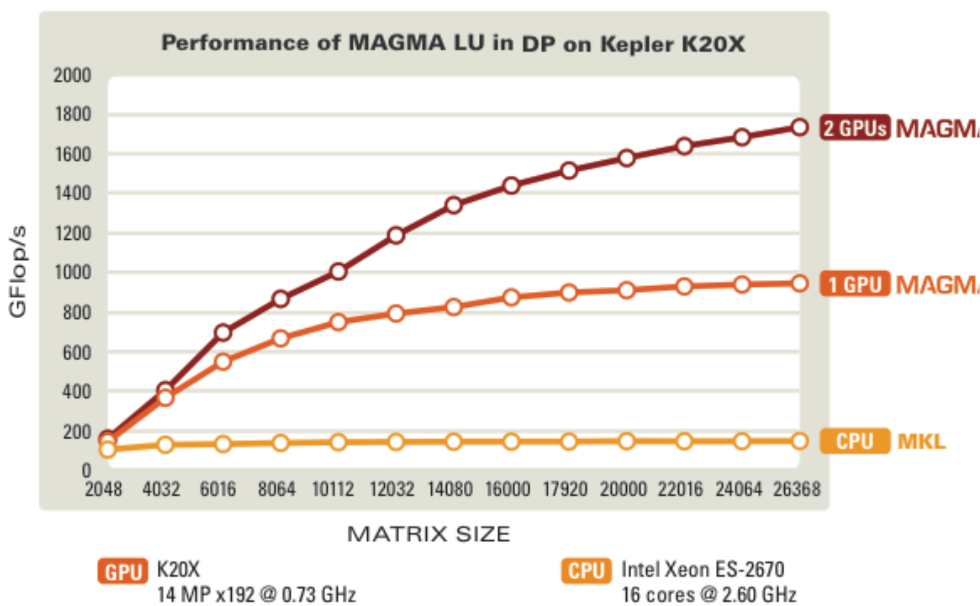
- **MAGMA**
  - Matrix algebra for GPU and multicore architecture
  - To provide LAPACK/ScaLAPACK on hybrid architectures
  - http://icl.cs.utk.edu/magma/
- **MAGMA for CUDA, Intel Xeon Phi, and OpenCL**
  - Hybrid dense linear algebra:
    - One-sided factorizations and linear system solvers
    - Two-sided factorizations and eigenproblem solvers
    - A subset of BLAS and auxiliary routines
- **MAGMA developers & collaborators**
  - UTK, UC Berkeley, UC Denver, INRIA (France), KAUST (Saudi Arabia)
  - Community effort, similar to LAPACK/ScaLAPACK

Key Aspects of MAGMA

HYBRID ALGORITHMS

MAGMA uses a hybridization methodology where algorithms of interest are split into tasks of varying granularity and their execution scheduled over the available hardware components. Scheduling can be static or dynamic. In either case, small non-parallelizable tasks, often on the critical path, are scheduled on the CPU, and larger more parallelizable ones, often Level 3 BLAS, are scheduled on the GPU.

PERFORMANCE

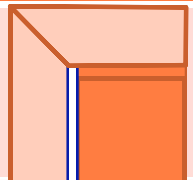
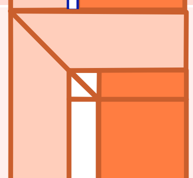
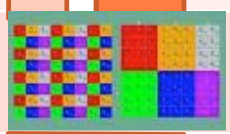
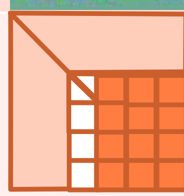
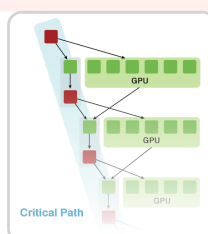


FEATURES AND SUPPORT

- **MAGMA 1.4** FOR **CUDA**
- **clMAGMA 1.0** FOR **OpenCL**
- **MAGMA MIC 1.0** FOR **Intel Xeon Phi**

CUDA	OpenCL	Intel Xeon Phi	
●	●	●	Linear system solvers
●	●	●	Eigenvalue problem solvers
●			MAGMA BLAS
●			CPU Interface
●	●	●	GPU Interface
●	●	●	Multiple precision support
●			Non-GPU-resident factorizations
●		●	Multicore and multi-GPU support
●			Tile factorizations with StarPU dynamic scheduling
●	●	●	LAPACK testing
●	●	●	Linux
●			Windows
●			Mac OS

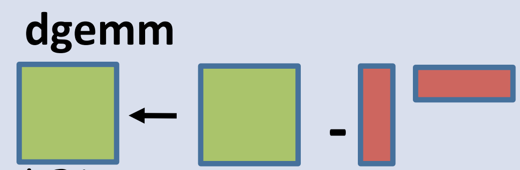
A New Generation of DLA Software

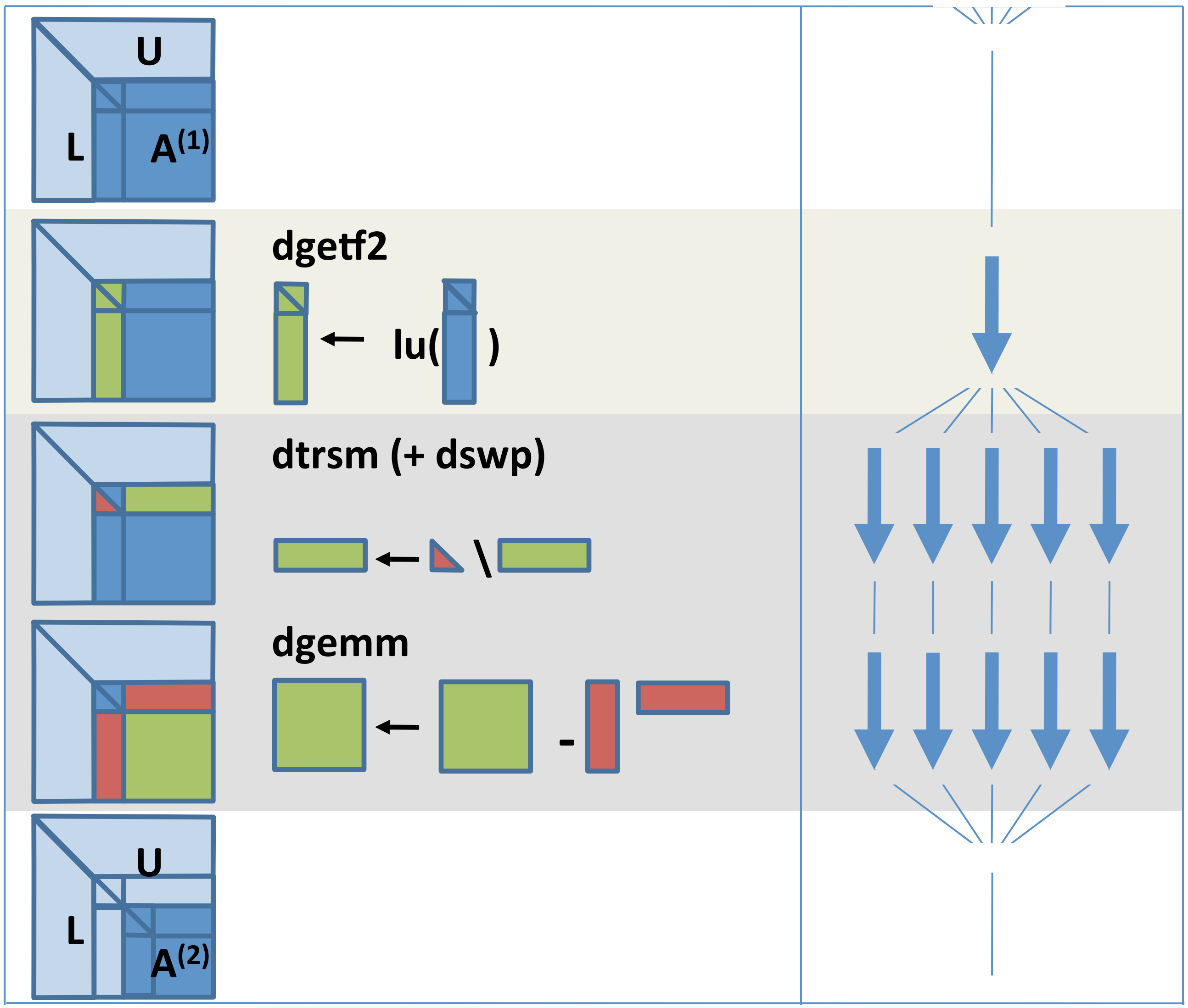
Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing
PLASMA (00's) New Algorithms (many-core friendly)		Rely on - a DAG/scheduler - block data layout - some extra kernels
MAGMA Hybrid Algorithms (heterogeneity friendly)		Rely on - hybrid scheduler - hybrid kernels

Parallelization of LU and QR.

Parallelize the update:

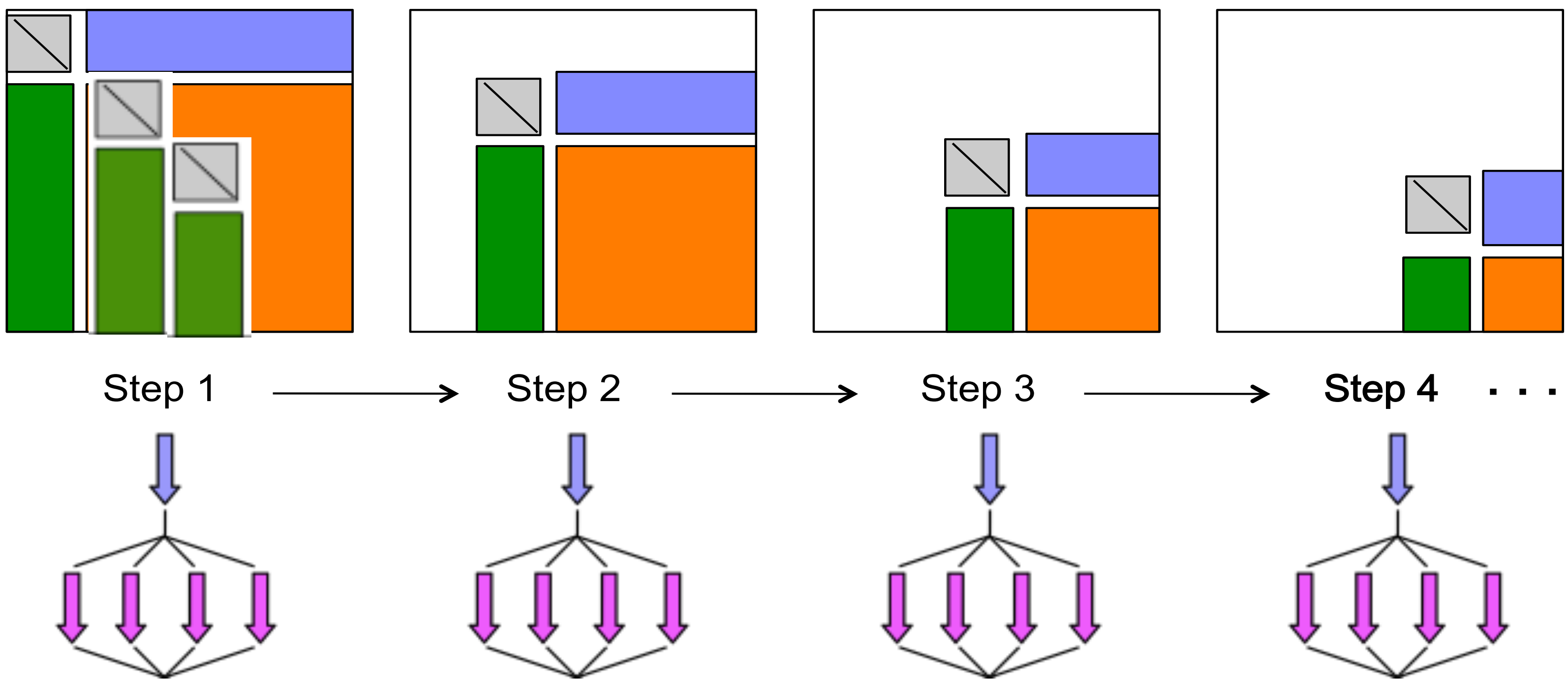
- Easy and done in any reasonable software.
- This is the $2/3n^3$ term in the FLOPs count.
- Can be done efficiently with LAPACK+multithreaded BLAS

