Weighted Dynamic Scheduling with Many Parallelism Grains for Offloading of Numerical Workloads to Multiple Varied Accelerators

Azzam Haidar
Yulu Jia
Piotr Luszczek
Stanimire Tomov
Asim YarKhan
Jack Dongarra

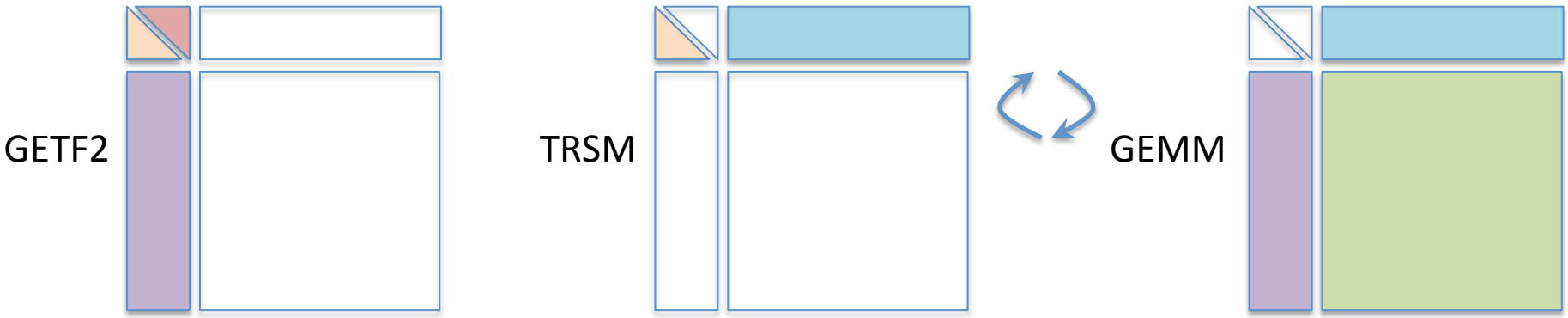
ICL INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY OF TENNESSEE KNOXVILLE

