Hybrid Empirical and Simulation-Based Autotuning of a Dense Linear Algebra Library for Heterogeneous Architectures

Jesús Cámara

Joint work with

E. Agullo, J. Cuenca and D. Giménez

SIAM Conference on Parallel Processing for Scientific Computing

Seattle, Washington, February 2020

Overview

1. Motivation
2. Application Case
3. Search of the AP values
4. Selection of Computing Units
5. Selection of Scheduling Policy
6. Ongoing Work
7. Conclusions

Overview

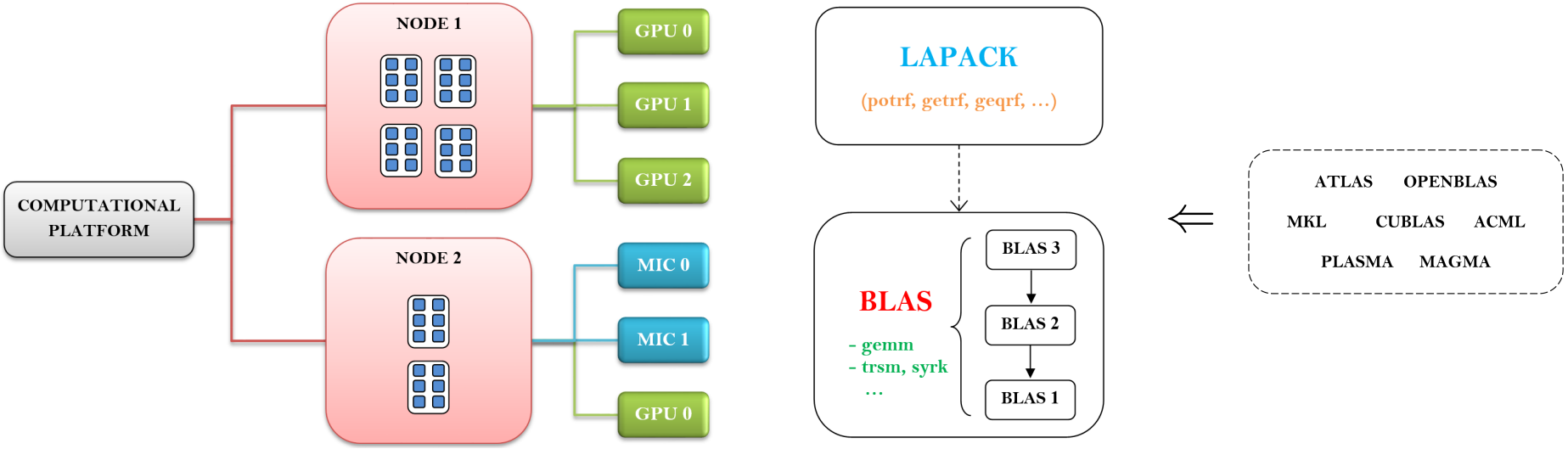
1. Motivation
2. Application Case
3. Search of the AP values
4. Selection of Computing Units
5. Selection of Scheduling Policy
6. Ongoing Work
7. Conclusions

Motivation

The complexity of modern computational platforms makes the design of high performance numerical libraries extremely challenging.

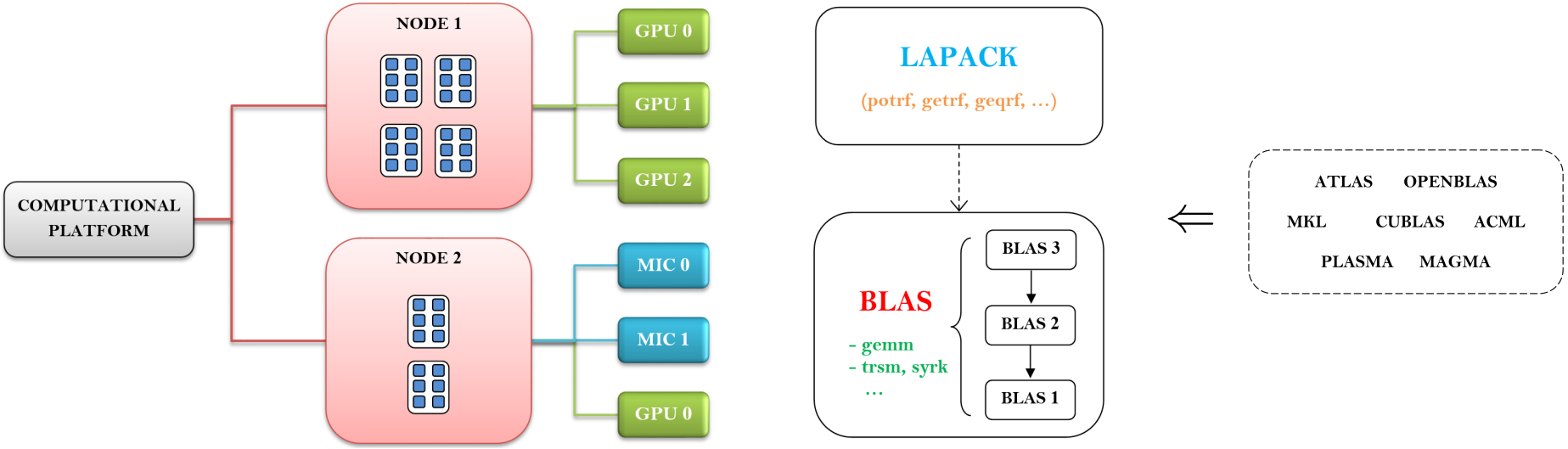
Motivation

The complexity of modern computational platforms makes the design of high performance numerical libraries extremely challenging.



Motivation

The complexity of modern computational platforms makes the design of high performance numerical libraries extremely challenging.



<u>Task-Based Libraries</u> = Sequential Task Flow (STF) design of Linear Algebra Routines on top of Runtime Systems.

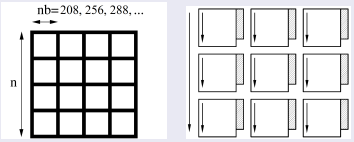
Motivation

Chameleon: Dense Linear Algebra Library derived from PLASMA (tile algorithms) with a STF design on top of Runtime Systems.

Linear Algebra

$AX = B$

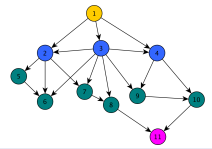
Tile Matrix Layout



Sequential-Task-Flow

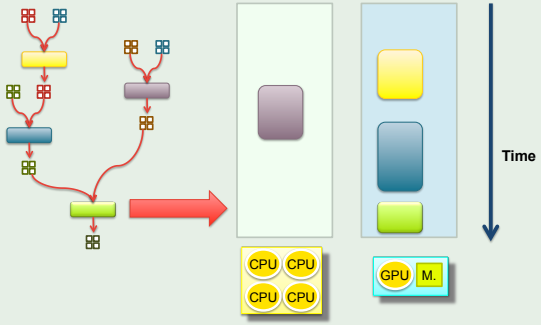
```
for (j = 0; j < N; j++)
    Task(A[j]);
```

Direct Acyclic Graph



$\Rightarrow$

Runtime Systems (Scheduling of Tasks)



StarPU

Optimized Kernels

MKL, cuBLAS, ...

$\Rightarrow$

Heterogeneous Platforms

Motivation

The Runtime System:

- Handles data consistency.
- Handles data dependencies.
- Handles scheduling of tasks to the CUs.
- ...but it does not take care about the best value for some APs.

Motivation

The Runtime System:

- Handles data consistency.
- Handles data dependencies.
- Handles scheduling of tasks to the CUs.
- ...but it does not take care about the best value for some APs.

Motivation

The Runtime System:

- Handles data consistency.
- Handles data dependencies.
- Handles scheduling of tasks to the CUs.
- ...but it does not take care about the best value for some APs.

Motivation

The Runtime System:

- Handles data consistency.
- Handles data dependencies.
- Handles scheduling of tasks to the CUs.
- ...but it does not take care about the best value for some APs.

Motivation

The Runtime System:

- Handles data consistency.
- Handles data dependencies.
- Handles scheduling of tasks to the CUs.
- ...but it does not take care about the best value for some APs.

Motivation

The Runtime System:

- Handles data consistency.
- Handles data dependencies.
- Handles scheduling of tasks to the CUs.
- ...but it does not take care about the best value for some APs.

Which APs?

Motivation

The Runtime System:

- Handles data consistency.
- Handles data dependencies.
- Handles scheduling of tasks to the CUs.
- ...but it does not take care about the best value for some APs.

Which APs?

- Tile Size (nb)

Motivation

The Runtime System:

- Handles data consistency.
- Handles data dependencies.
- Handles scheduling of tasks to the CUs.
- ...but it does not take care about the best value for some APs.

Which APs?

- Tile Size (*nb*)
- Inner Block Size (*ib*)

Motivation

The Runtime System:

- Handles data consistency.
- Handles data dependencies.
- Handles scheduling of tasks to the CUs.
- ...but it does not take care about the best value for some APs.

Which APs?

- Tile Size (*nb*)
- Inner Block Size (*ib*)

Other Parameters?

Motivation

The Runtime System:

- Handles data consistency.
- Handles data dependencies.
- Handles scheduling of tasks to the CUs.
- ...but it does not take care about the best value for some APs.

Which APs?

- Tile Size (*nb*)
- Inner Block Size (*ib*)

Other Parameters?

- Number of Computing Units (CUs)

Motivation

The Runtime System:

- Handles data consistency.
- Handles data dependencies.
- Handles scheduling of tasks to the CUs.
- ...but it does not take care about the best value for some APs.

Which APs?

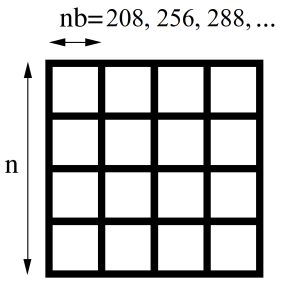
- Tile Size (nb)
- Inner Block Size (ib)

Other Parameters?

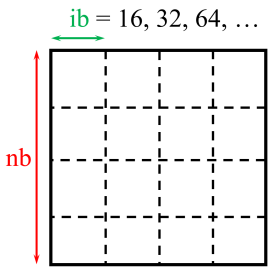
- Number of Computing Units (CUs)
- Scheduling Policies.

Optimization Strategies

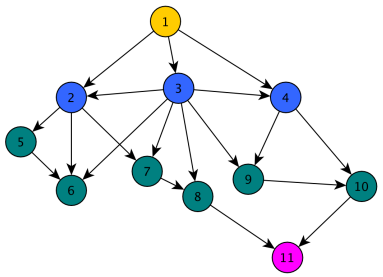
nb



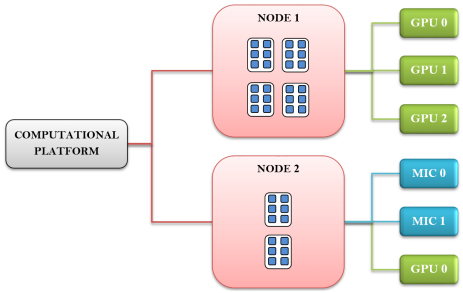
ib



Scheduling Policies



Computing Units

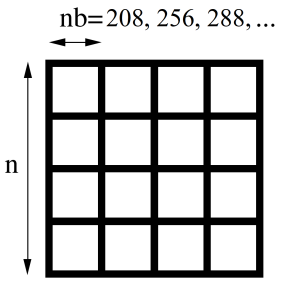


Auto-Tuning

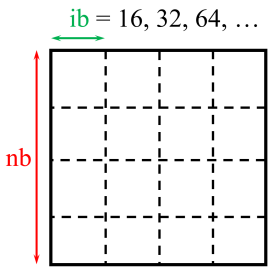
Process to obtain the best value for some algorithmic parameters in order to reduce the execution time of the routines with an efficient use of the heterogeneous platform.