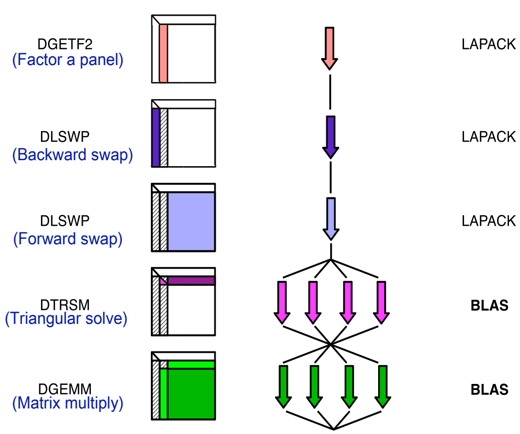




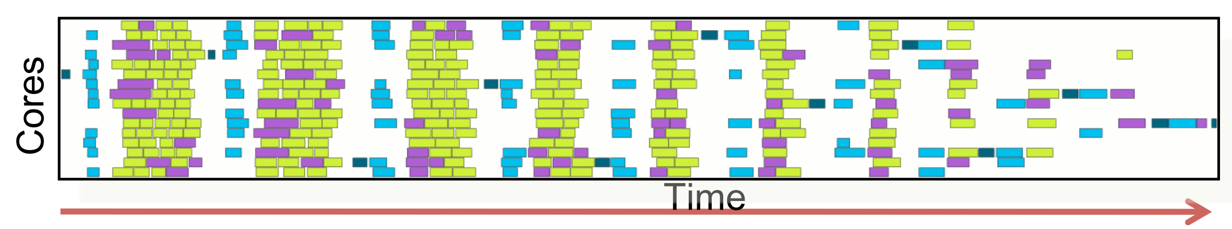
Fork - Join parallelism
Bulk Sync Processing

Synchronization (in LAPACK LU)





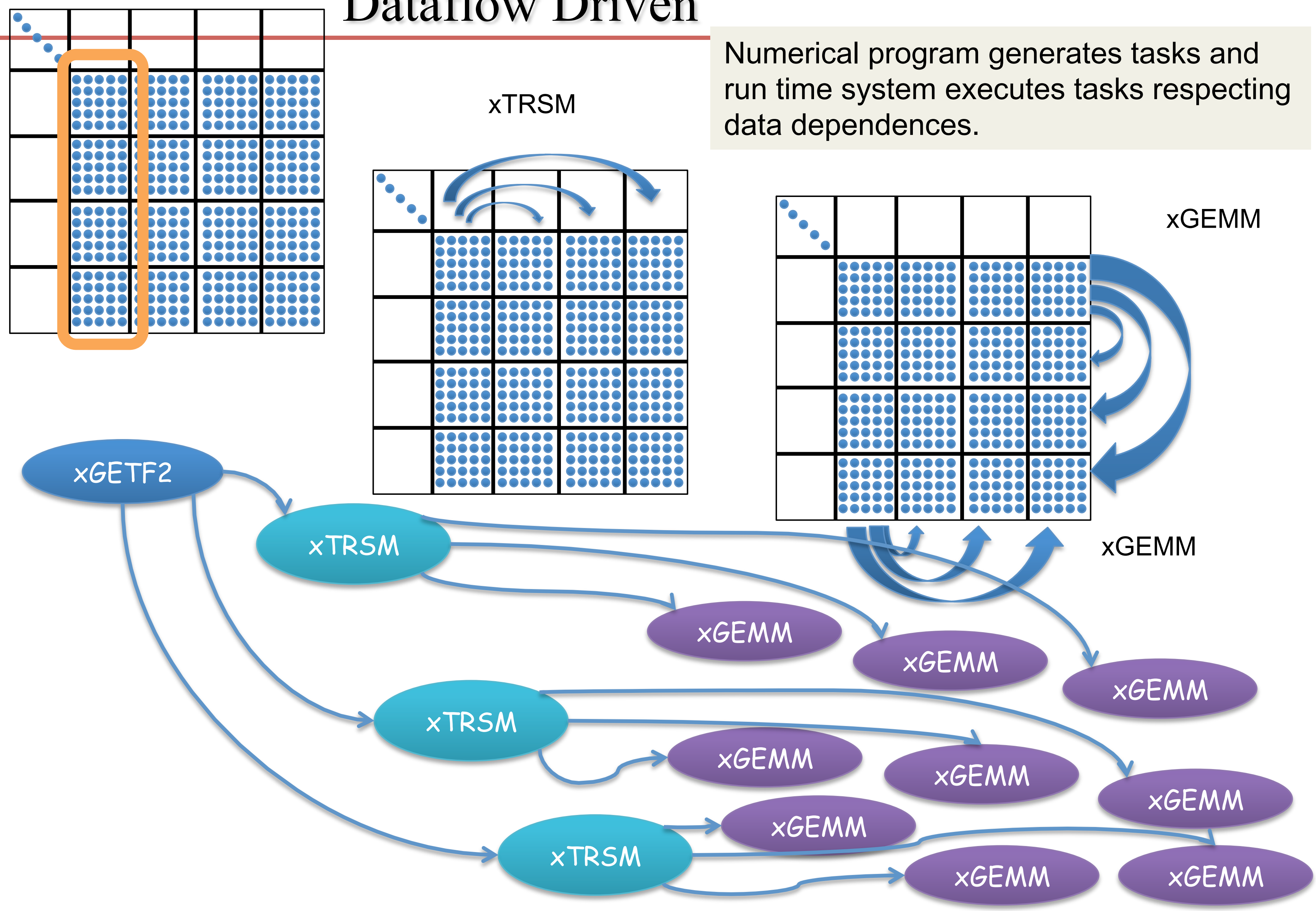
- fork join
- bulk synchronous processing



PLASMA LU Factorization

Dataflow Driven

Numerical program generates tasks and run time system executes tasks respecting data dependences.



High Performance Computing : Current Development

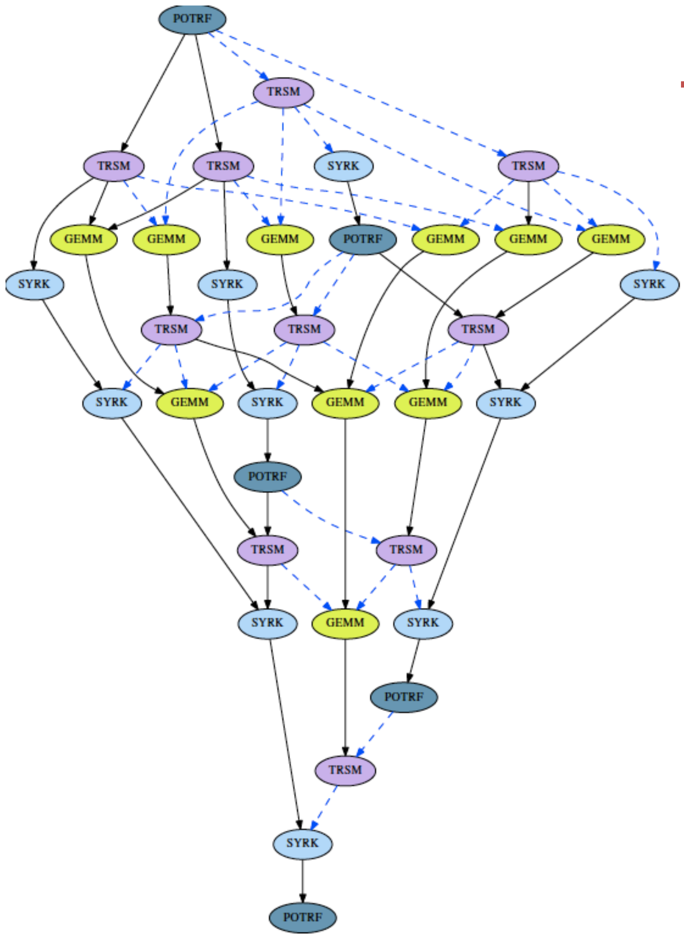
We are developing a strategy :

- That prioritizes the data-intensive operations to be executed by the accelerator

- That keep the memory-bound ones for the CPUs since the hierarchical caches with out-of-order superscalar scheduling are more appropriate to handle it.

- Moreover, in order to keep the accelerator busy, we redesign the kernels and propose dynamically guided data distribution to exploit enough parallelism to keep the accelerators and processors busy.

QUARK

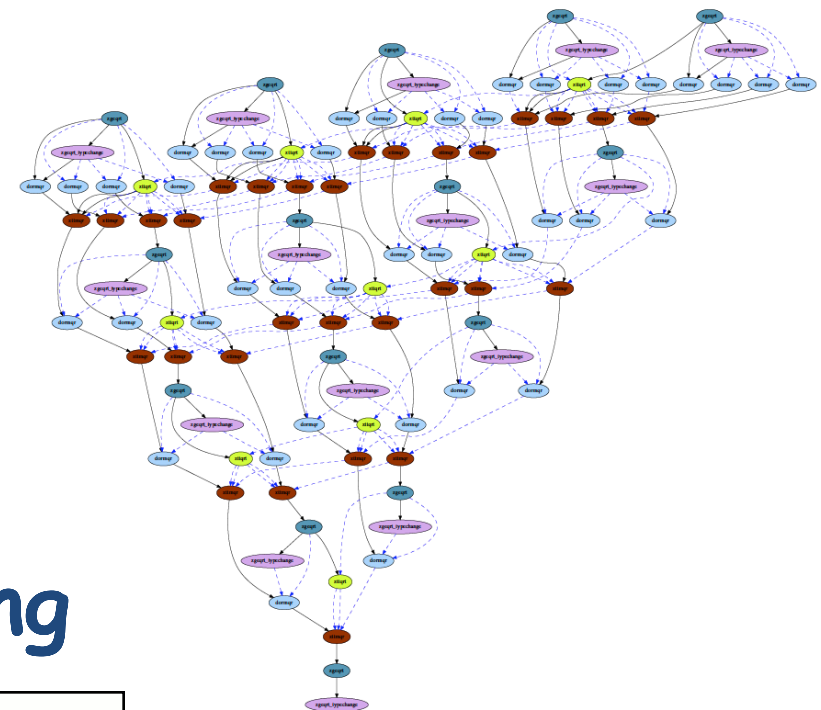


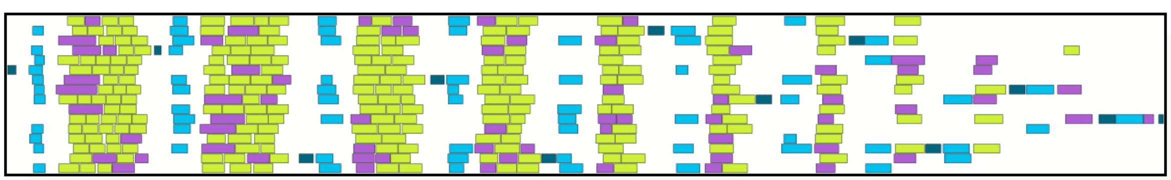
A runtime environment for the dynamic execution of precedence-constraint tasks (DAGs) in a multicore machine

- Translation
- If you have a serial program that consists of computational kernels (tasks) that are related by data dependencies, QUARK can help you execute that program (relatively efficiently and easily) in parallel on a multicore machine

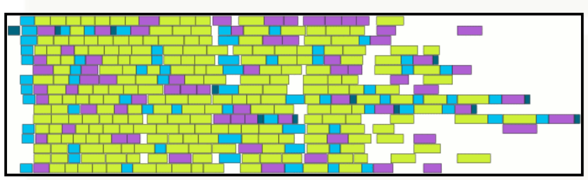
The Purpose of a QUARK Runtime

- **Objectives**
  - High utilization of each core
  - Scaling to large number of cores
  - Synchronization reducing algorithms
- **Methodology**
  - Dynamic DAG scheduling (QUARK)
  - Explicit parallelism
  - Implicit communication
  - Fine granularity / block data layout
- **Arbitrary DAG with dynamic scheduling**





Fork-join parallelism
Notice the synchronization penalty in the presence of heterogeneity.



DAG scheduled parallelism

QUARK

Shared Memory Superscalar Scheduling

```
FOR k = 0..TILES-1
  A[k][k] ← DPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    A[m][k] ← DTRSM(A[k][k], A[m][k])
  FOR m = k+1..TILES-1
    A[m][m] ← DSYRK(A[m][k], A[m][m])
    FOR n = k+1..m-1
      A[m][n] ← DGEMM(A[m][k], A[n][k], A[m][n])
```