Problem Statement: Factorization

$Ax = b$ $\quad$ $PA = LU$ $\quad$ $Ly = P^{-1}b$ $\quad$ $Ux = y$

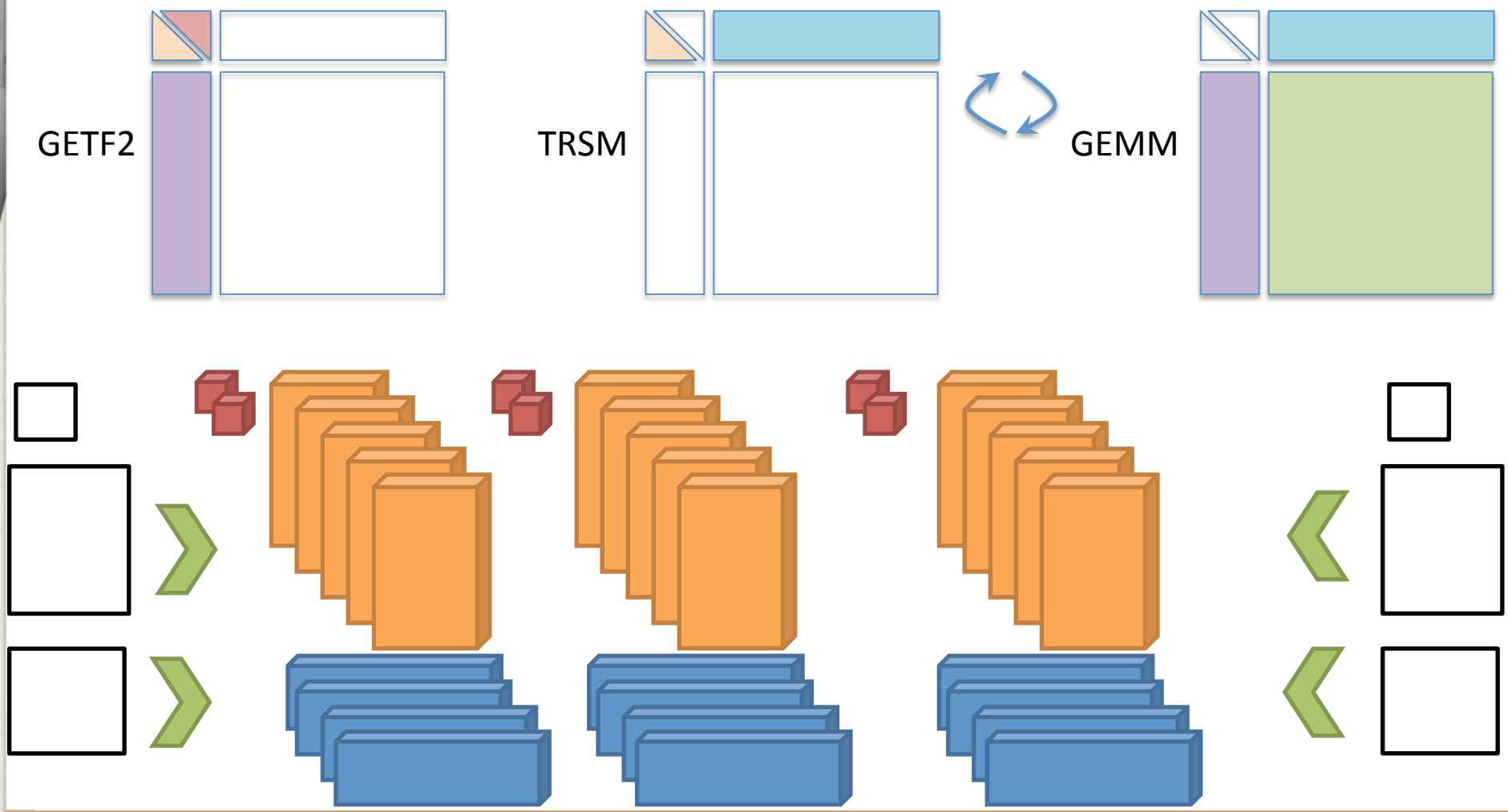
Problem Statement: Algorithm

$Ax = b$ $PA = LU$ $Ly = P^{-1}b$ $Ux = y$



Problem Statement: Mapping to Hardware

$Ax = b$ $PA = LU$ $Ly = P^{-1}b$ $Ux = y$



From Code to DAG

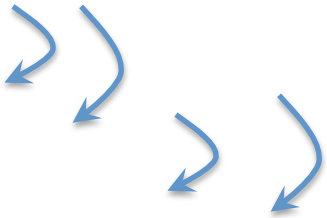
Start with canonical code:

```
GETRF(A[1:n, 1:n]) {
    for i = 1, nb, 2*nb, … {
        GETF2(A[i:n, i:i+nb])
        TRSM(A[i:i+nb, i:i+nb], A[i:i+nb, i:n])
        GEMM(A[i:n, i:i+nb], A[i:i+nb, i:n], A[i+nb:n, i+nb:n])
    }
}
```

Unwind the function calls:

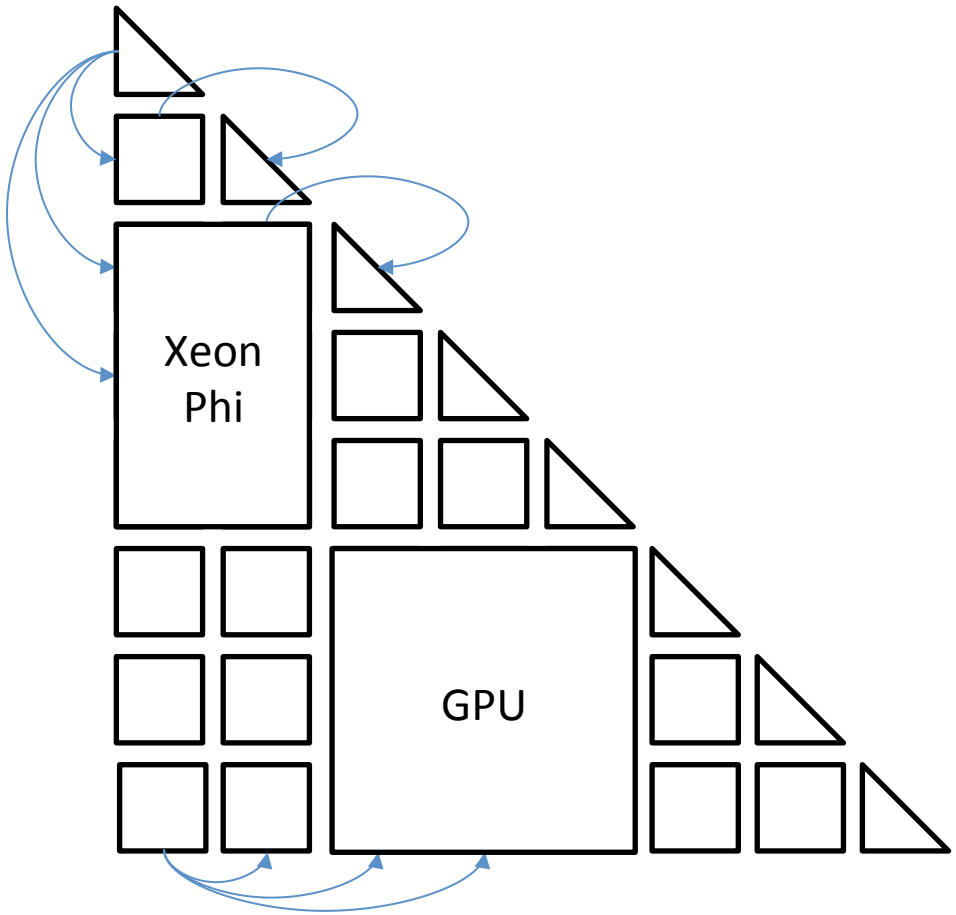
```
GETF2(A[1,1], A[2,1], …)
TRSM(A[1,1], A[1,2])
TRSM(A[1,1], A[1,3])
GEMM(A[2,1], A[1,2], A[2,2])
```

Track dependences:



The function calls and their dependences form a DAG.