Optimization Strategies

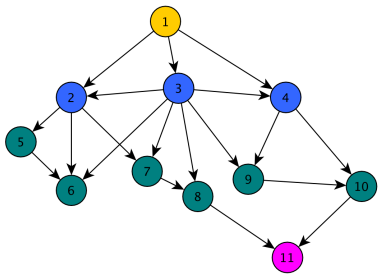
nb



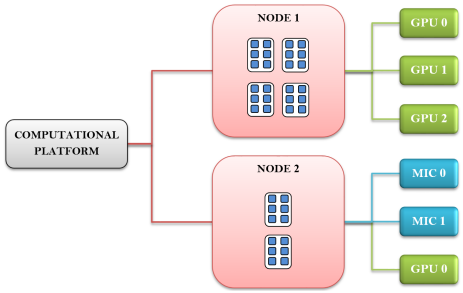
ib



Scheduling Policies



Computing Units



Auto-Tuning

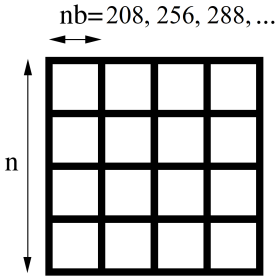
Process to obtain the best value for some algorithmic parameters in order to reduce the execution time of the routines with an efficient use of the heterogeneous platform.

Trade off: Performance vs. Concurrency

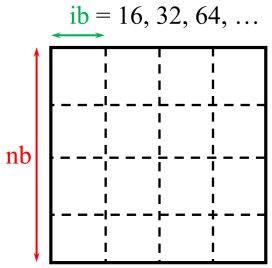
- Large *nb*: fast kernels (↑ performance) but few tasks (↓ concurrency)
- Small *nb*: many tasks (↑ concurrency) but slow kernels (↓ performance)

Optimization Strategies

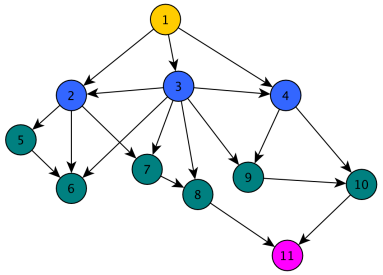
nb



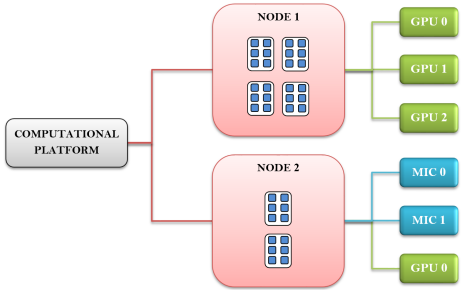
ib



Scheduling Policies



Computing Units

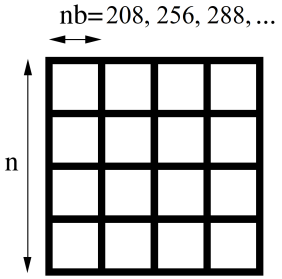


Goal

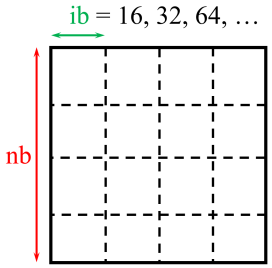
For each problem size *n*:

Optimization Strategies

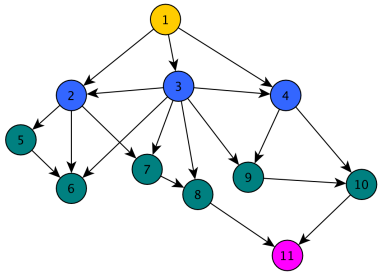
nb



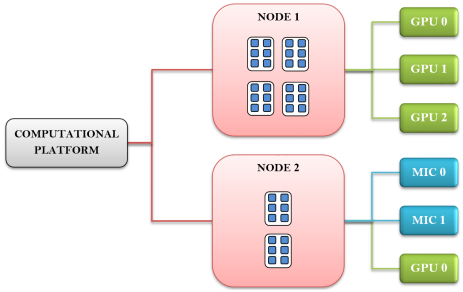
ib



Scheduling Policies



Computing Units



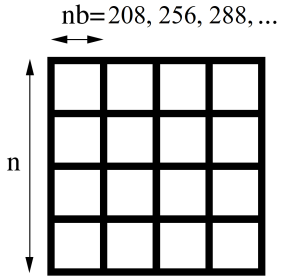
Goal

For each problem size *n*:

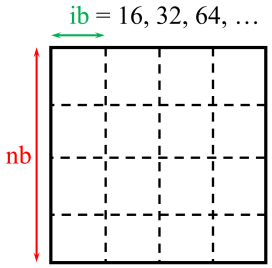
- Find the best values for *nb* and *ib*.

Optimization Strategies

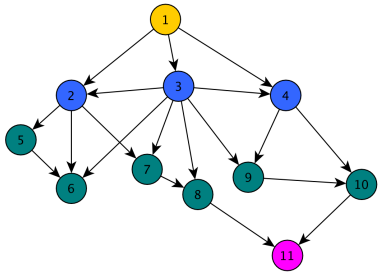
nb



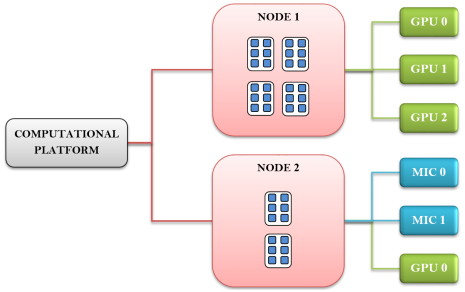
ib



Scheduling Policies



Computing Units



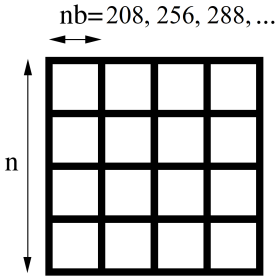
Goal

For each problem size *n*:

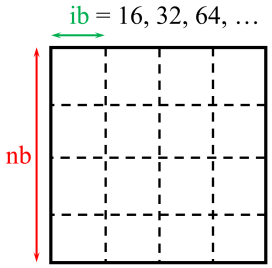
- Find the best values for *nb* and *ib*.
- Select the number of computing units (CPU threads and GPUs)

Optimization Strategies

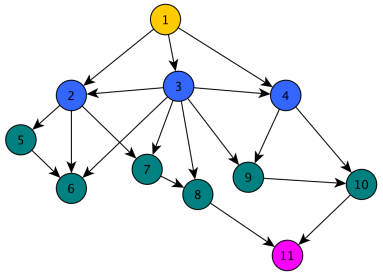
nb



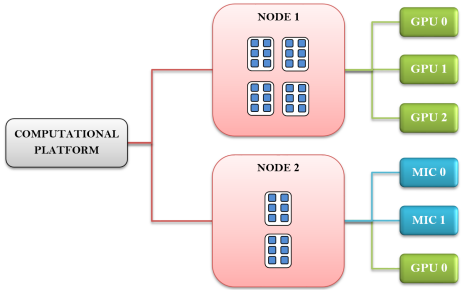
ib



Scheduling Policies



Computing Units



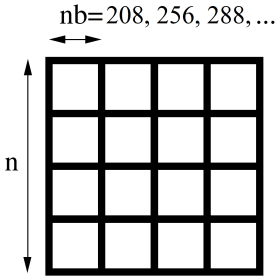
Goal

For each problem size *n*:

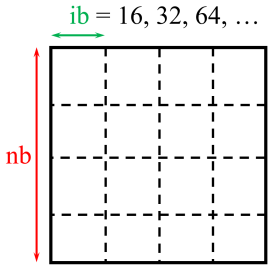
- Find the best values for *nb* and *ib*.
- Select the number of computing units (CPU threads and GPUs)
- Select the best Scheduling Policy.

Optimization Strategies

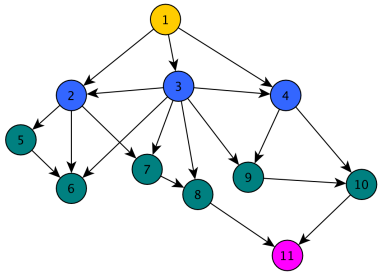
nb



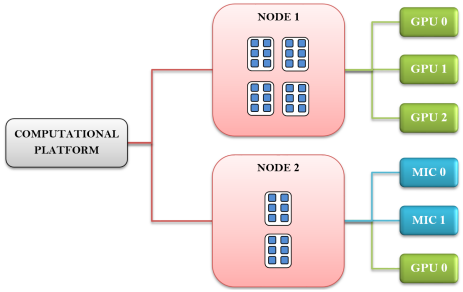
ib



Scheduling Policies



Computing Units



Goal

For each problem size *n*:

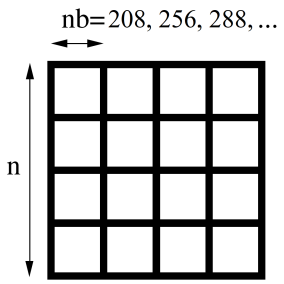
- Find the best values for *nb* and *ib*.
- Select the number of computing units (CPU threads and GPUs)
- Select the best Scheduling Policy.

Two Strategies (S1, S2):

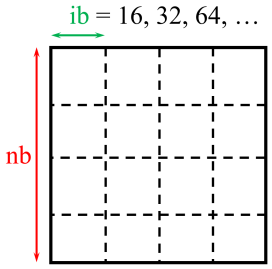
- **S1**: Empirical+Exhaustive.
- **S2**: Empirical+Pruned.

Optimization Strategies

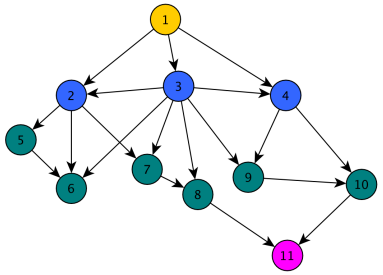
nb



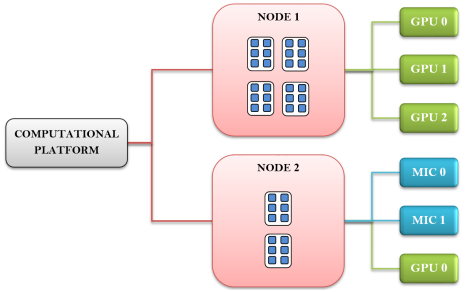
ib



Scheduling Policies



Computing Units



Goal

For each problem size *n*:

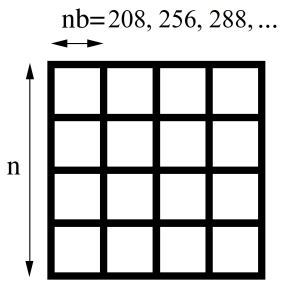
- Find the best values for *nb* and *ib*.
- Select the number of computing units (CPU threads and GPUs)
- Select the best Scheduling Policy.