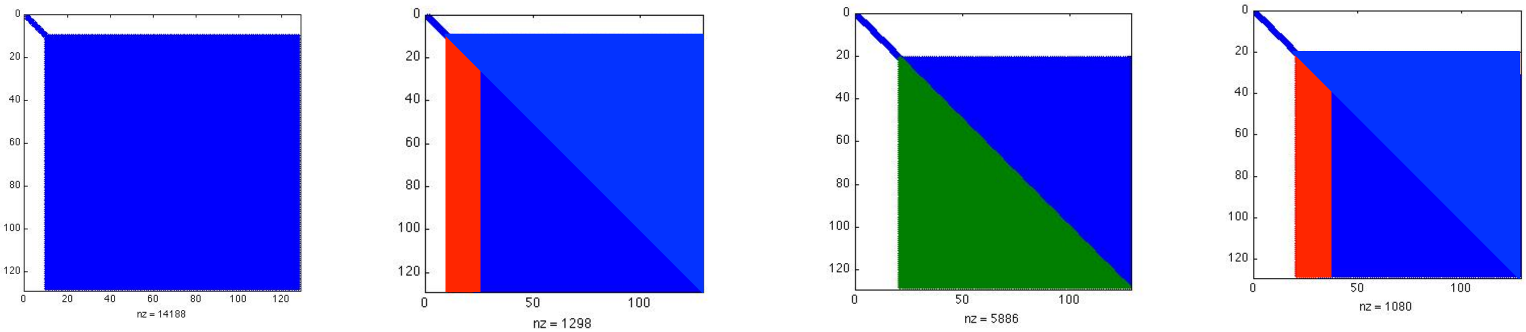
definition – pseudocode

```
for (k = 0; k < A.mt; k++) {
    QUARK_CORE_dpotrf(...);
    for (m = k+1; m < A.mt; m++) {
        QUARK_CORE_dtrsm(...);
    }
    for (m = k+1; m < A.mt; m++) {
        QUARK_CORE_dsyrk(...);
        for (n = k+1; n < m; n++) {
            QUARK_CORE_dgemm(...)
        }
    }
}
```

implementation – actual QUARK code in PLASMA

High Performance Computing : current development

1. Standard hybrid CPU-GPU implementation



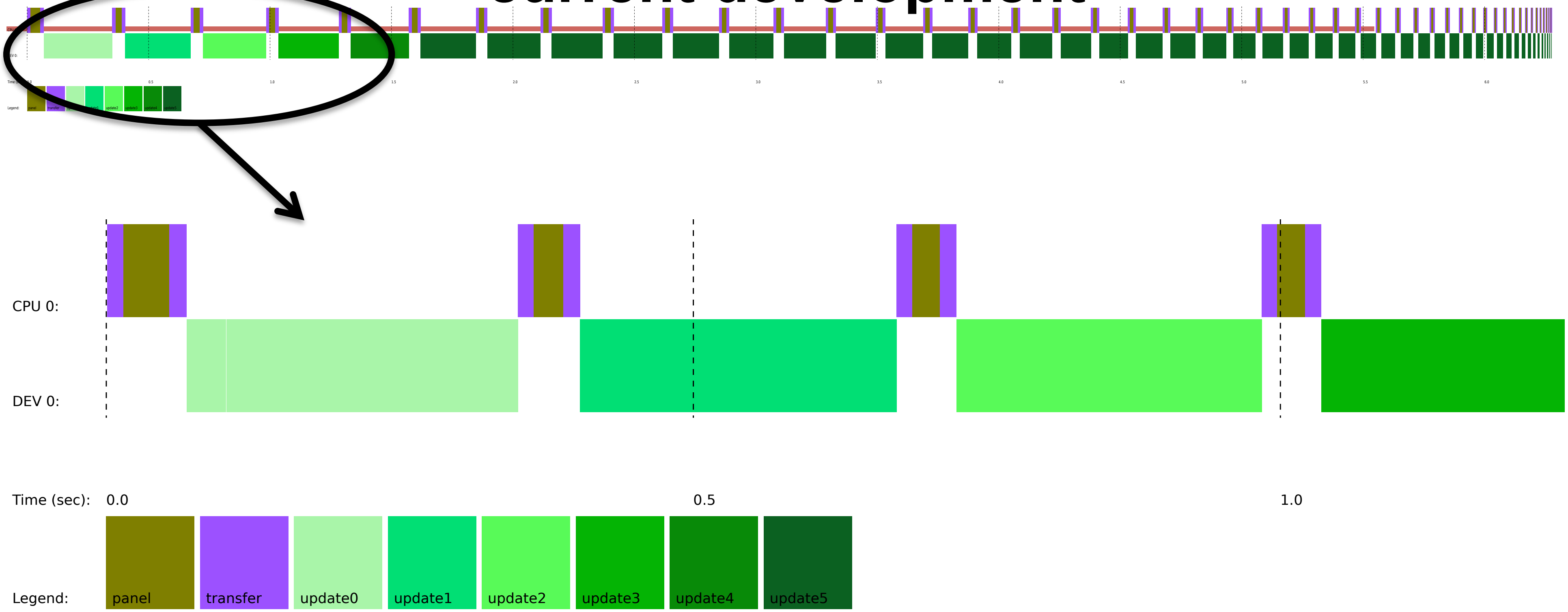
factor panel k then update ➔ factor panel k+1

Algorithm 1: Two-phase implementation of a one-sided factorization.

---

for $P_i \in \{P_1, P_2, \ldots, P_n\}$ **do**
- **CPU:**
- Receive Panel(P_i)
- PanelFactorize(P_i)
- Send Panel(P_i)
- **GPU:**
- TrailingMatrixUpdate($A^{(i)}$)

High Performance Computing : current development



Standard implementation without lookahead:

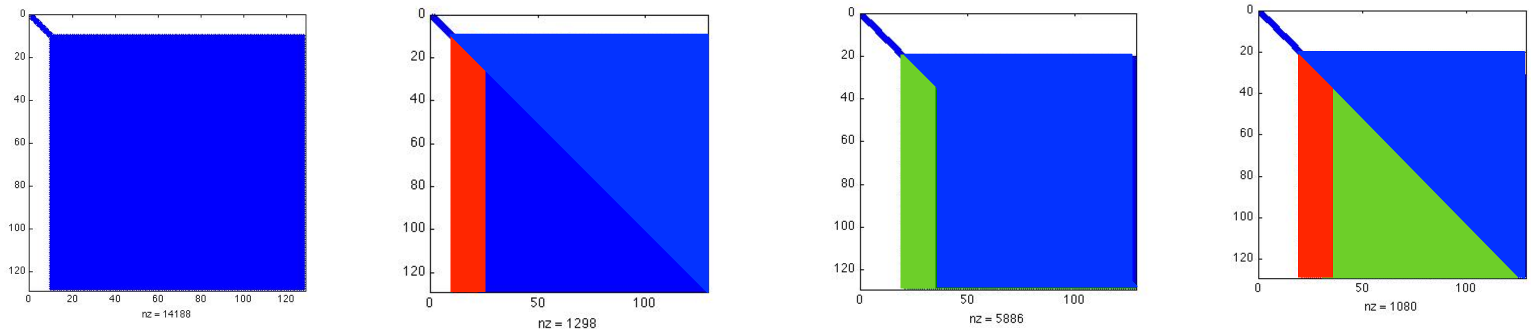
➢Execution trace of the Cholesky factorization on a single socket CPU (Sandy Bridge) and a K20c GPU.

➢We see that the computation on the CPU (e.g., the panel factorization) is not overlapped with the computation on the GPU.

➢The algorithm looks like sequential, the only advantage is that the data extensive operations are accelerated by the GPU.

High Performance Computing: current development

2. Introducing a lookahead panel to overlap CPU and GPU



factor panel k   then update ➔ factor panel k+1

next panel

continue update k

Algorithm 2: Two-phase implementation with a split update and explicit communication.

---

for $P_i \in \{P_1, P_2, \ldots, P_n\}$ **do**

  CPU:

  Receive Panel(P_i)

  PanelFactorize(P_i)

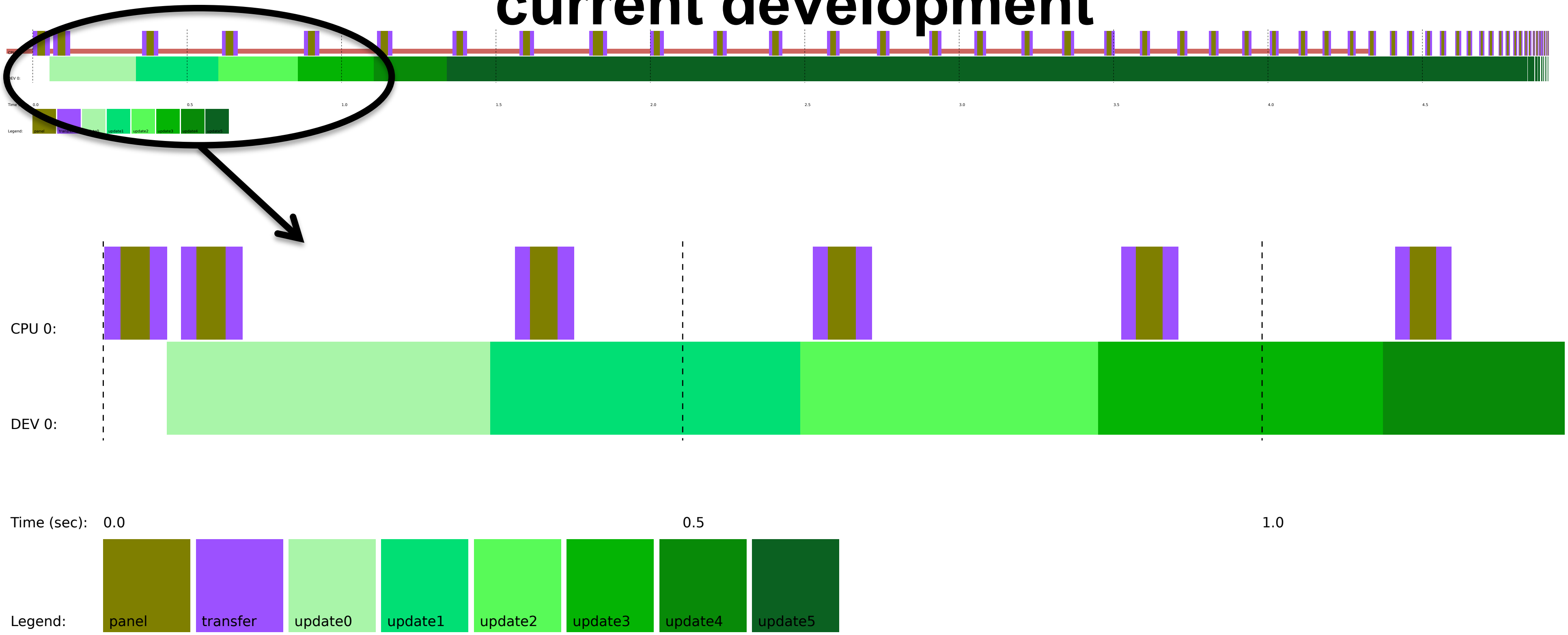
  Send Panel(P_i)

  GPU:

  NextPanelUpdate(lookahead $P_{(i+1)}$) $\rightarrow$ goto CPU

  TrailingMatrixUpdate($A^{(i)}$)

High Performance Computing: current development



New implementation with lookahead:

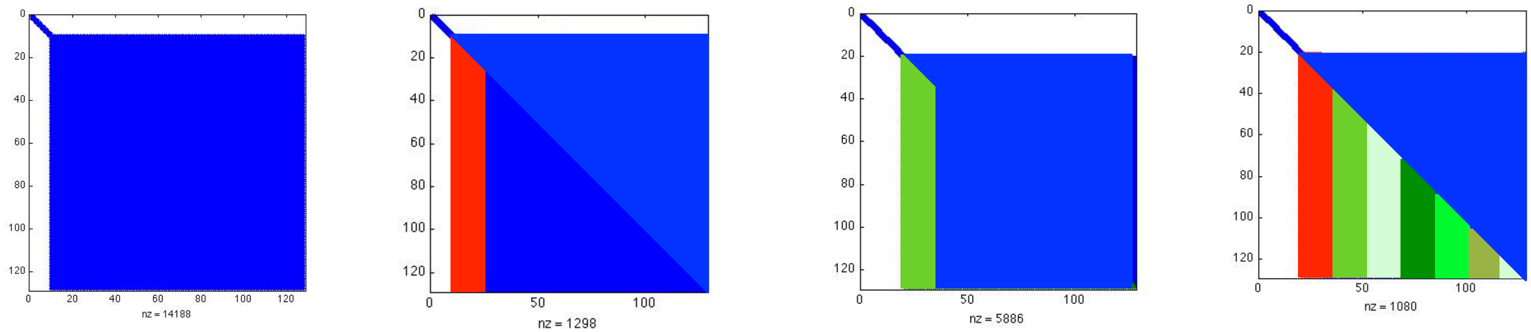
➢Execution trace of the Cholesky factorization on a single socket CPU (Sandy Bridge) and a K20c GPU.

➢We see that the memory-bound kernel (e.g., the panel factorization) has been allocated to the CPU while the compute-bound kernel (e.g., the update performed by DSYRK) has been allocated to the accelerator.

➢the advantage of such strategy is not only to hide the data transfer cost between the CPU and GPU but also to keep the GPU busy all the way until the end of execution.