Matrix-View of Dependences



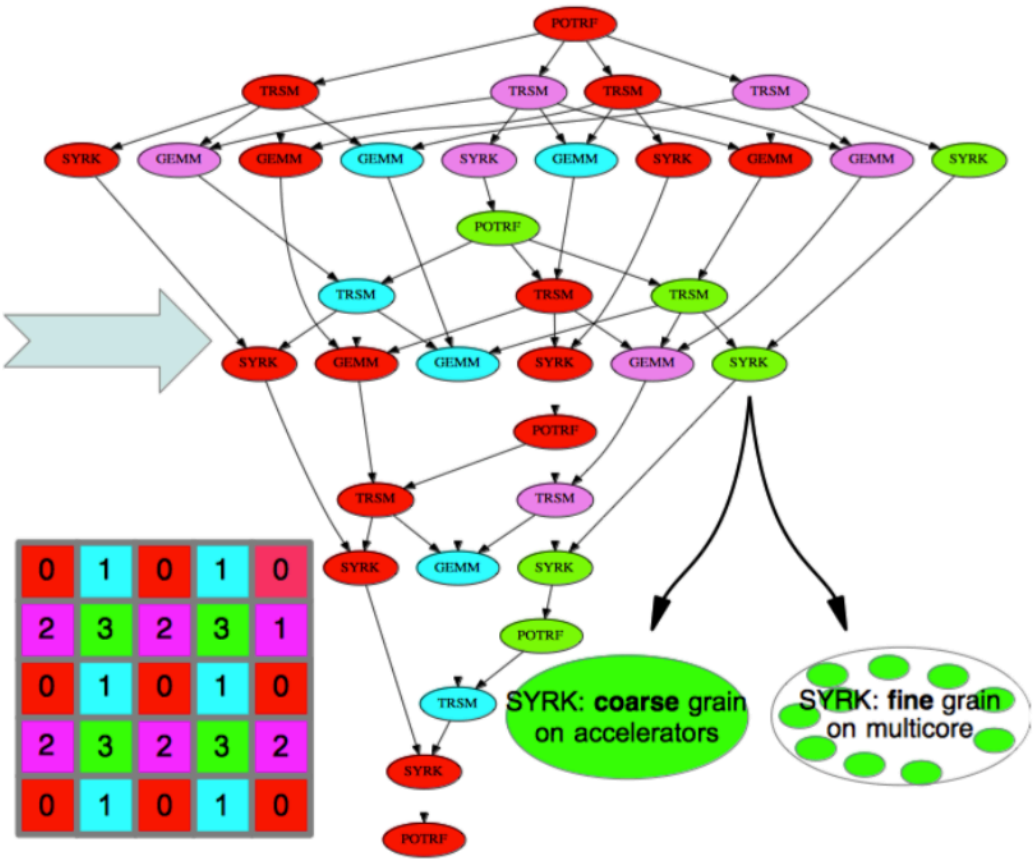
From Code to DAG

From sequential
LAPACK

```
for( j=0, j<n; j+=nb) {
    jb = min(nb, n-j);
    MagmaUpper
    If (j+jb) < n) {
    ( MagmaLeft,
                jb, n
    ( MagmaUpper
        jb, j, one, dA(0
    ( MagmaCo
        dA(0,j), ldd
  }
}
```

to parallel (distributed and heterogeneous)
LAPACK algorithms

```
for( j=0, j<n; j+=nb) {
  jb = min(nb, n-j);
  ( MagmaUpperStr, &jb, work, &jl
  If (j+jb) < n) {
    broadcast_column_procs(A(j,j));
    MagmaLeft, MagmaUpper,al
                dA(j,j+jb), ldda, que
    broadcast_specific_proc(A(j,j));
    broadcast_multi-devices(A(j,j));
    ( MagmaUpper,
                one, dA(j,j), ldda, qu
    ( MagmaConjTrans
                dA(0,j), ldda, dA(0,j+j
  }
}
```



Asynchronous Algorithm

for $k \in \{0, nb, 2 \times nb, \ldots, n\}$ **do**

Factorization of the panel A(k:n,k:k+nb)

TASK: getf2(A(k:n,k:k+nb))

swap the rows to the left and the right of the panel

TASK: laswp(A(k:n,1:k))

TASK: laswp(A(k:n,k+nb:n))

Schur complement update: The Schur update task, $A_{22} \leftarrow A_{22} - A_{21}A_{11}^{-1}A_{12}$, is split into a set of parallel, compute intensive tasks that call trsm and gemm. This increases parallelism and enhances the overall performance. Note that the first task (trsm and gemm) consists of the update of the next panel. The scheduler inserts it immediately in the queue since with a high priority. Right after the dependences are satisfied and once the high priority tasks finish, the data are sent it to the CPU in order to perform the panel factorization of the next step – this technique is called lookahead.

for $j \in \{k+nb, k+2nb, \ldots, n-nb\}$ **do**

trsm(A(k:k+nb,k:k+nb) $\rightarrow$ A(k:k+nb,j:j+nb)

if $panel_m > panel_n$ **then**

gemm with trailing matrix

for $j \in \{k+nb, k+2nb, \ldots, n-nb\}$ **do**

gemm(A(j:n,k:k+nb) $\times$ A(k:k+nb,j,j+nb) $\rightarrow$ A(j:n,j:j+nb))