Two Strategies (S1, S2):

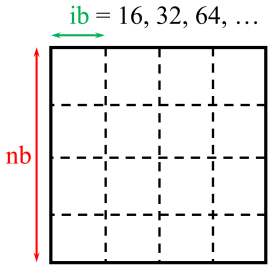
- **S1**: Empirical+Exhaustive.
- **S2**: Empirical+Pruned.

Optimization Strategies

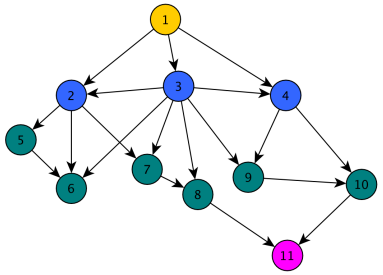
nb



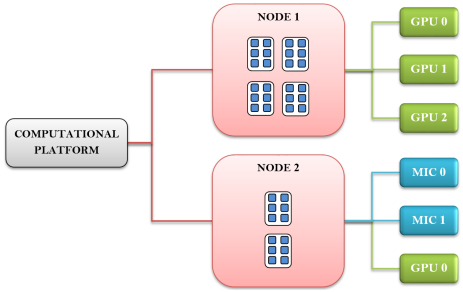
ib



Scheduling Policies



Computing Units



Goal

For each problem size *n*:

- Find the best values for *nb* and *ib*.
- Select the number of computing units (CPU threads and GPUs)
- Select the best Scheduling Policy.

Two Strategies (S1, S2):

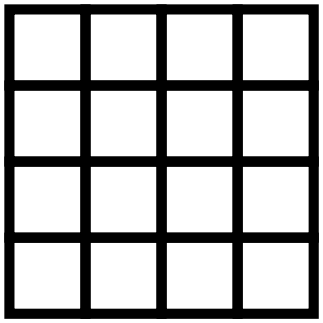
- **S1**: Empirical+Exhaustive.
- **S2**: Empirical+Pruned.

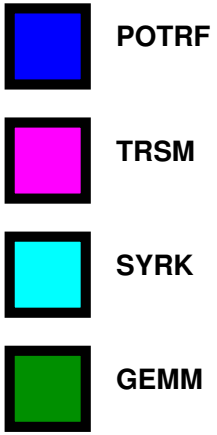
Overview

1. Motivation
2. Application Case
3. Search of the AP values
4. Selection of Computing Units
5. Selection of Scheduling Policy
6. Ongoing Work
7. Conclusions

The Cholesky Decomposition

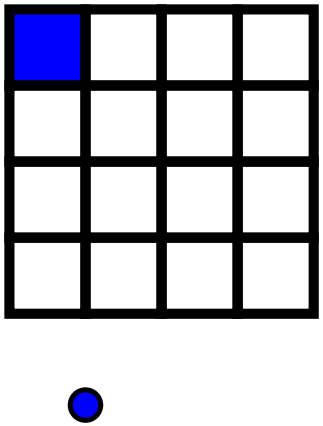
```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```

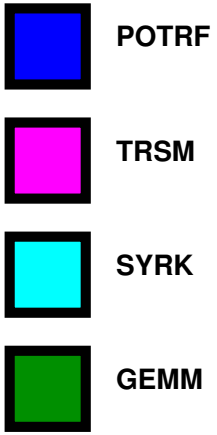




The Cholesky Decomposition

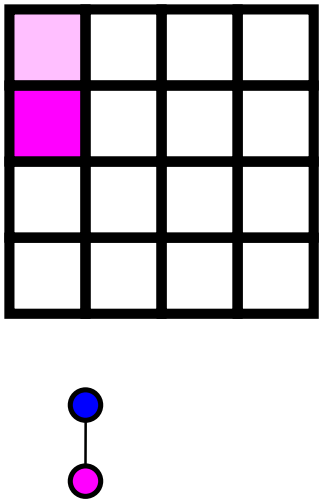
```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```

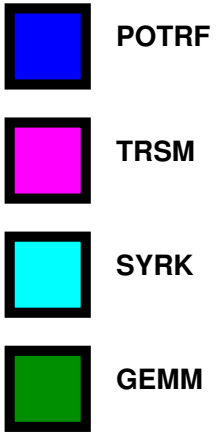




The Cholesky Decomposition

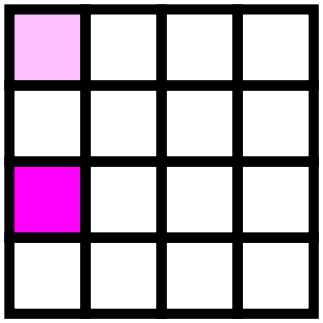
```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```

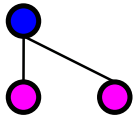


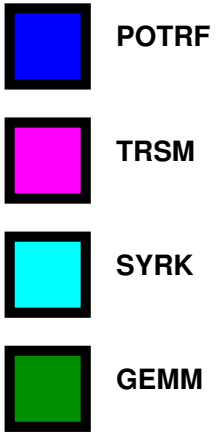


The Cholesky Decomposition

```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```

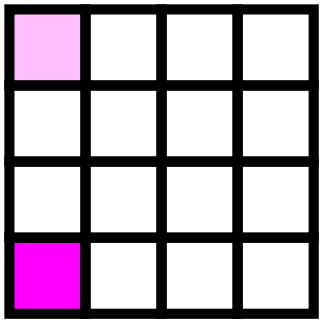


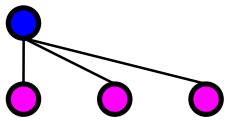


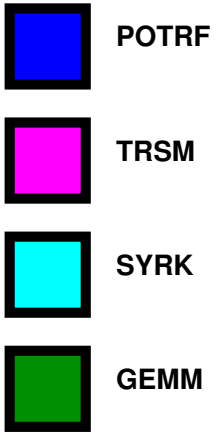


The Cholesky Decomposition

```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```

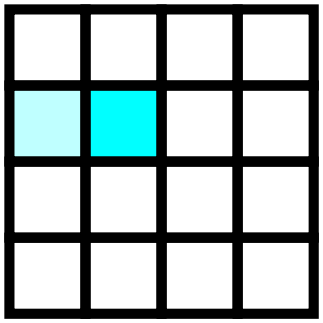


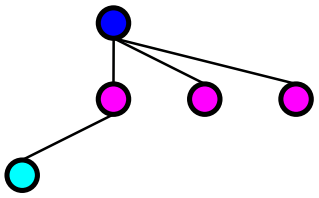


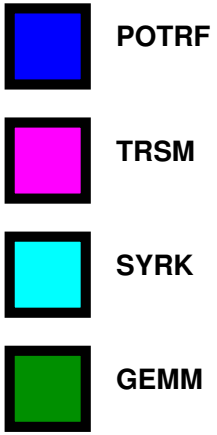


The Cholesky Decomposition

```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```

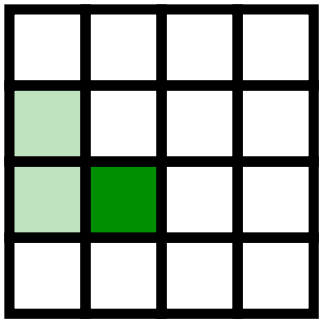


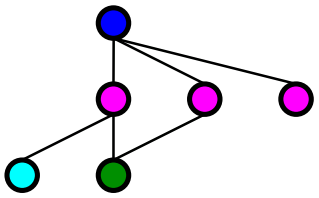


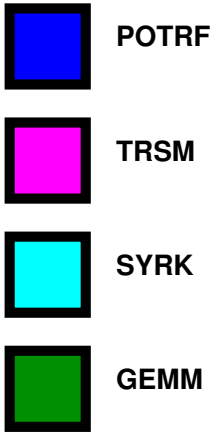


The Cholesky Decomposition

```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```

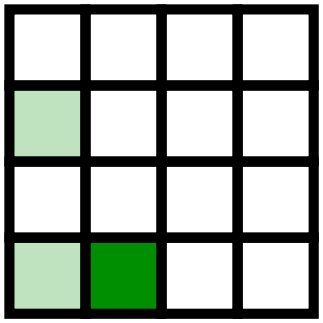


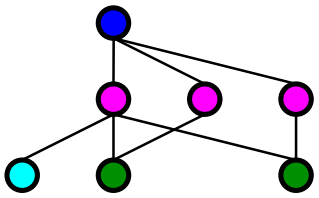


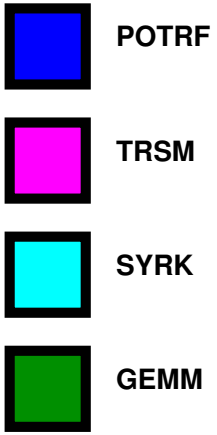


The Cholesky Decomposition

```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```

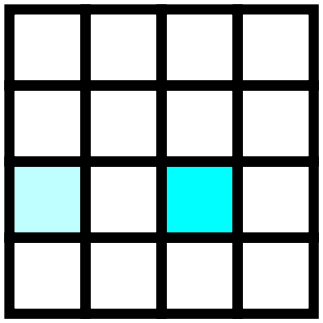


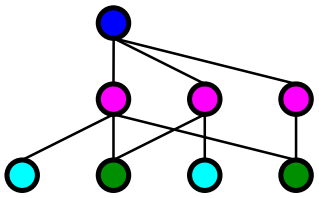


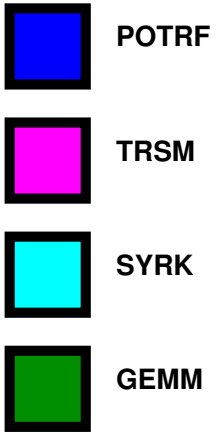


The Cholesky Decomposition

```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```

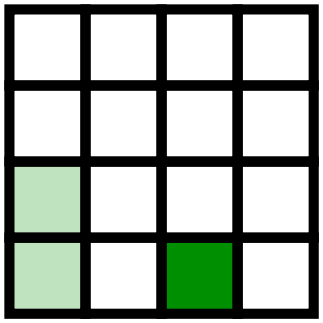


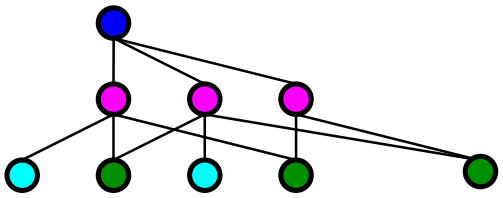


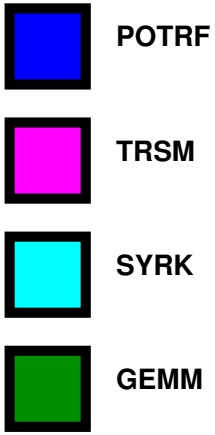


The Cholesky Decomposition