High Performance Computing : current development

3. Prioritizing critical path to provide more parallelism if needed



factor panel k then update → factor panel k+1

next panel

continue update k

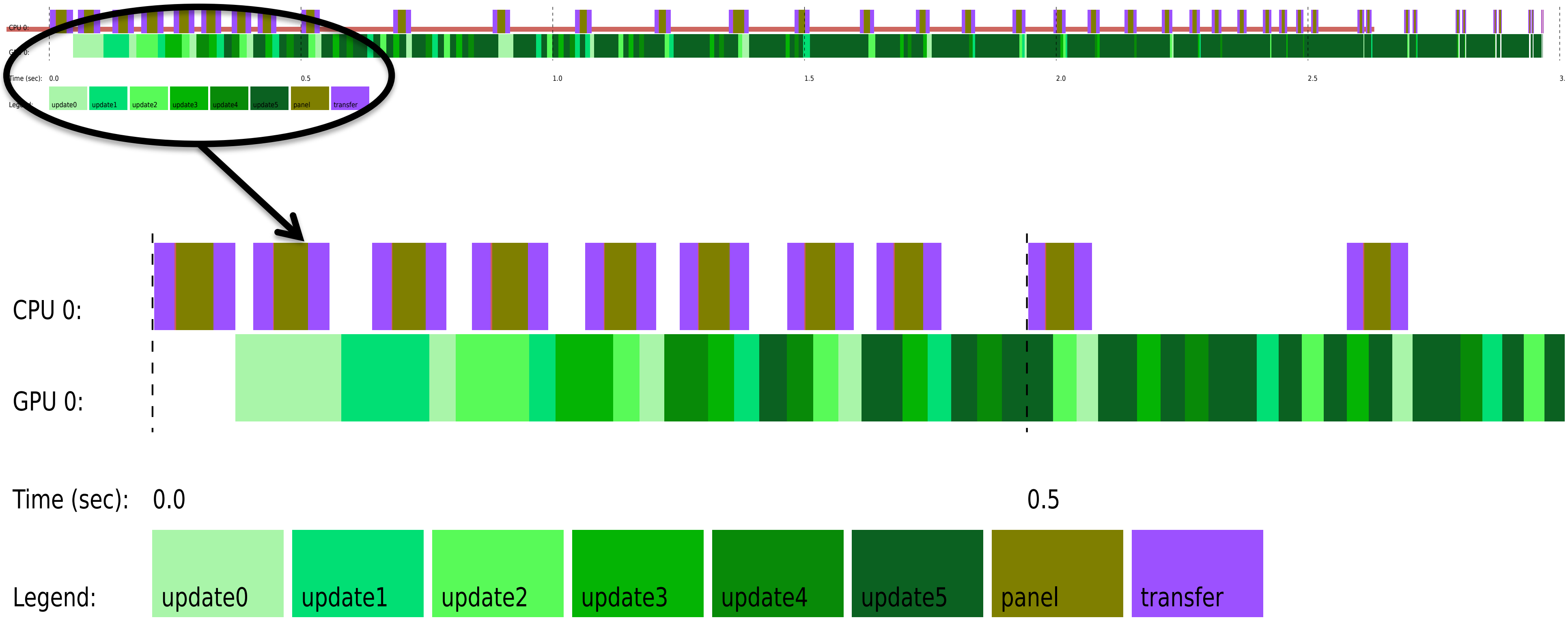
Algorithm 3: Two-phase implementation with a split update prioritizing critical path.

---

for $P_i \in \{P_1, P_2, \ldots, P_n\}$ **do**

- **CPU:**
- Receive Panel(P_i)
- PanelFactorize(P_i)
- Send Panel(P_i)
- **GPU:**
- **for** $j \in \{P_{i+1}, P_{i+2}, \ldots, P_n\}$ **do**
  - MatrixUpdate of block j($P_{(j)}$) with priority $p - j$

High Performance Computing : current development

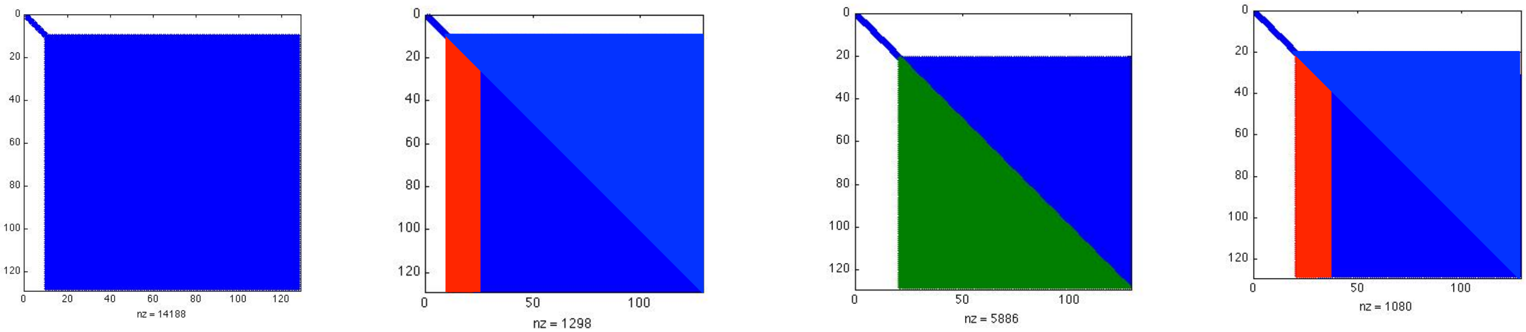


Prioritize the critical path :

- ➢the panel factorization can be executed earlier. This will increase the lookahead depth that the algorithm exposes, increasing parallelism, so that there are more update tasks available to be executed by the device resources.
- ➢This options has advantage when a lot of parallelism is needed especially for small sizes.

High Performance Computing: current development

1. Standard hybrid CPU-GPU implementation



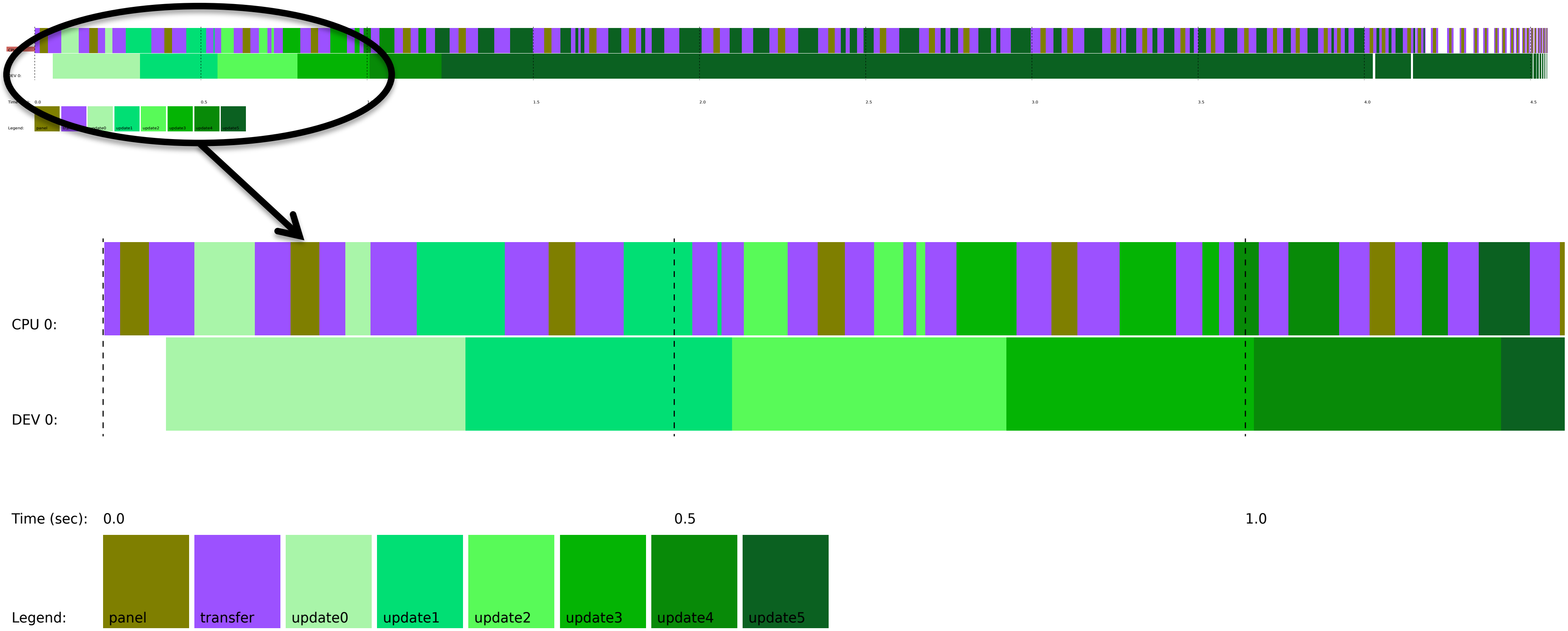
factor panel k then update ➔ factor panel k+1

Algorithm 1: Two-phase implementation of a one-sided factorization.

for $P_i \in \{P_1, P_2, \ldots, P_n\}$ **do**

- **CPU:**
- Receive Panel(P_i)
- PanelFactorize(P_i)
- Send Panel(P_i)
- **GPU:**
- TrailingMatrixUpdate($A^{(i)}$)

High Performance Computing : current development

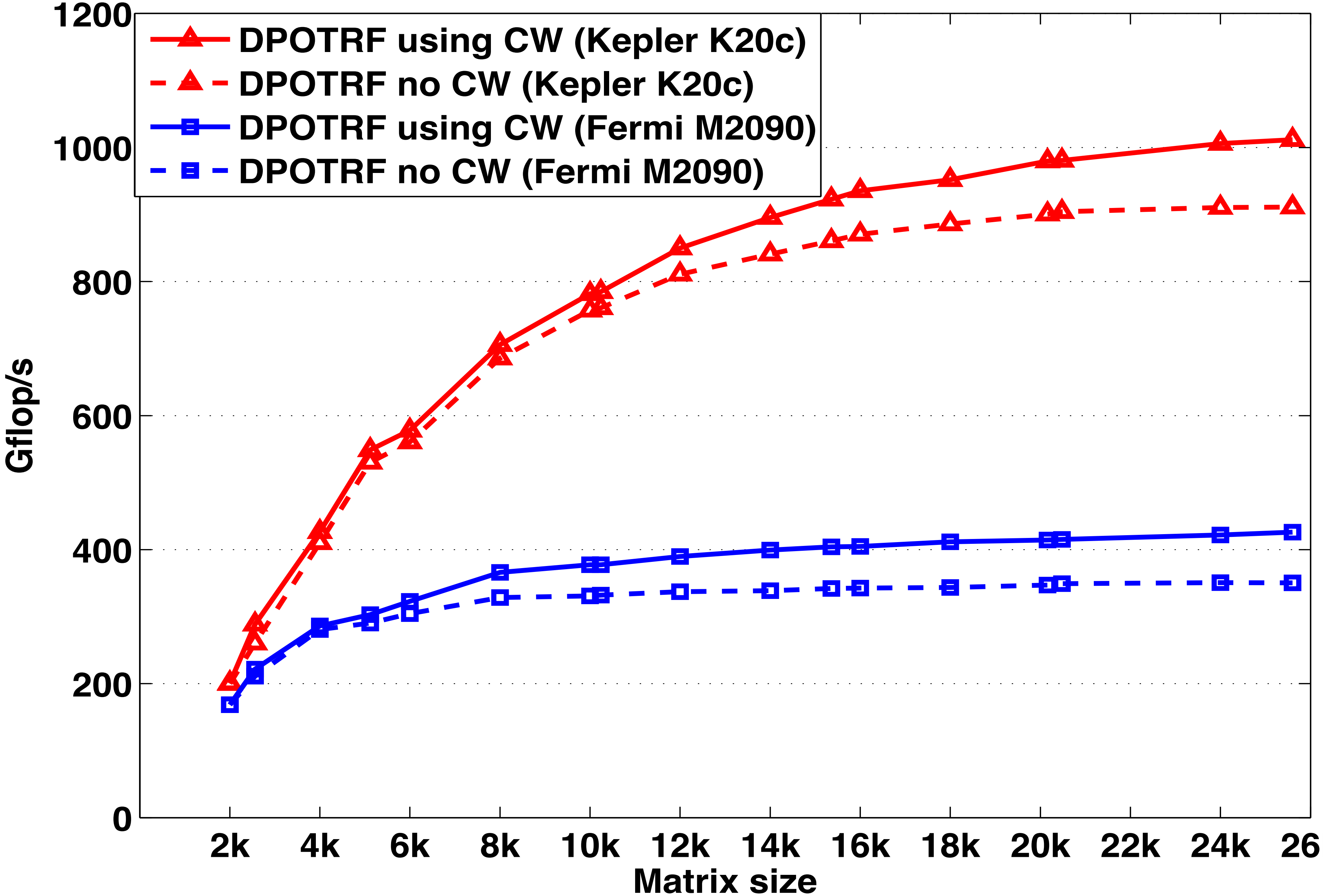


Resource Capability Weight :

➢ the advantage of such strategy is to keep all resources busy all the way until the end of execution.

➢ Careful management of the capability-weights ensures that the CPU does not take any work that would cause a delay to the GPU, since that would negatively affect the performance.