Panel factorization

Pivoting (row swaps)

Schedule tasks according to:
- data sizes
- hardware weights
- affinity
- perform lookahead

Triangular solve

Matrix multiply (Schur complement)

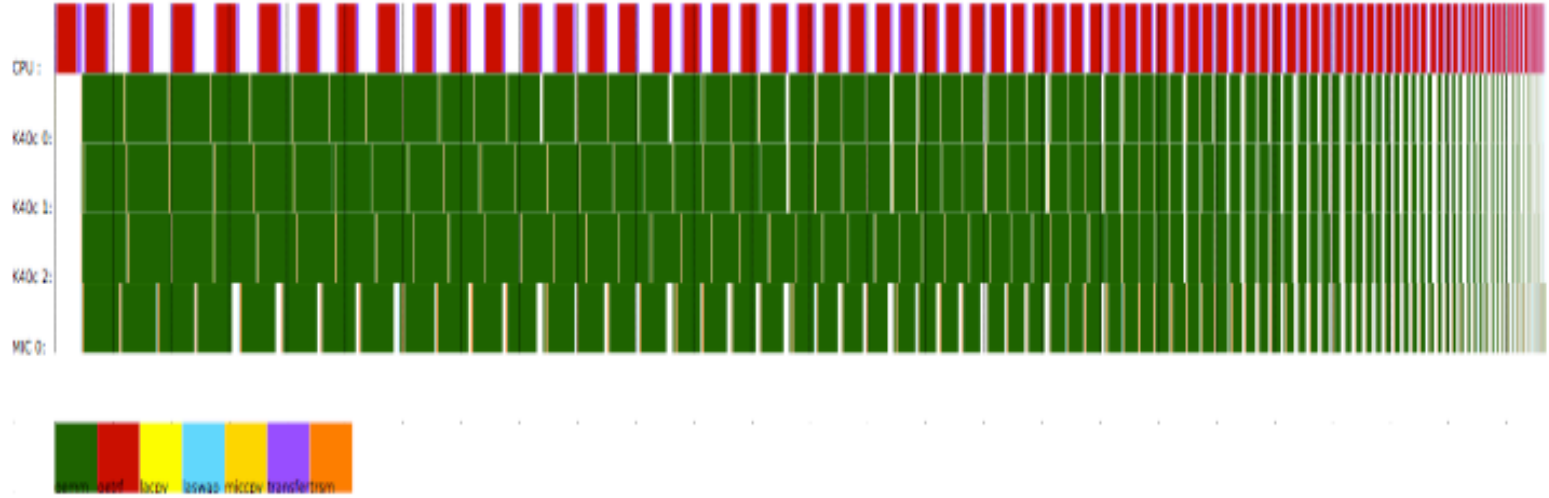
Main Features of the Scheduler

- Resource capability weight for the task
  - w(kernel, device)
- Adaptive scheduling with weights
- Dynamic lookahead
- Enhanced task priorities
  - Regulates lookahead (DAG exploration order)
- Transparent data movement
  - Uses (and tracks) asynchronous data transfers
- Dynamic data redistribution
  - Data is moved as needed and only if needed