```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```

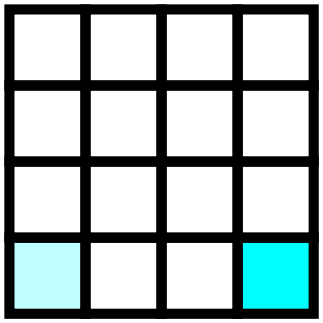


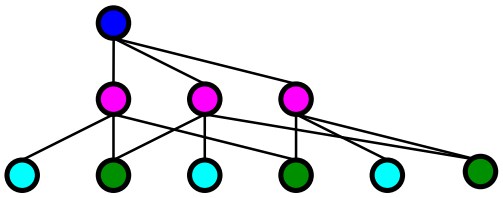


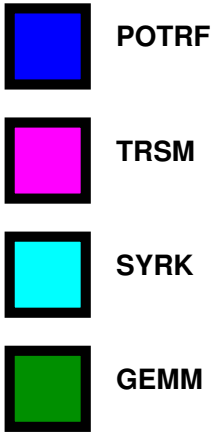


The Cholesky Decomposition

```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```

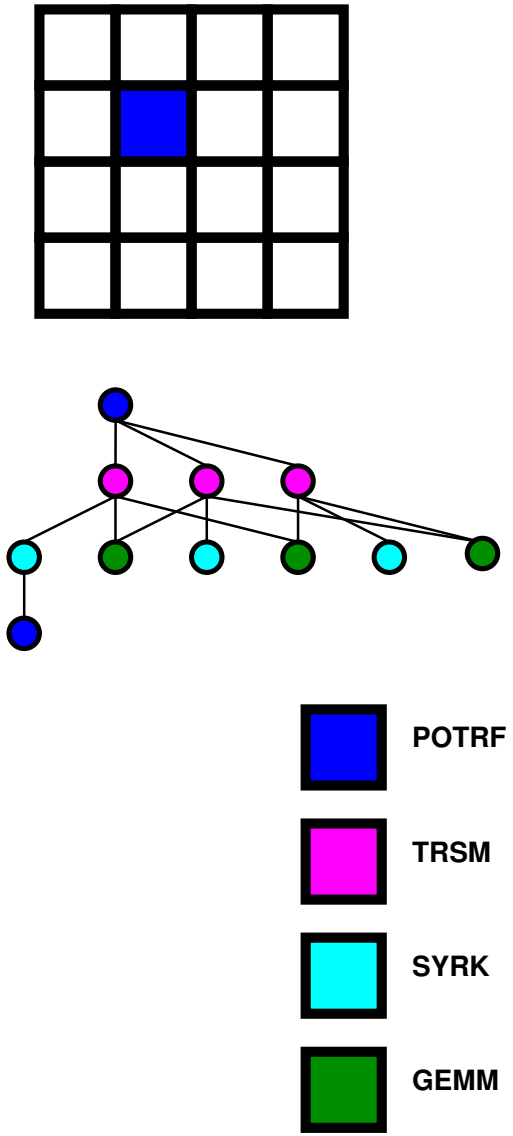






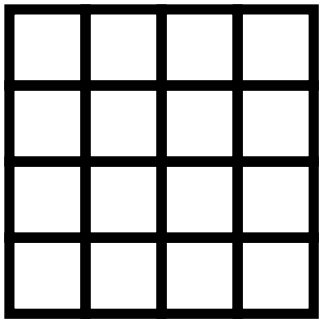
The Cholesky Decomposition

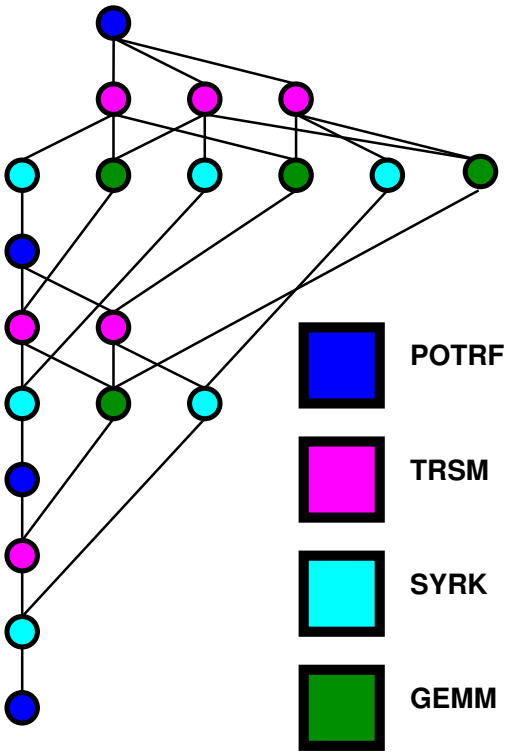
```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```



The Cholesky Decomposition

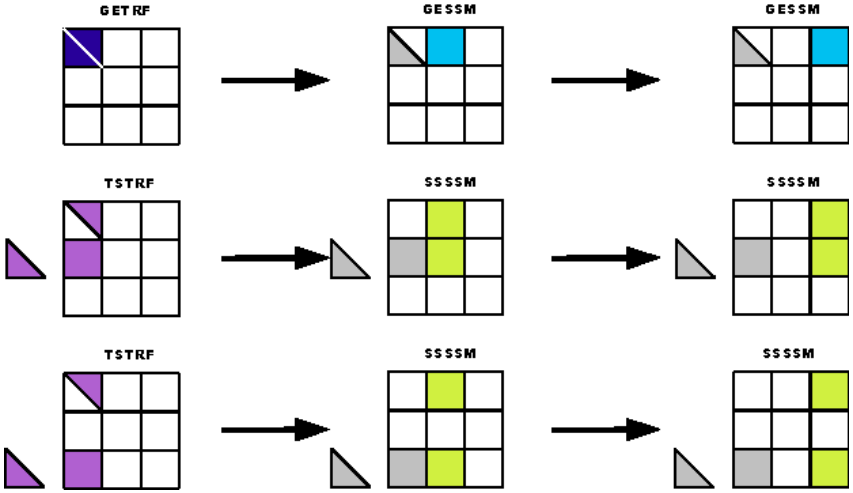
```
for (j = 0; j < n/nb; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < n/nb; i++) {
        TRSM (RW,A[i][j], R,A[j][j]);
    }
    for (i = j+1; i < n/nb; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++) {
            GEMM (RW,A[i][k],
                  R,A[i][j],
                  R,A[k][j]);
        }
    }
}
```



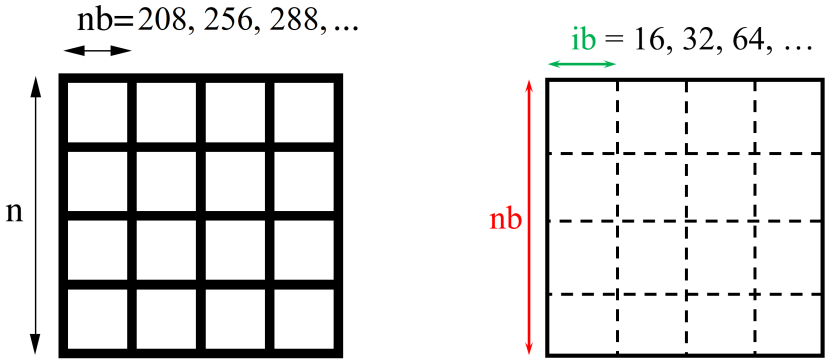


The LU Factorization

```
for (j = 0; j < n/nb; j++) {
    DGETRF(R,A[j][j], W,L[j][j], W,U[j][j]);
    for (i = j+1; i < n/nb; i++)
        DGESSM(R,A[j][i], R,L[j][j], W,U[j][i]);
    for (i = j+1; i < n/nb; i++) {
        DTSTRF(R,A[i][j], R,L[i][j], RW,U[j][j]);
        for (k = j+1; k < n/nb; k++)
            DSSSSM(R,L[i][j], RW,U[j][k], RW,A[i][k]);
    }
}
```



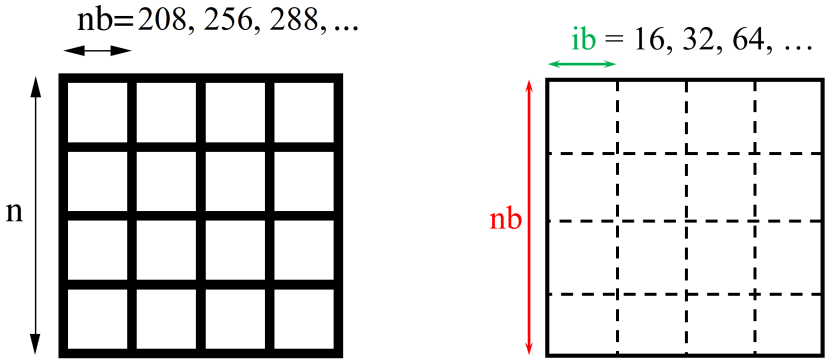
Algorithmic Parameters



Goal

Given a problem size n:

Algorithmic Parameters

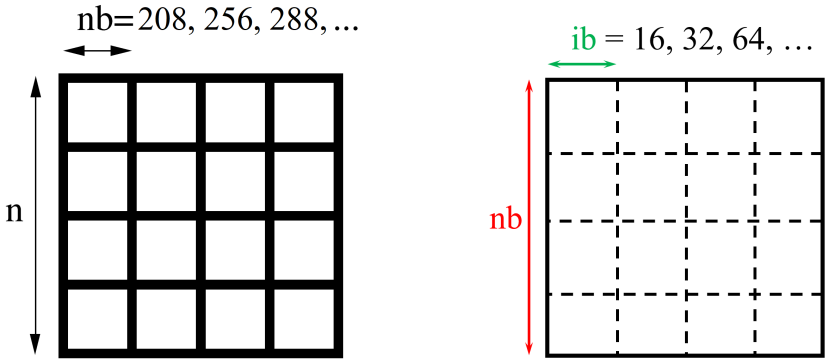


Goal

Given a problem size *n*:

- **Cholesky**: find the best *nb* value.

Algorithmic Parameters

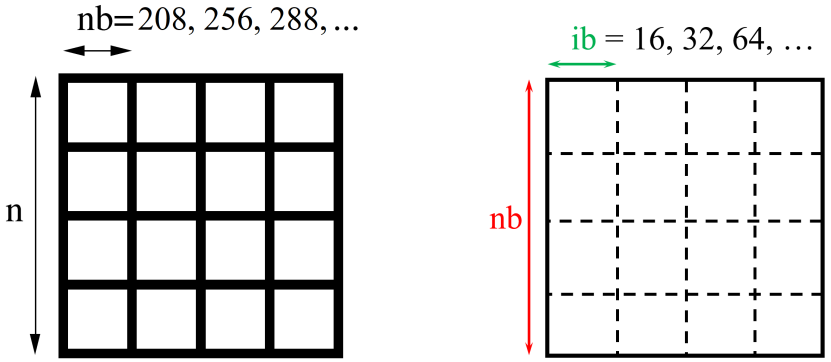


Goal

Given a problem size n:

- **Cholesky**: find the best nb value.
- **LU**: find the best (nb, ib) combination.