High Performance Computing : current development

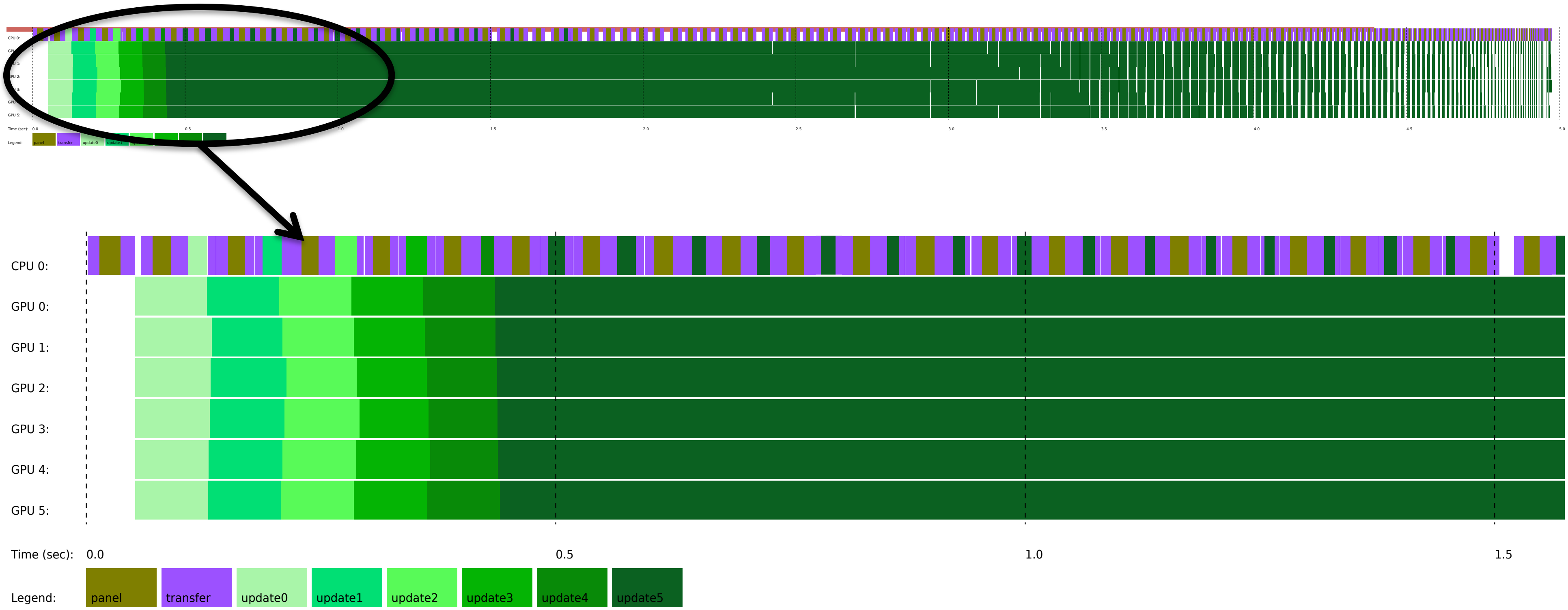


High Performance Computing : current development

- **Multiple GPU Case**
  - **Experiments with 6 GPUs**

Scalability and performance of such implementation

High Performance Computing : current development

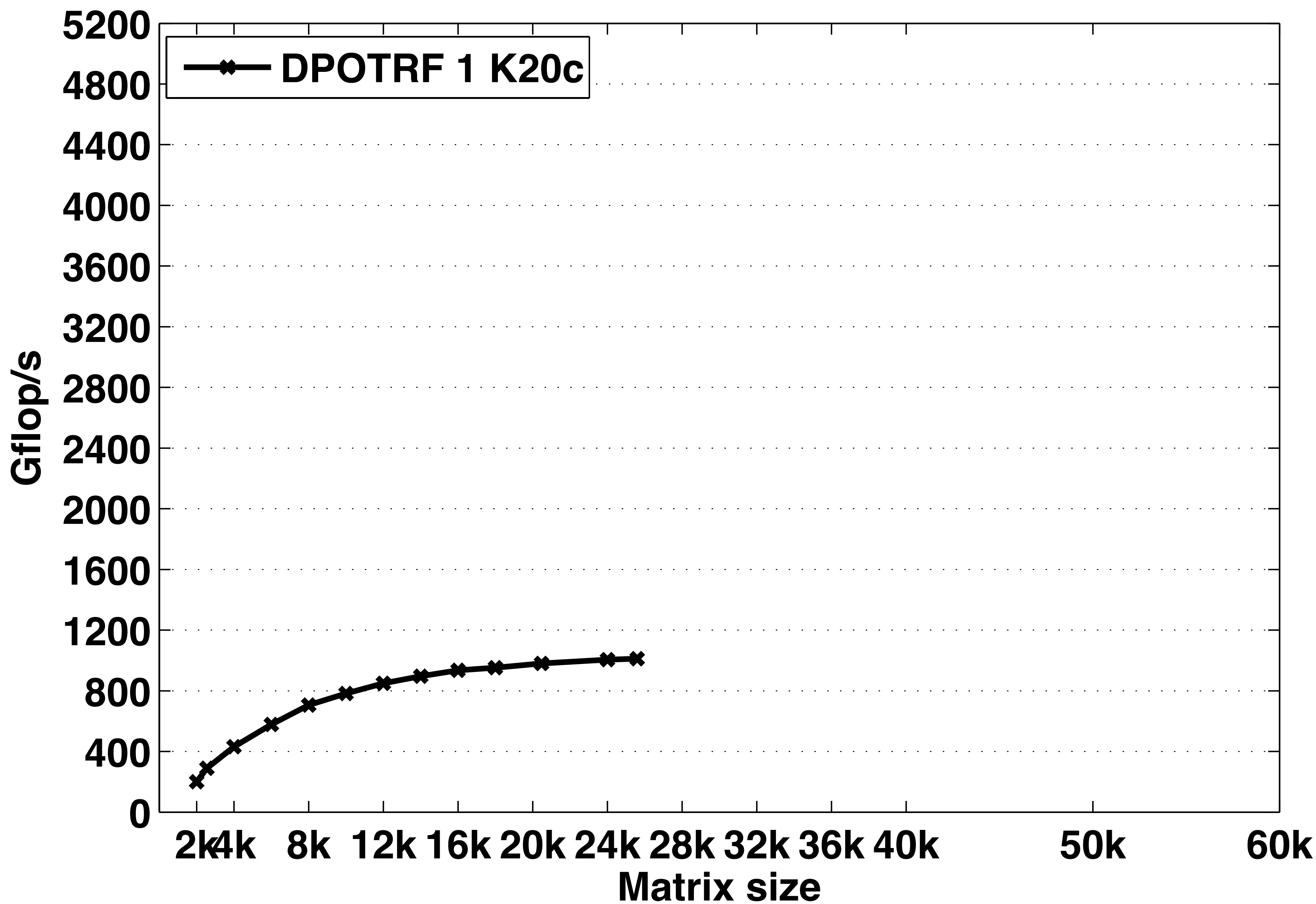


Scalability and efficiency :

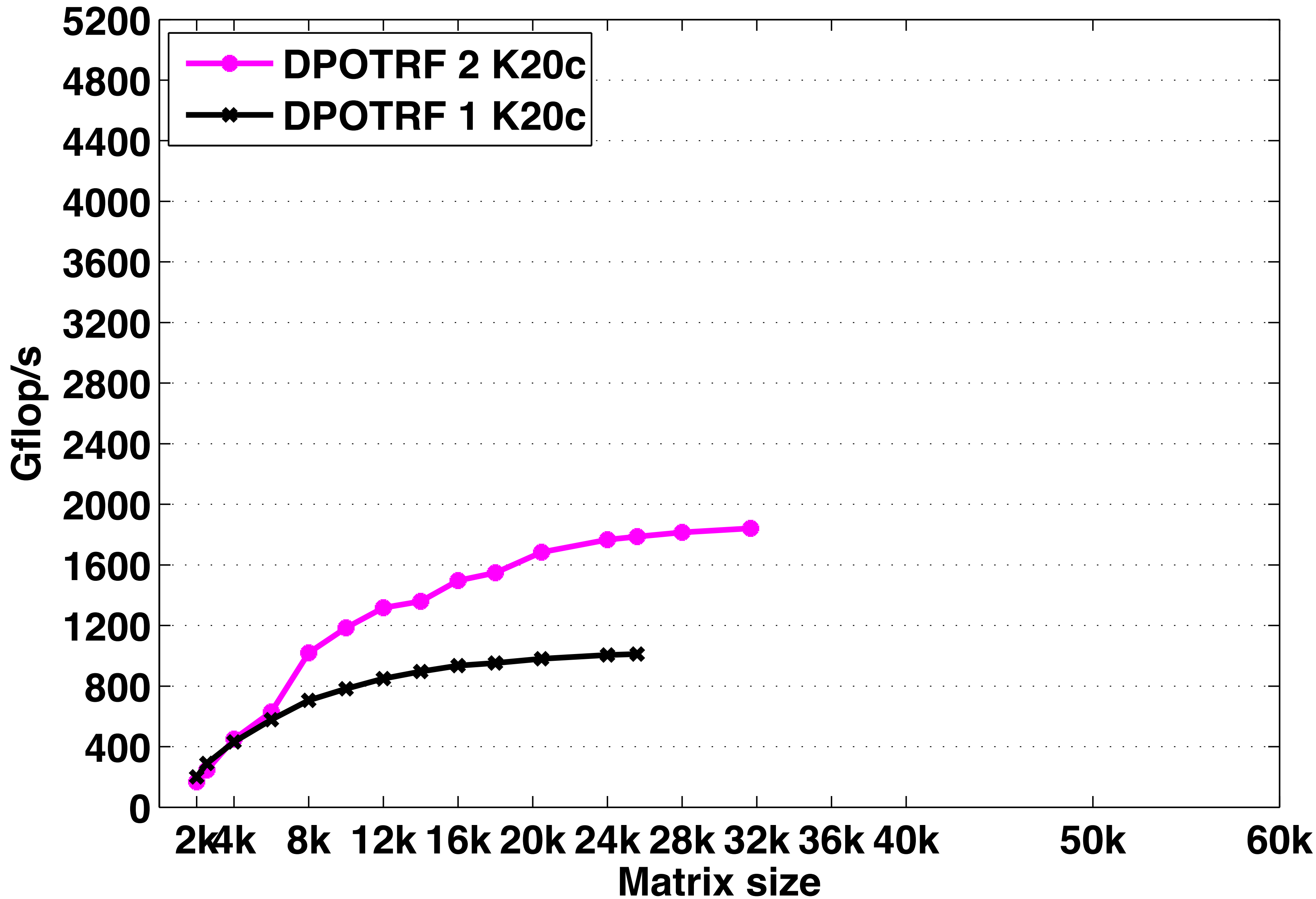
➢snapshot of the execution trace of the Cholesky factorization on System A for a matrix of size 40K using six GPUs K20c.

➢As expected the pattern of the trace looks compressed which means that our implementation is able to schedule and balance the tasks on the whole six GPUs devices.