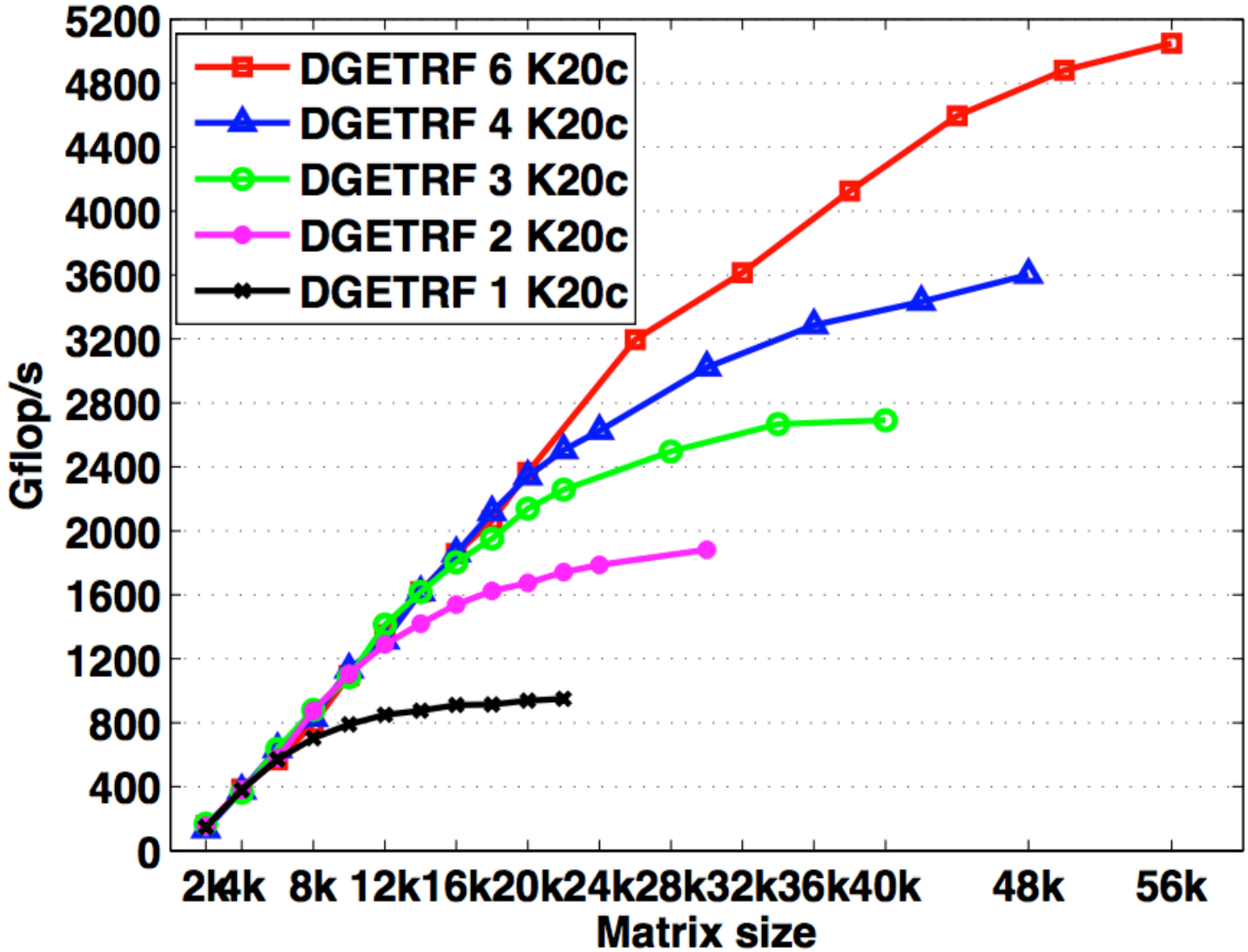
Runtime Scheduler Loop

```
def main_thread_loop(user_code, queues, threshold):
    # if there are enough tasks for cores
    if queues.total_length() > threshold:
        # resume user's code for submitting tasks
        task = user_code.get_next_task()
        q = queues.find_closes_queue(task.devices())
        q.insert(task, task.priority())
    else:
        task = queues.steal_task(main_cpu)
        # if tasks available for stealing
        if not task.is_empty():
            task.execute() # execute single task
        else:
            queues.wait_for_tasks()
```