Algorithmic Parameters



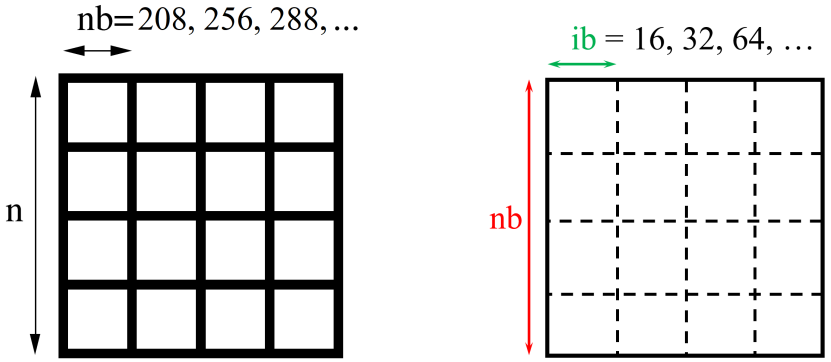
Goal

Given a problem size n:

- **Cholesky**: find the best nb value.
- **LU**: find the best (nb, ib) combination.

Which nb range?

Algorithmic Parameters



Goal

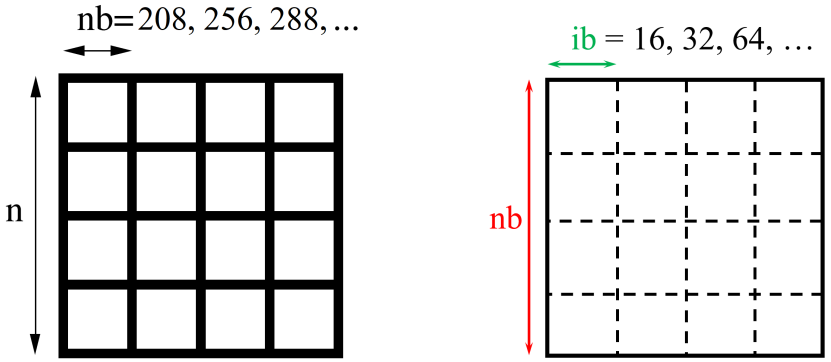
Given a problem size n:

- **Cholesky**: find the best nb value.
- **LU**: find the best (nb, ib) combination.

Which nb range?

- In principle any.

Algorithmic Parameters



Goal

Given a problem size n:

- **Cholesky**: find the best nb value.
- **LU**: find the best (nb, ib) combination.

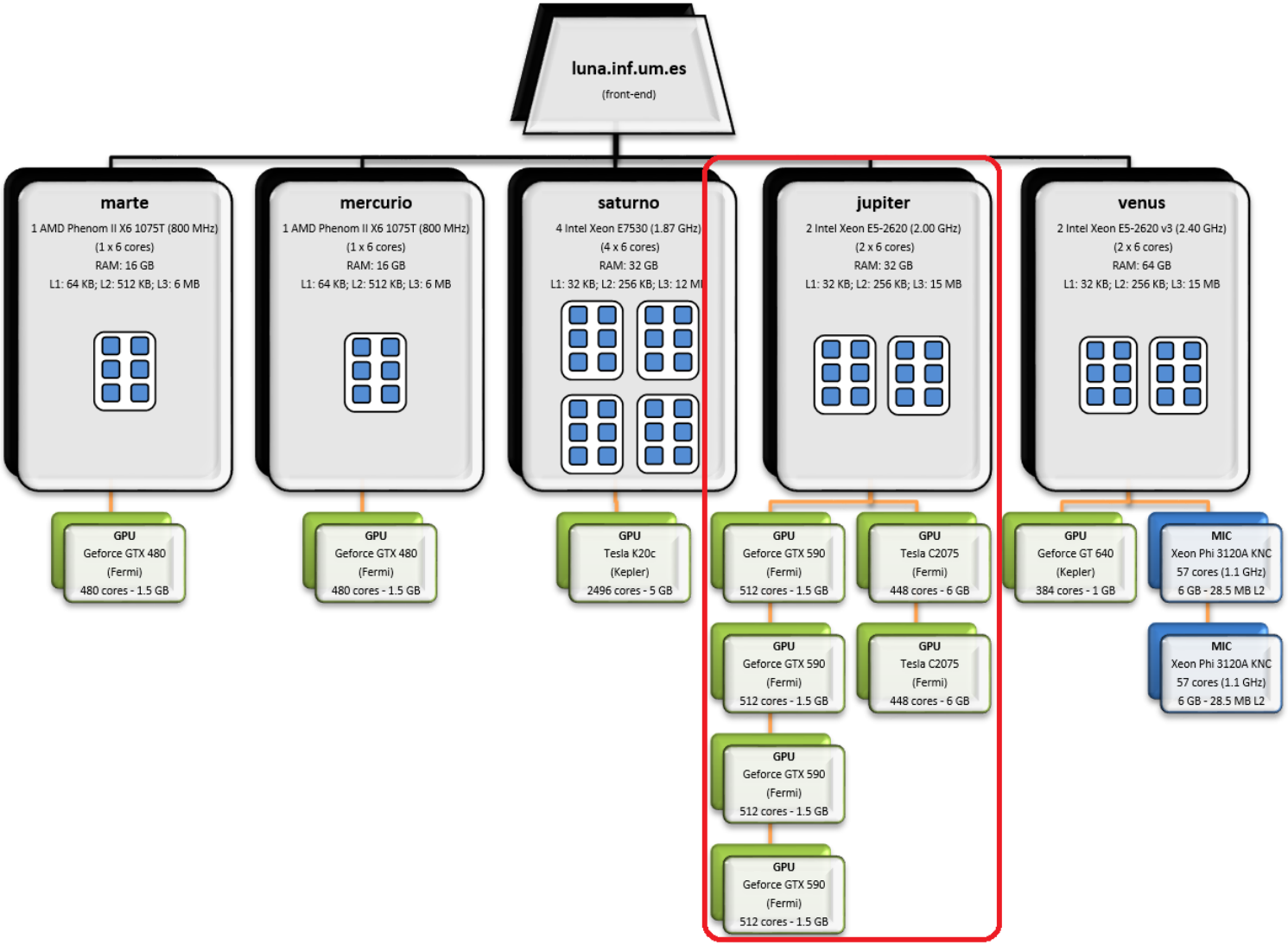
Which nb range?

- In principle any.
- An arbitrary set is considered for nb:
  $\{208, 256, 288, 320, 384, 448, 512, 576\}$

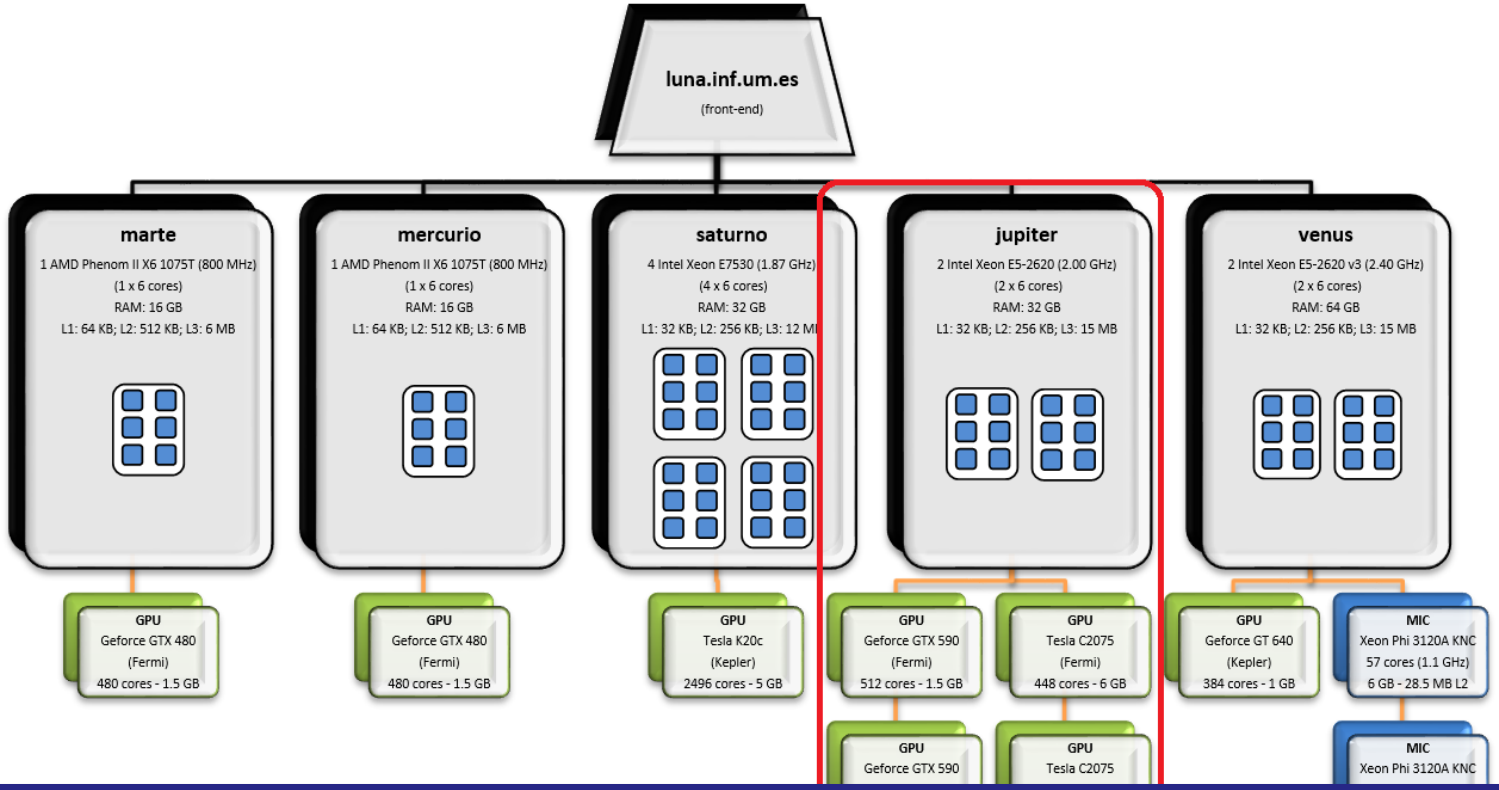
Overview

1. Motivation
2. Application Case
3. Search of the AP values
4. Selection of Computing Units
5. Selection of Scheduling Policy
6. Ongoing Work
7. Conclusions