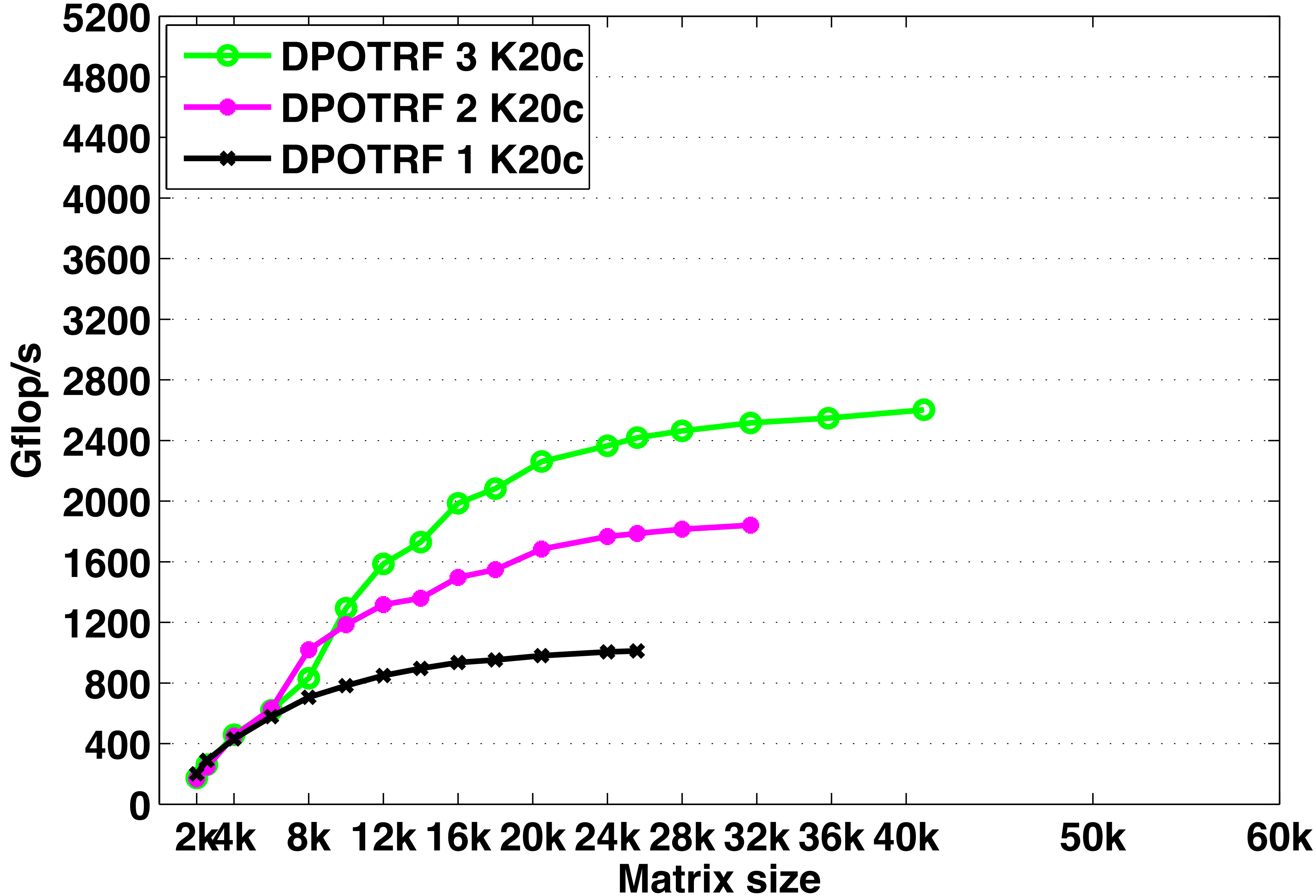
magma_quark DPOTRF Kepler K20c



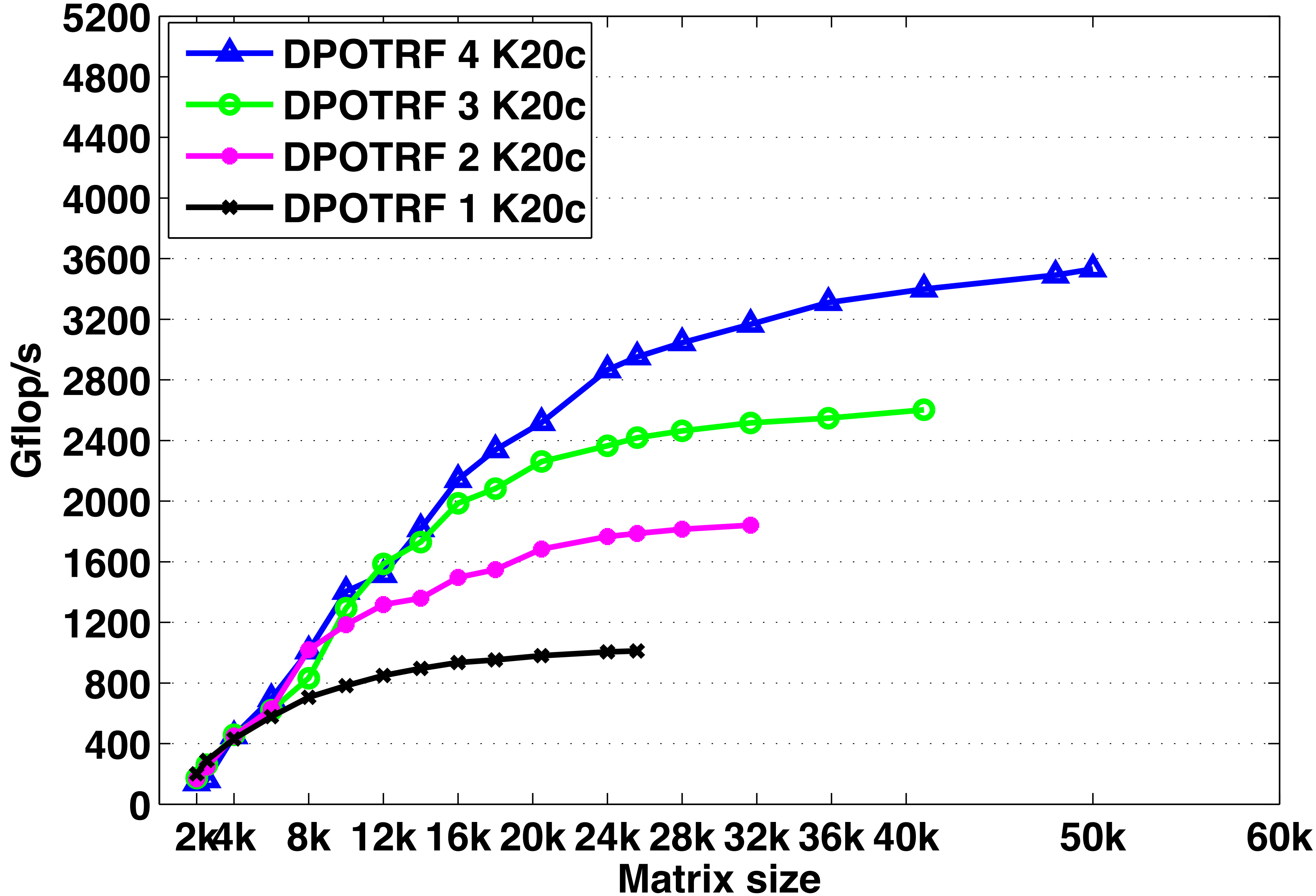
magma_quark DPOTRF Kepler K20c



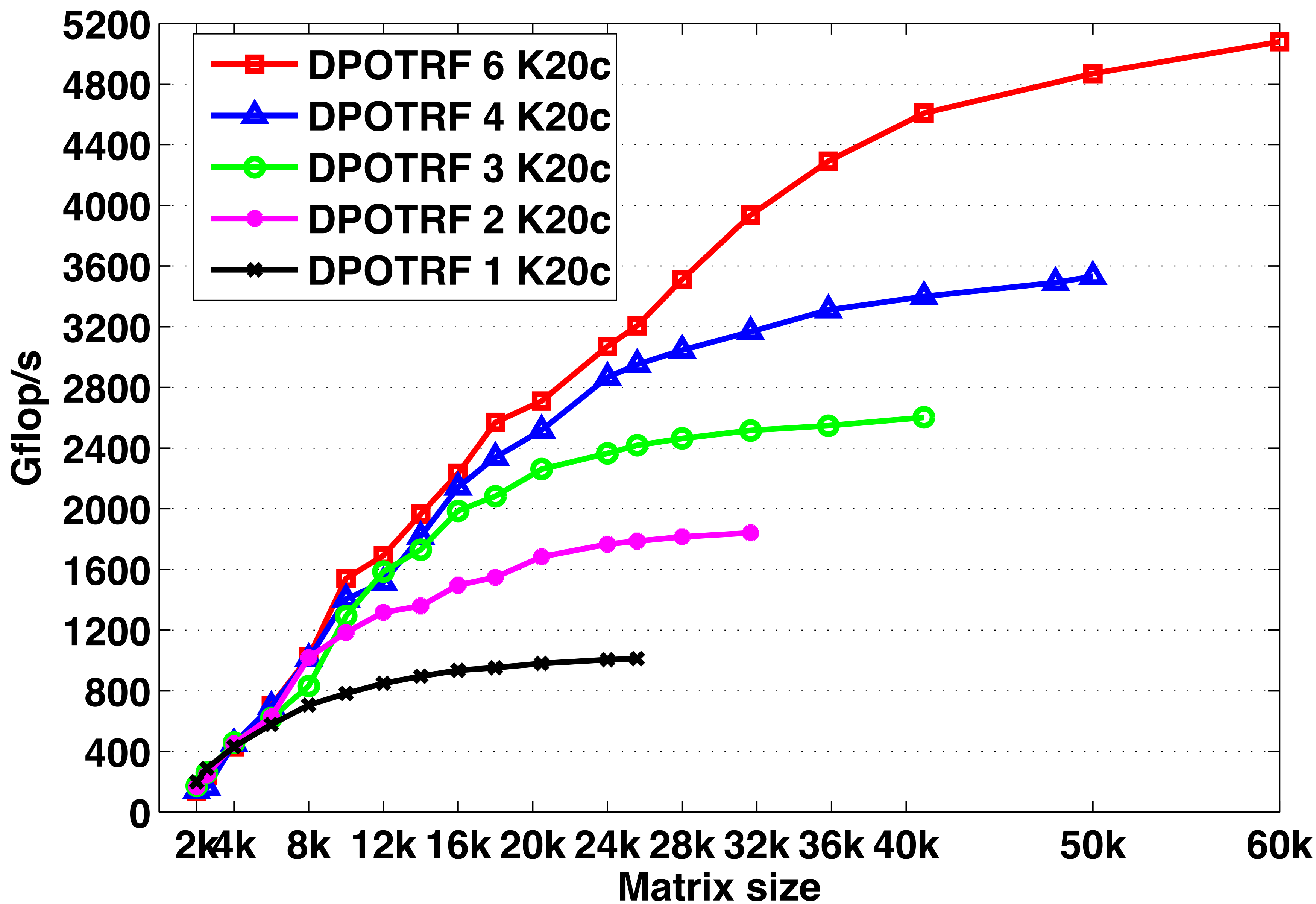
magma_quark DPOTRF Kepler K20c



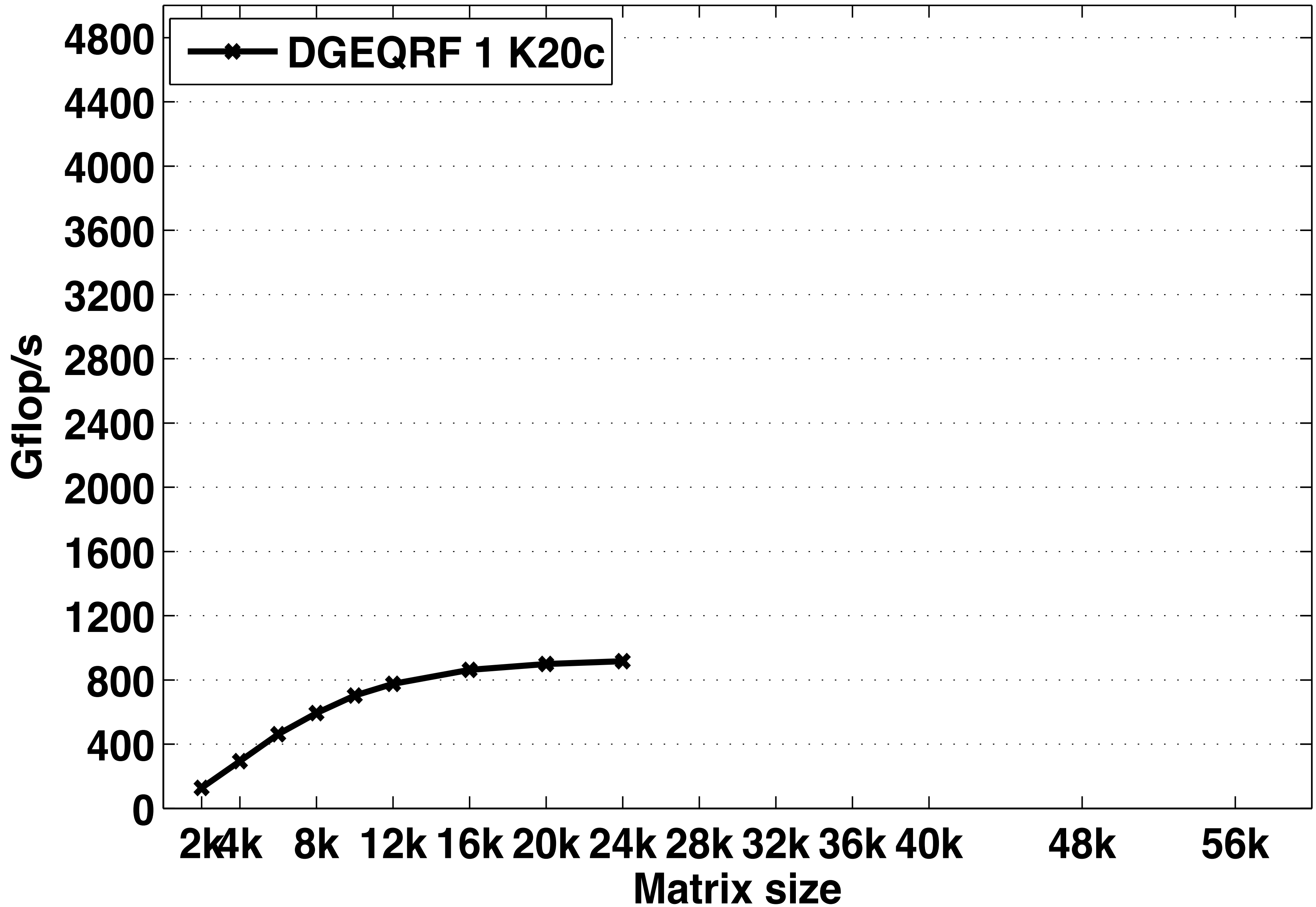
magma_quark DPOTRF Kepler K20c



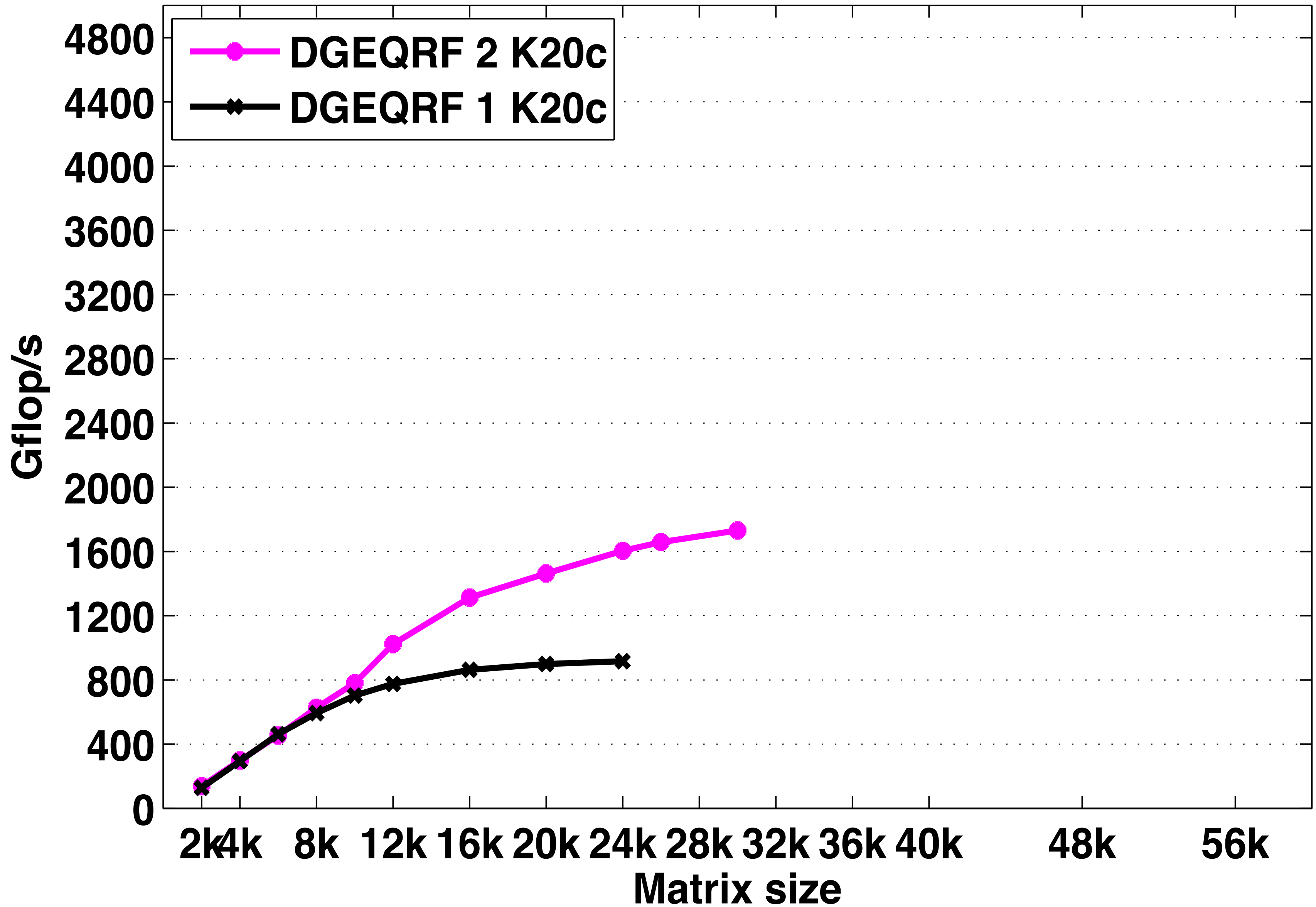
magma_quark DPOTRF Kepler K20c



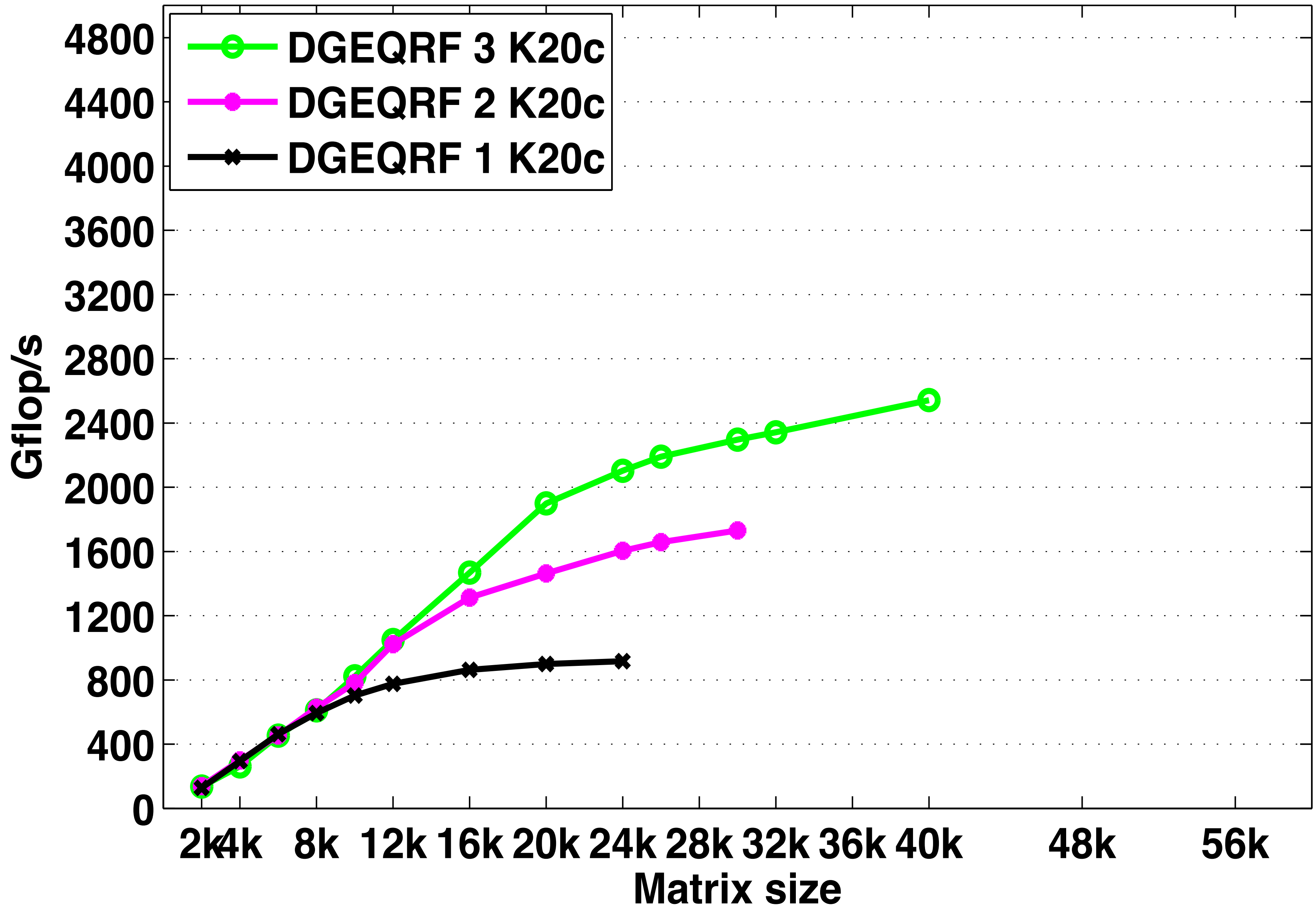
magma_quark DGEQRF Kepler K20c



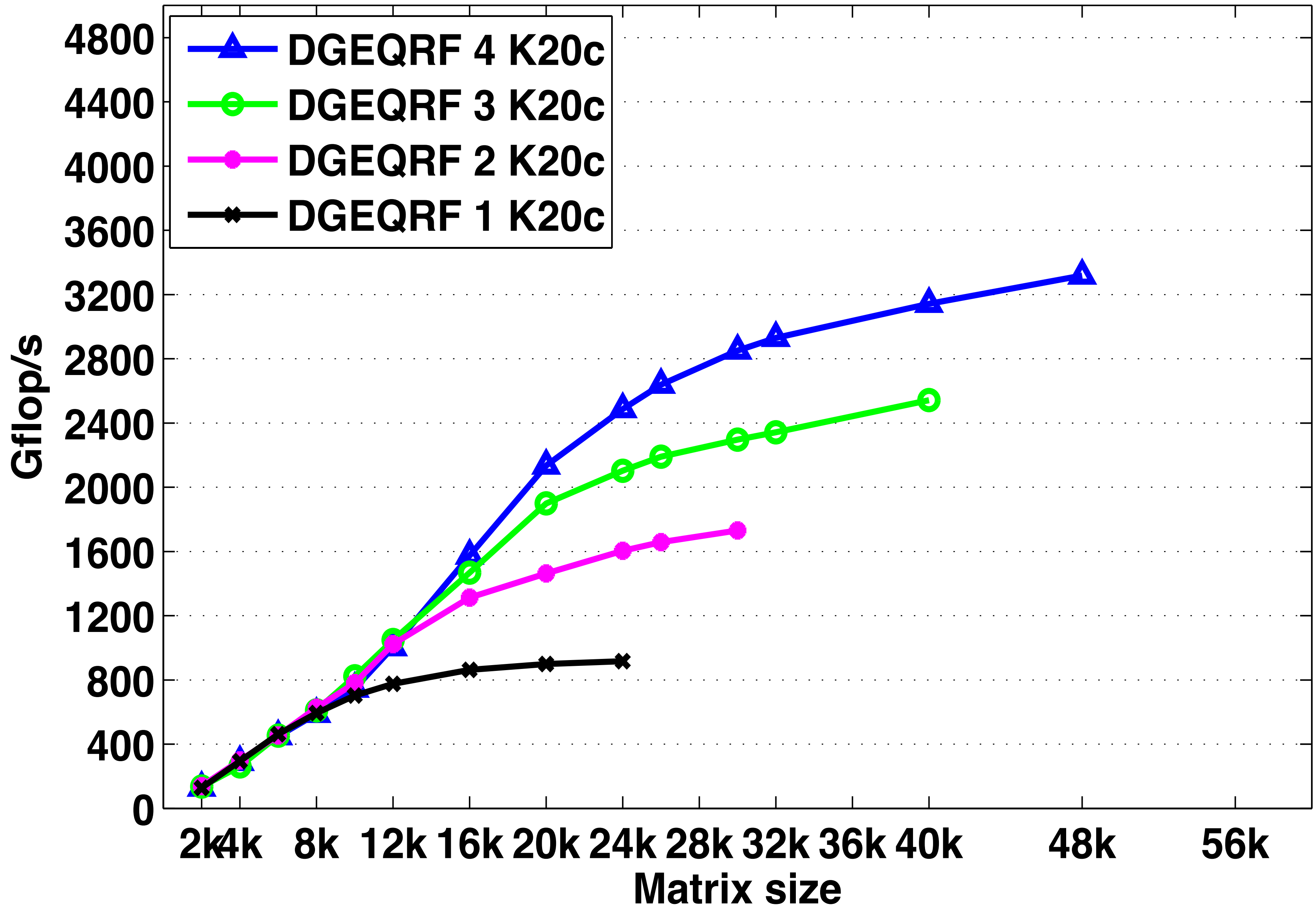
magma_quark DGEQRF Kepler K20c



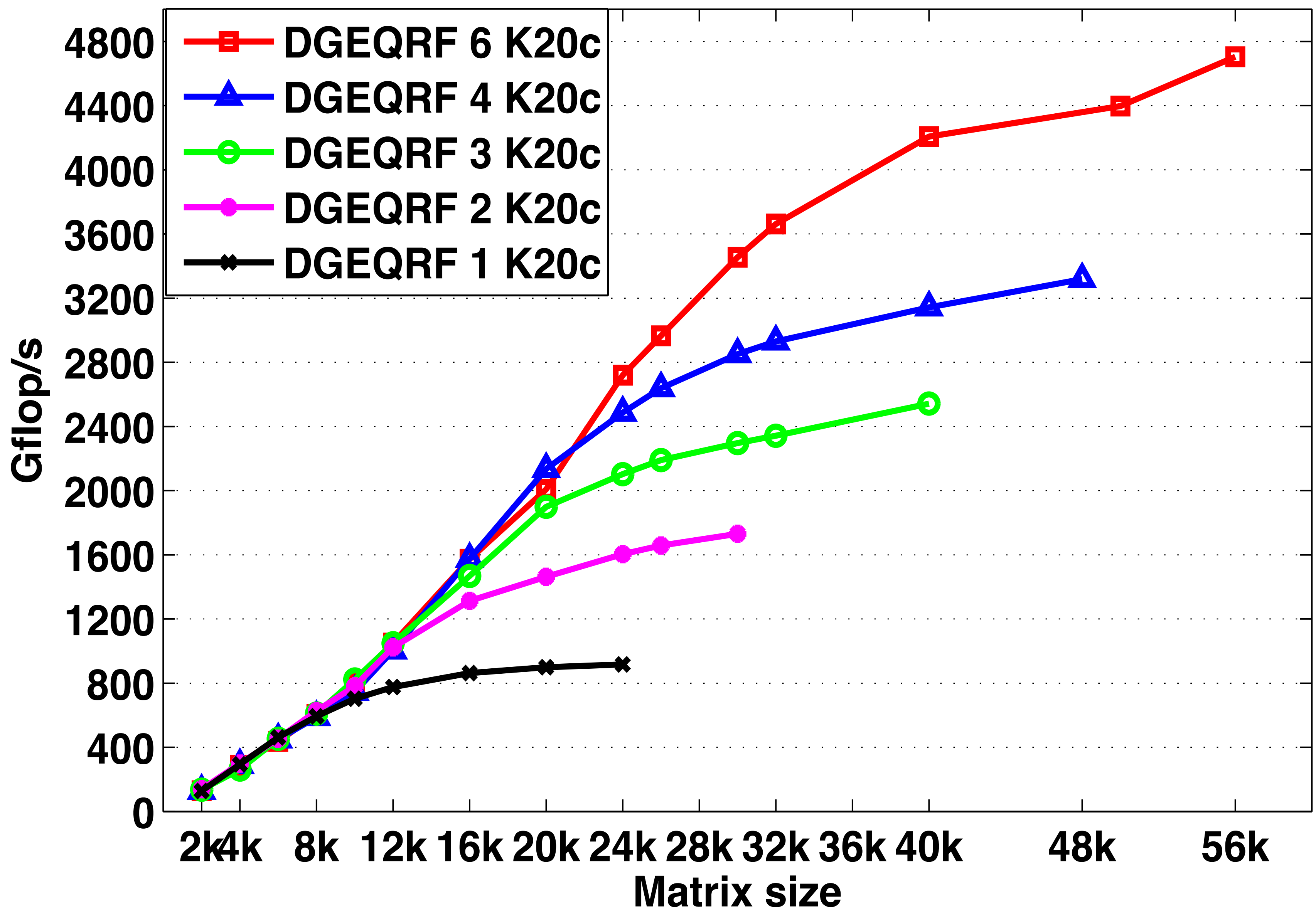
magma_quark DGEQRF Kepler K20c



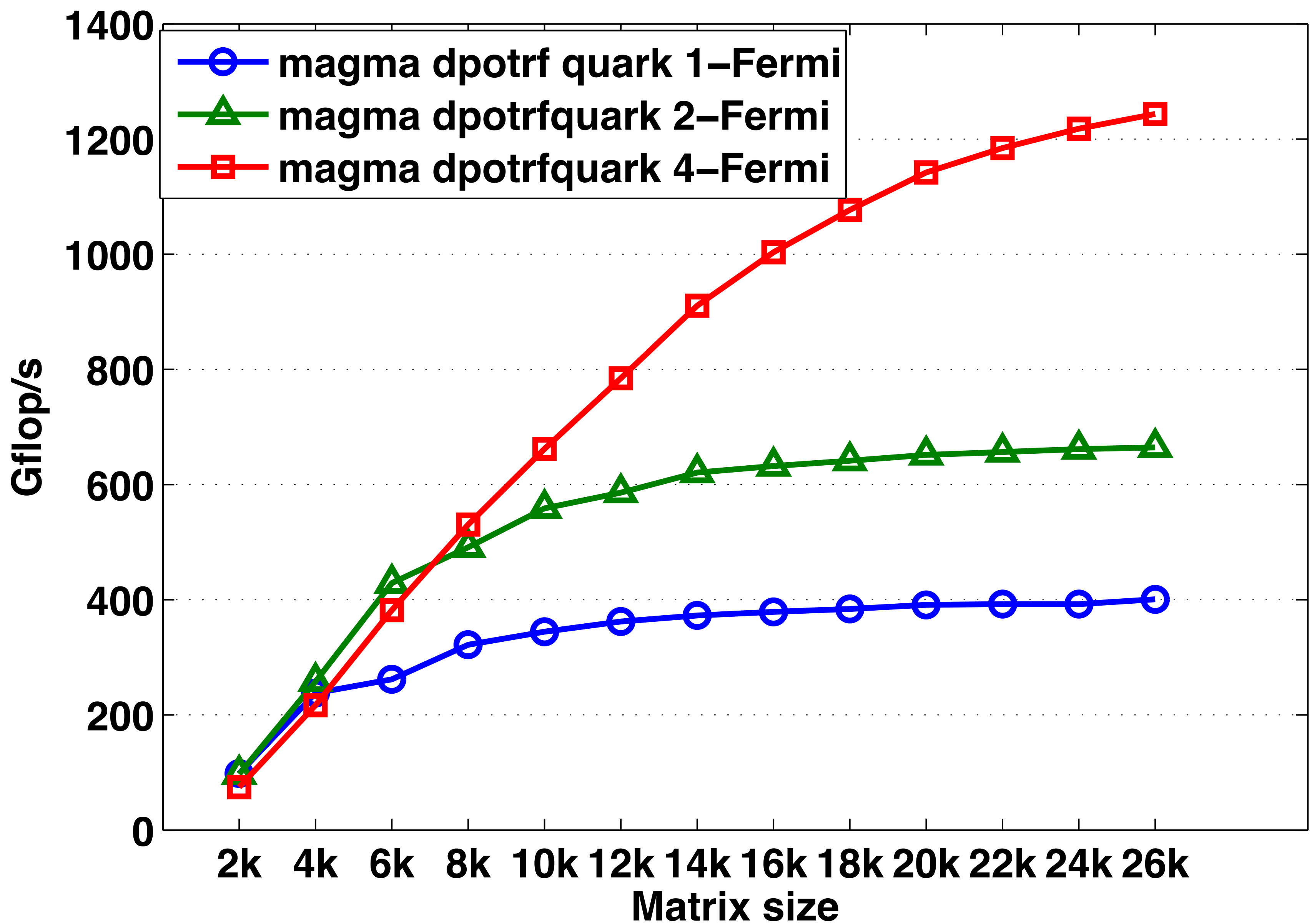
magma_quark DGEQRF Kepler K20c



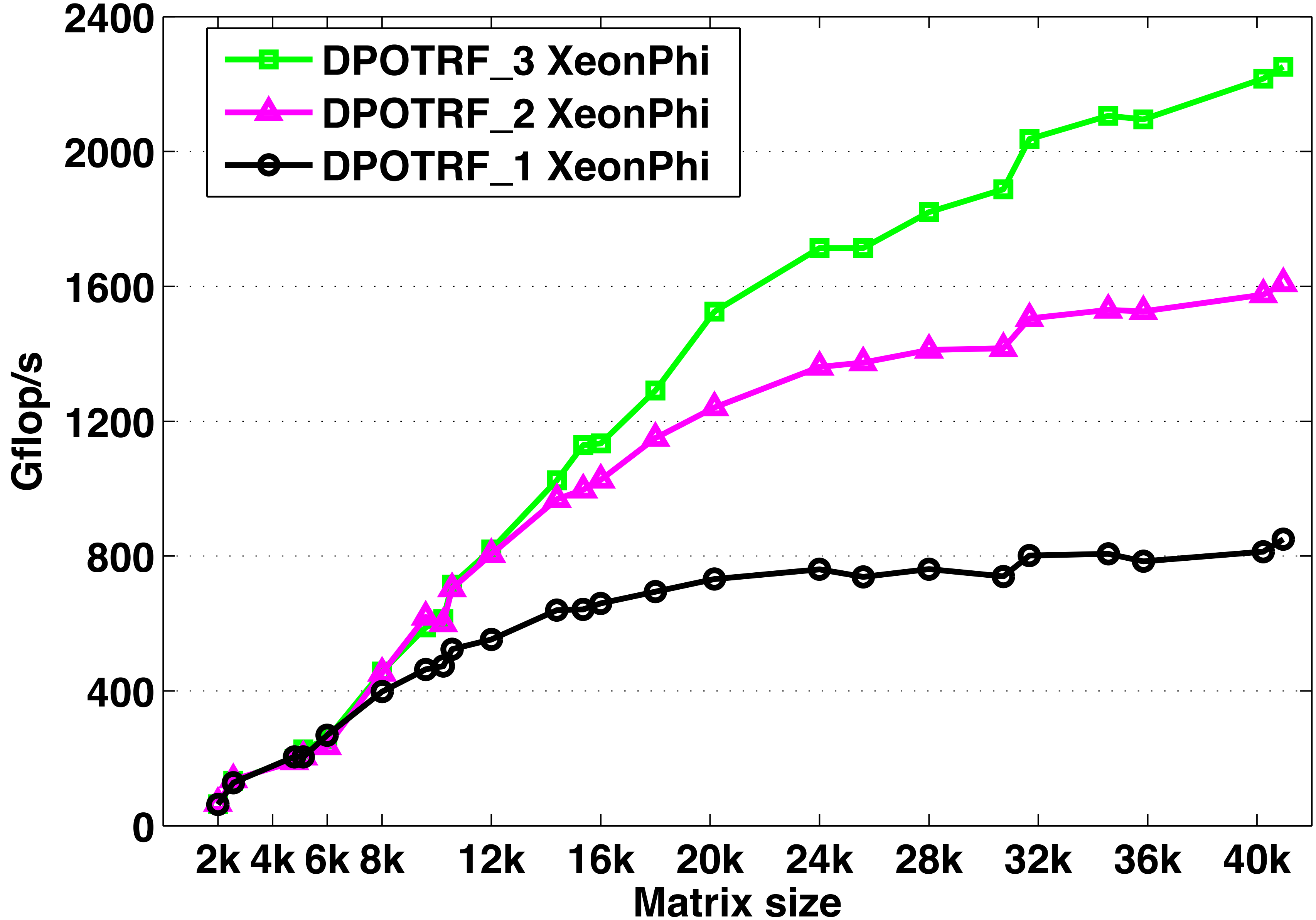