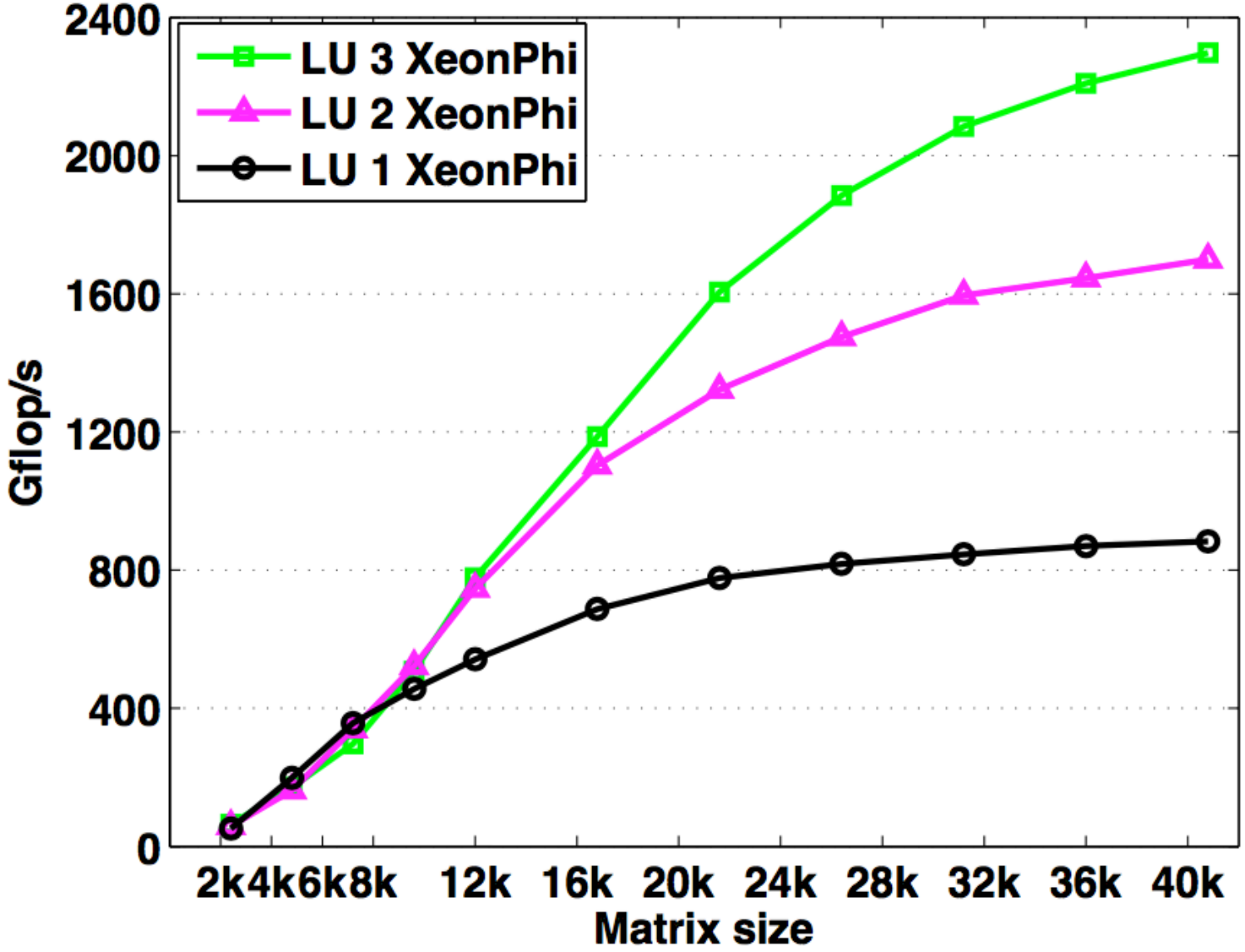
Effect of Dynamic Scheduling



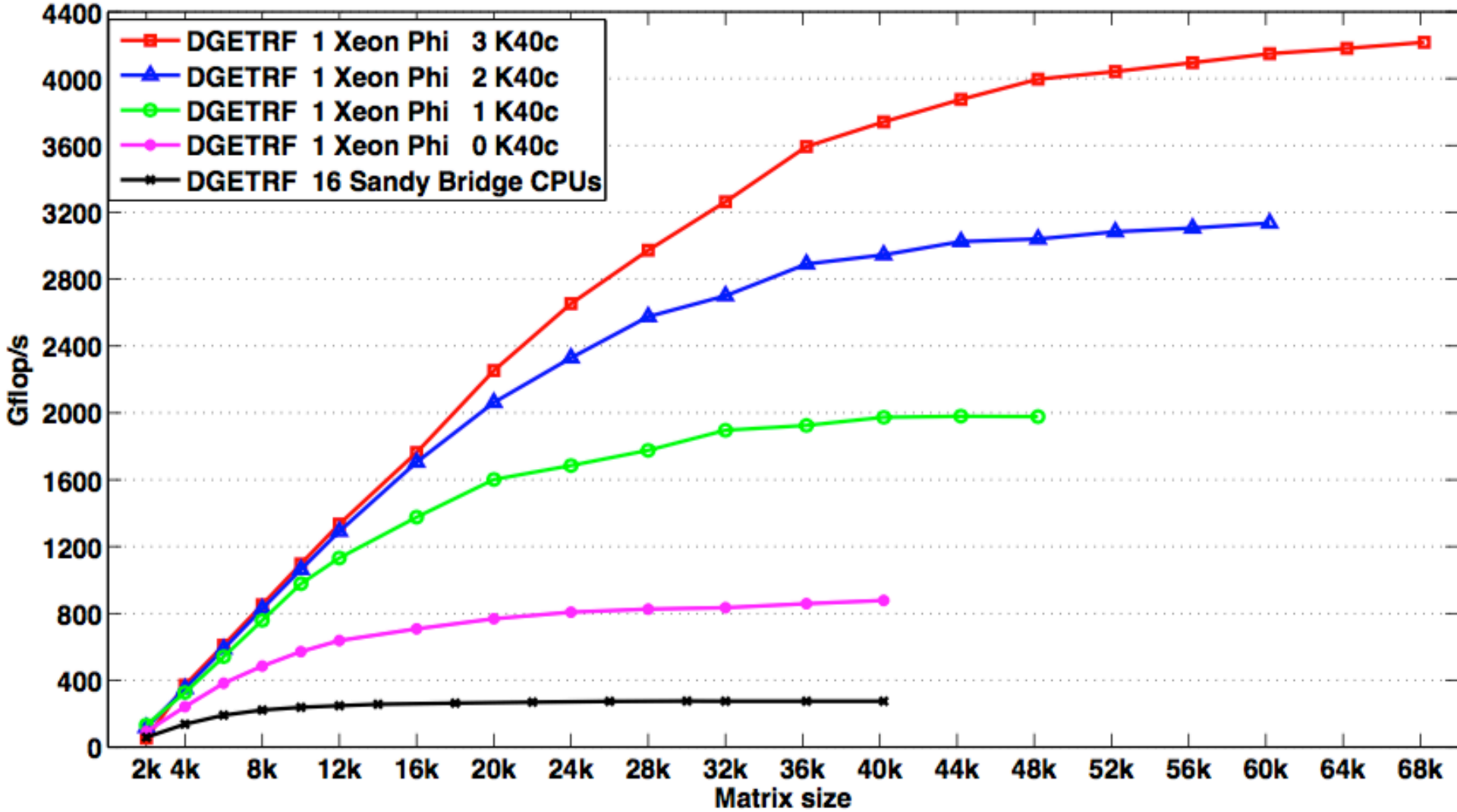
Performance: Kepler K20



Performance: Xeon Phi



Performance: Xeon Phi + Kepler K40



Summary of Contributions

- Fine- and coarse-grained tasks for scheduling
- Capability weights for hardware description
- Unified scheduling across CPUs, GPUs, and coprocessors
- Synchronous memory-transfer model with transparent asynchronous progress