Heterogeneous Platform



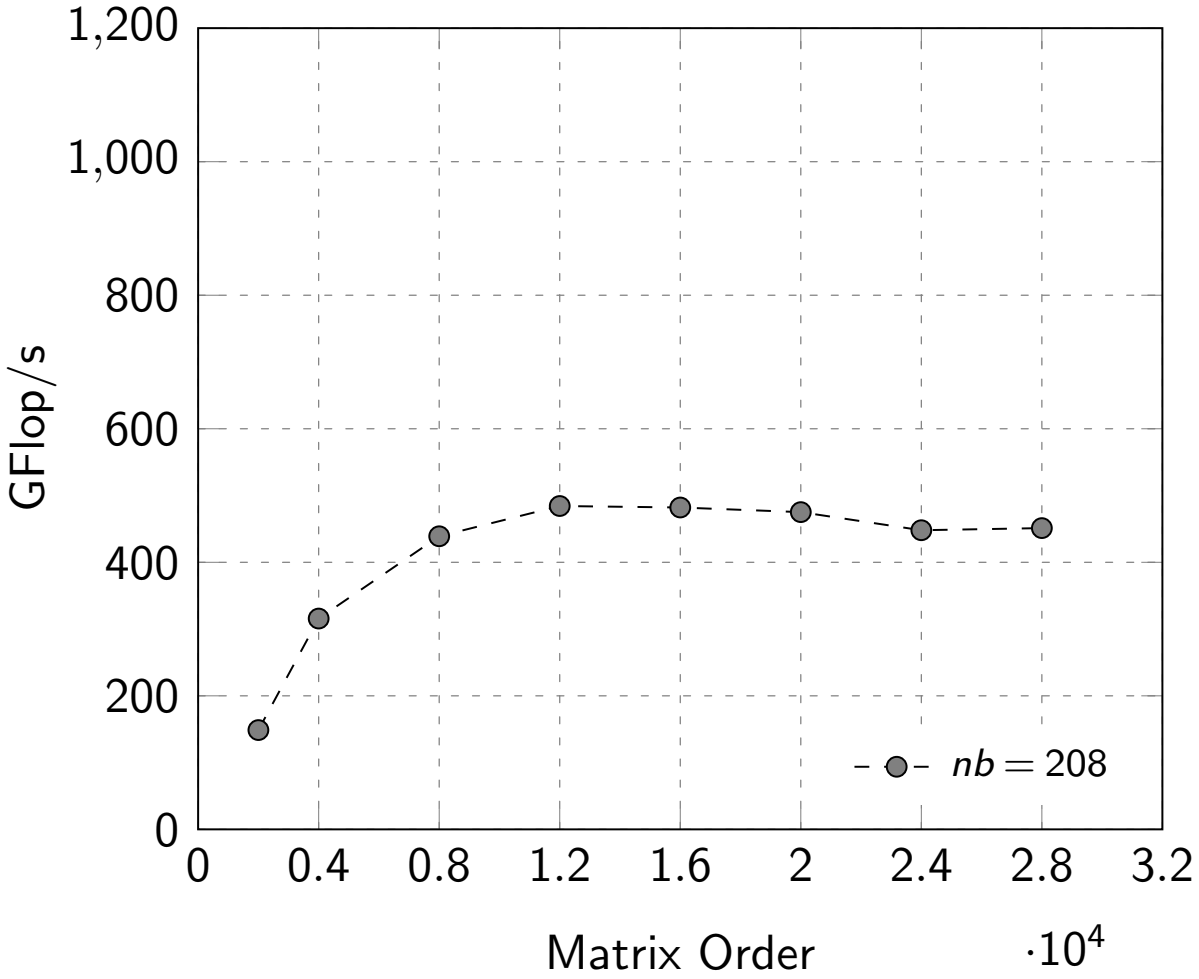
Heterogeneous Platform



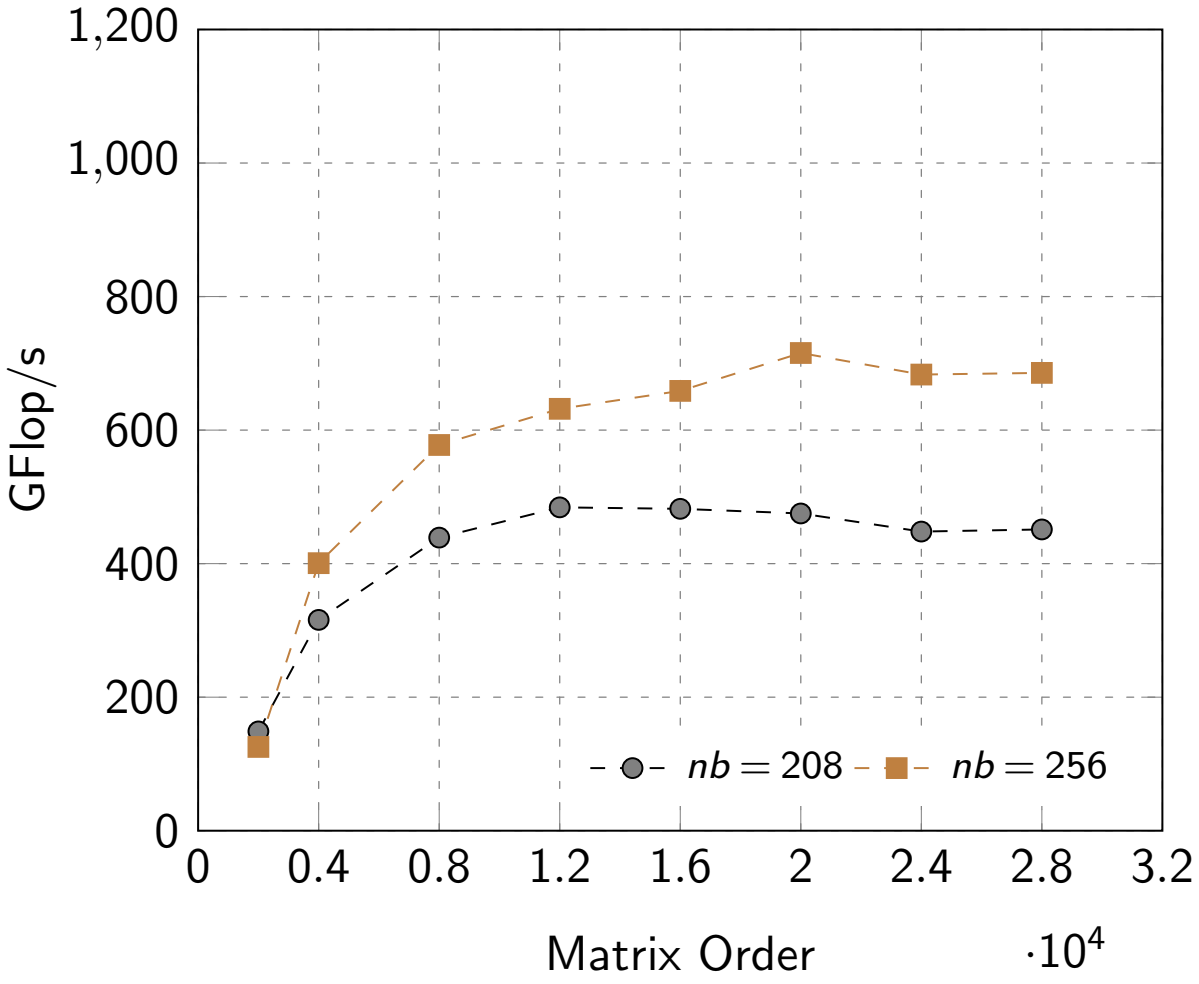
Jupiter

- 2 Intel Xeon E5-2620 Hexa-Core (12 CPU Cores).
- 4 GPU NVIDIA GeForce GTX590.
- 2 GPU NVIDIA Tesla C2075.

S1: Empirical+Exhaustive (Cholesky)

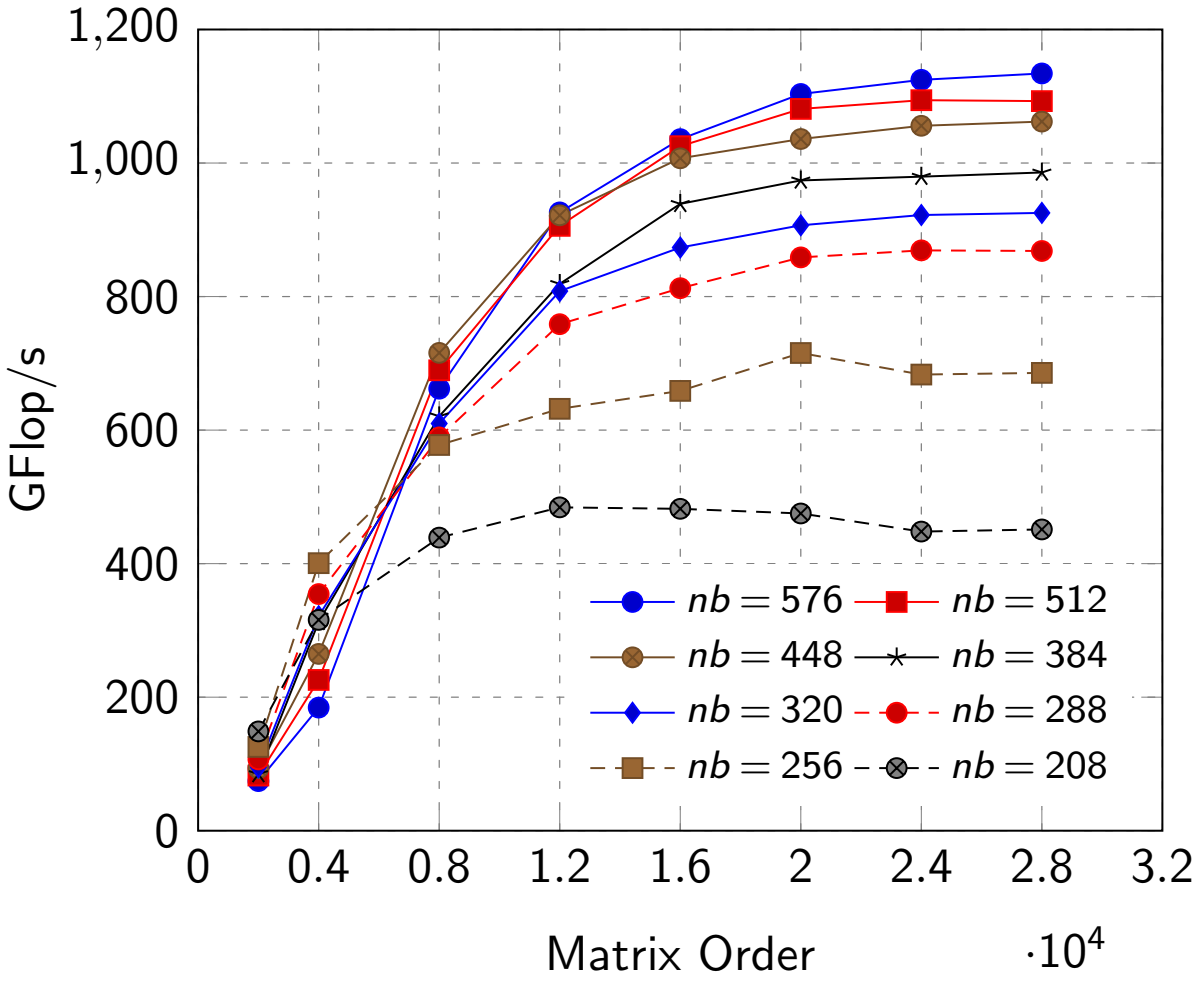


S1: Empirical+Exhaustive (Cholesky)