magma_quark DGEQRF Kepler K20c



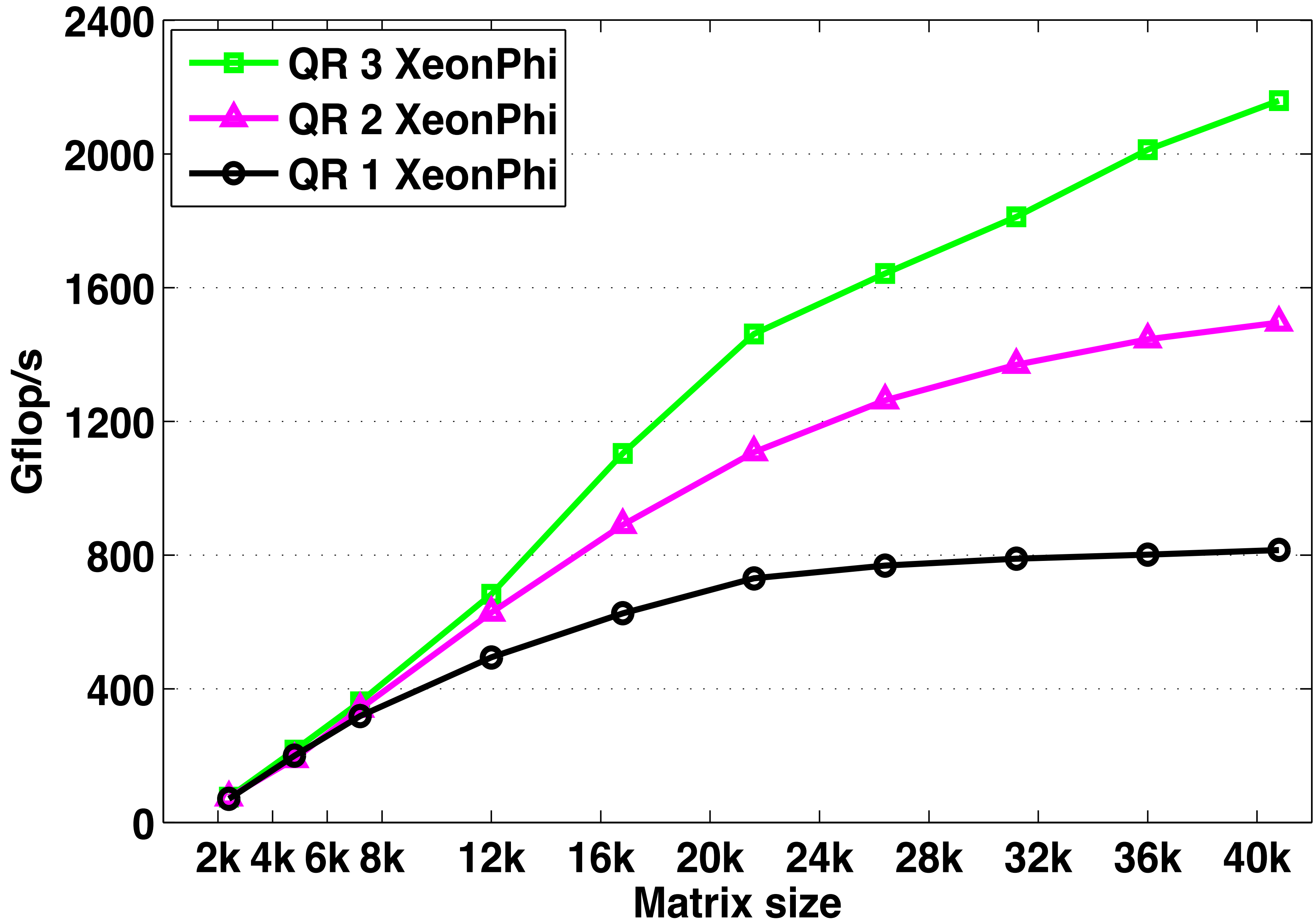
magma_quark scalability DPOTRF



magma_quark scalability DPOTRF Xeon-Phi



magma_quark scalability DGEQRF Xeon-Phi



Major Changes to Software & Algorithms

- **Must rethink the design of our algorithms and software**
  - **Another disruptive technology**
    - **Similar to what happened with cluster computing and message passing**
  - **Rethink and rewrite the applications, algorithms, and software**
  - **Data movement is expense**
  - **Flop/s are cheap, so are provisioned in excess**

Summary

- Major Challenges are ahead for extreme computing
  - Parallelism O(10^9)
    - Programming issues
  - Hybrid
    - Peak and HPL may be very misleading
    - No where near close to peak for most apps
  - Fault Tolerance
    - Today Sequoia BG/Q node failure rate is 1.25 failures/day
  - Power
    - 50 Gflops/w (today at 2 Gflops/w)
- We will need completely new approaches and technologies to reach the Exascale level

Collaborators / Software / Support

- **PLASMA**
  http://icl.cs.utk.edu/plasma/
- **MAGMA**
  http://icl.cs.utk.edu/magma/
- **Quark (RT for Shared Memory)**
- **http://icl.cs.utk.edu/quark/**
- **PaRSEC(**Parallel Runtime Scheduling and Execution Control**)**
- **http://icl.cs.utk.edu/parsec/**

- Collaborating partners
  University of Tennessee, Knoxville
  University of California, Berkeley
  University of Colorado, Denver

MAGMA   PLASMA