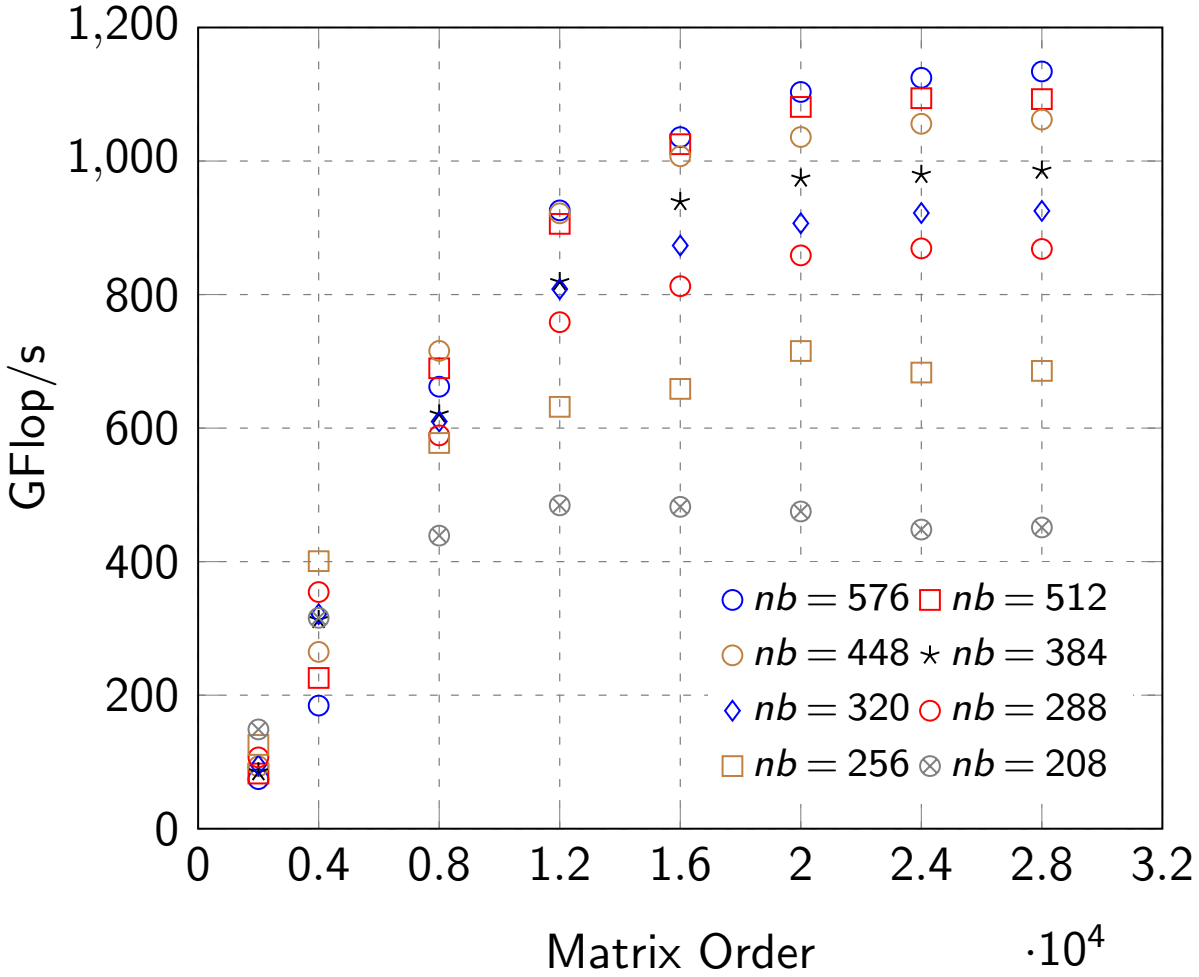
Performance depends on the value of the Tile Size

S1: Empirical+Exhaustive (Cholesky)



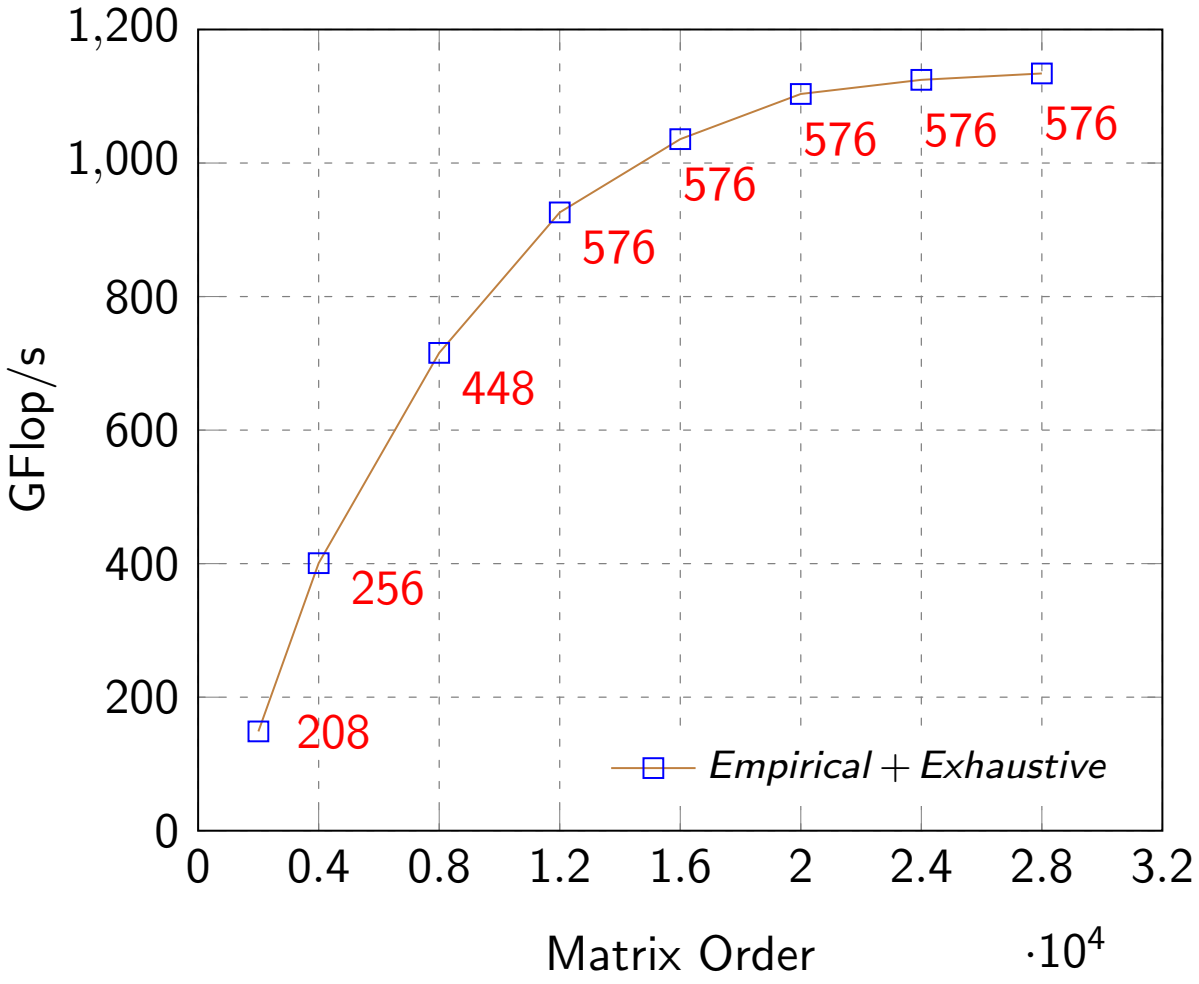
No. Experiments: 8 problem sizes * 8 tile sizes = 64

S1: Empirical+Exhaustive (Cholesky)



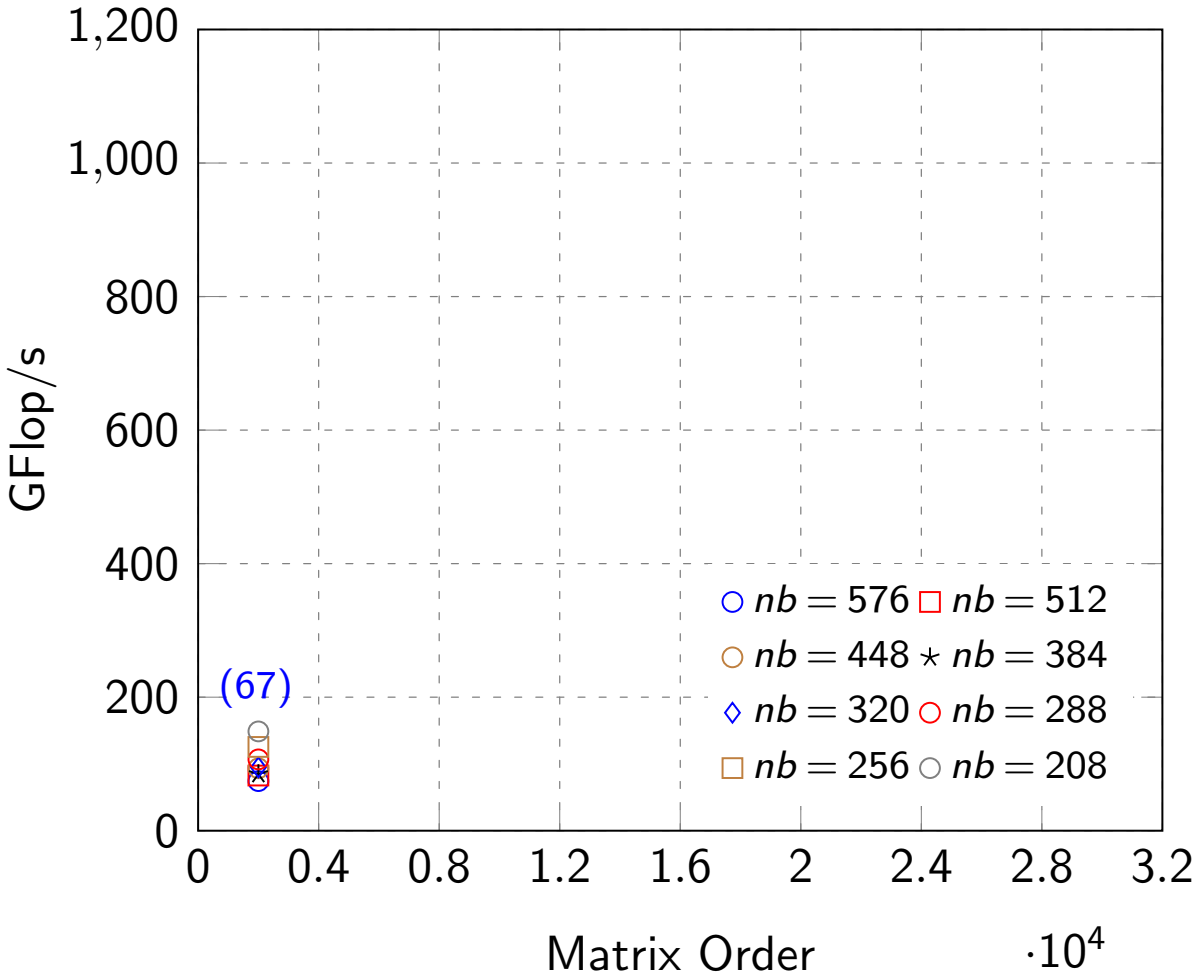
Which *nb* among these 8 values for a *n* given at runtime?

S1: Empirical+Exhaustive (Cholesky)



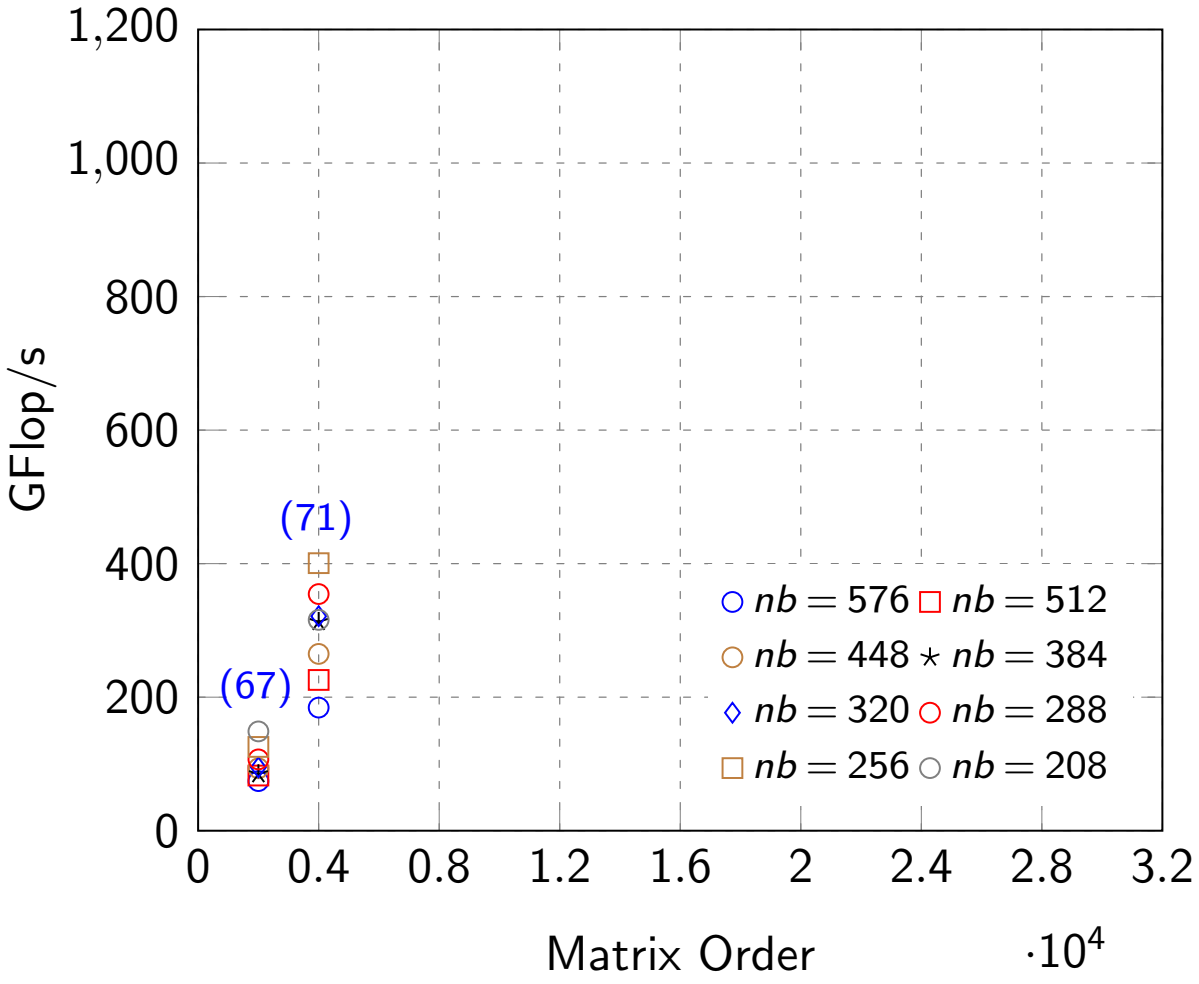
nb with a higher performance for each problem size *n*

S1: Empirical+Exhaustive (Cholesky)



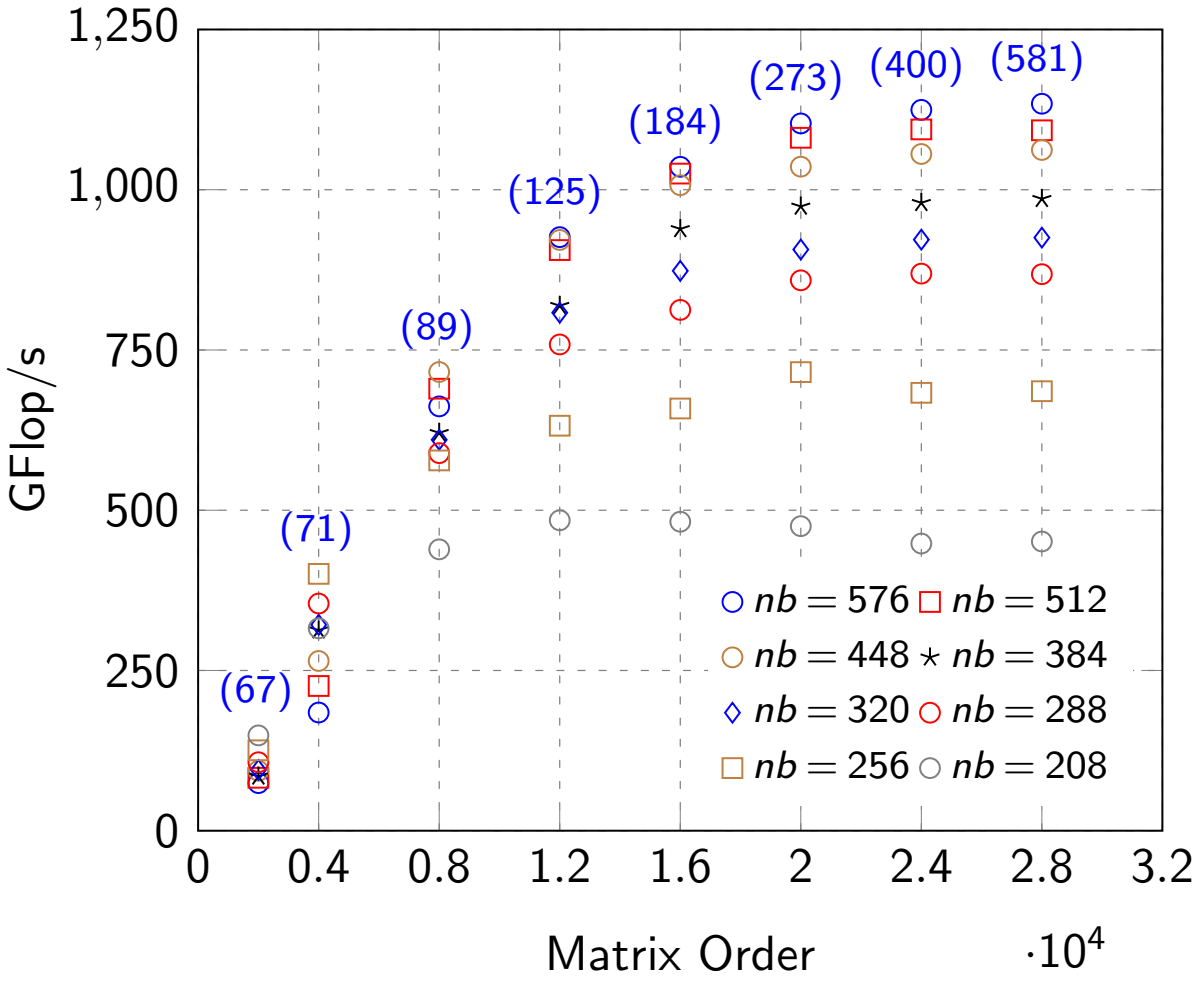
Experimental Time: 67 seconds

S1: Empirical+Exhaustive (Cholesky)



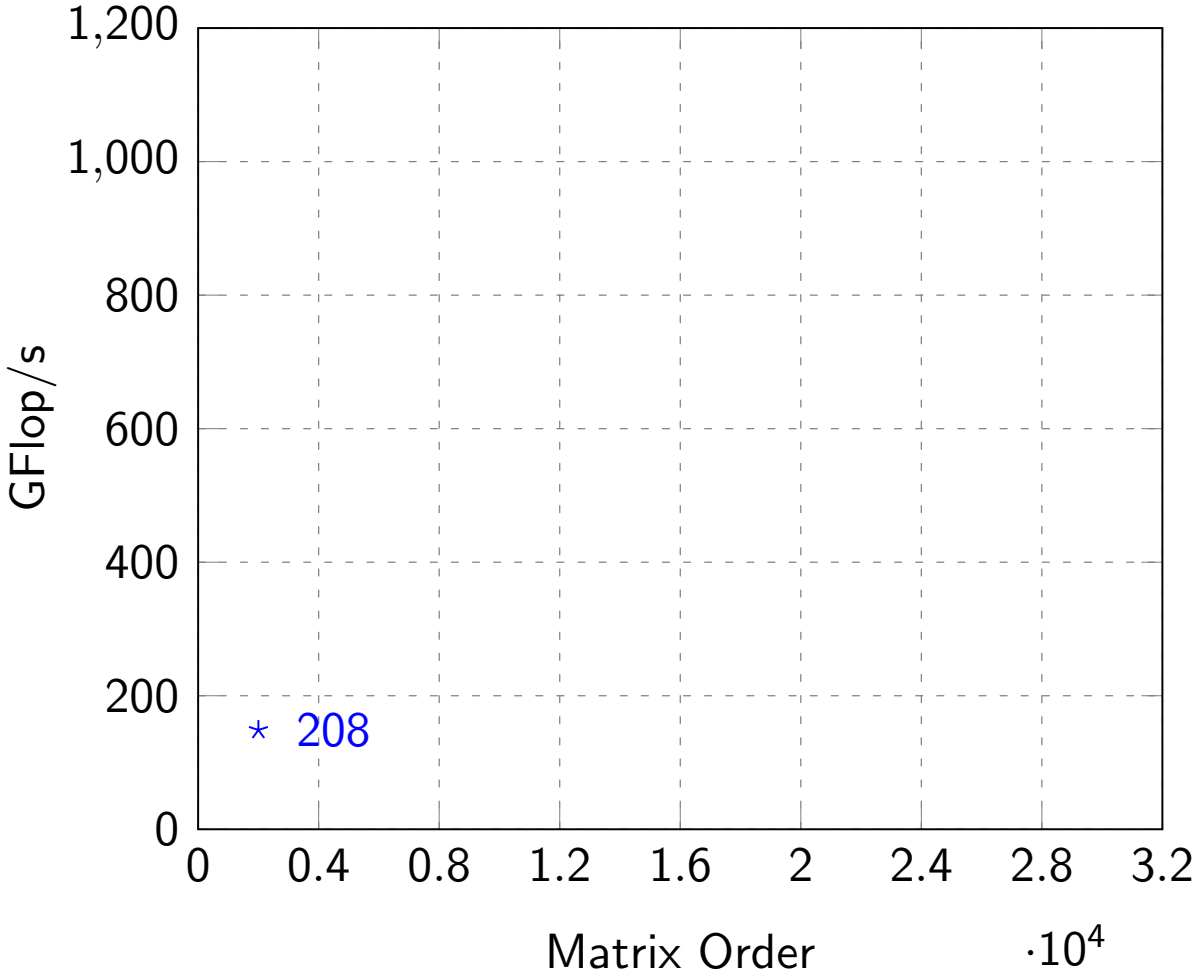
Experimental Time: 138 seconds

S1: Empirical+Exhaustive (Cholesky)

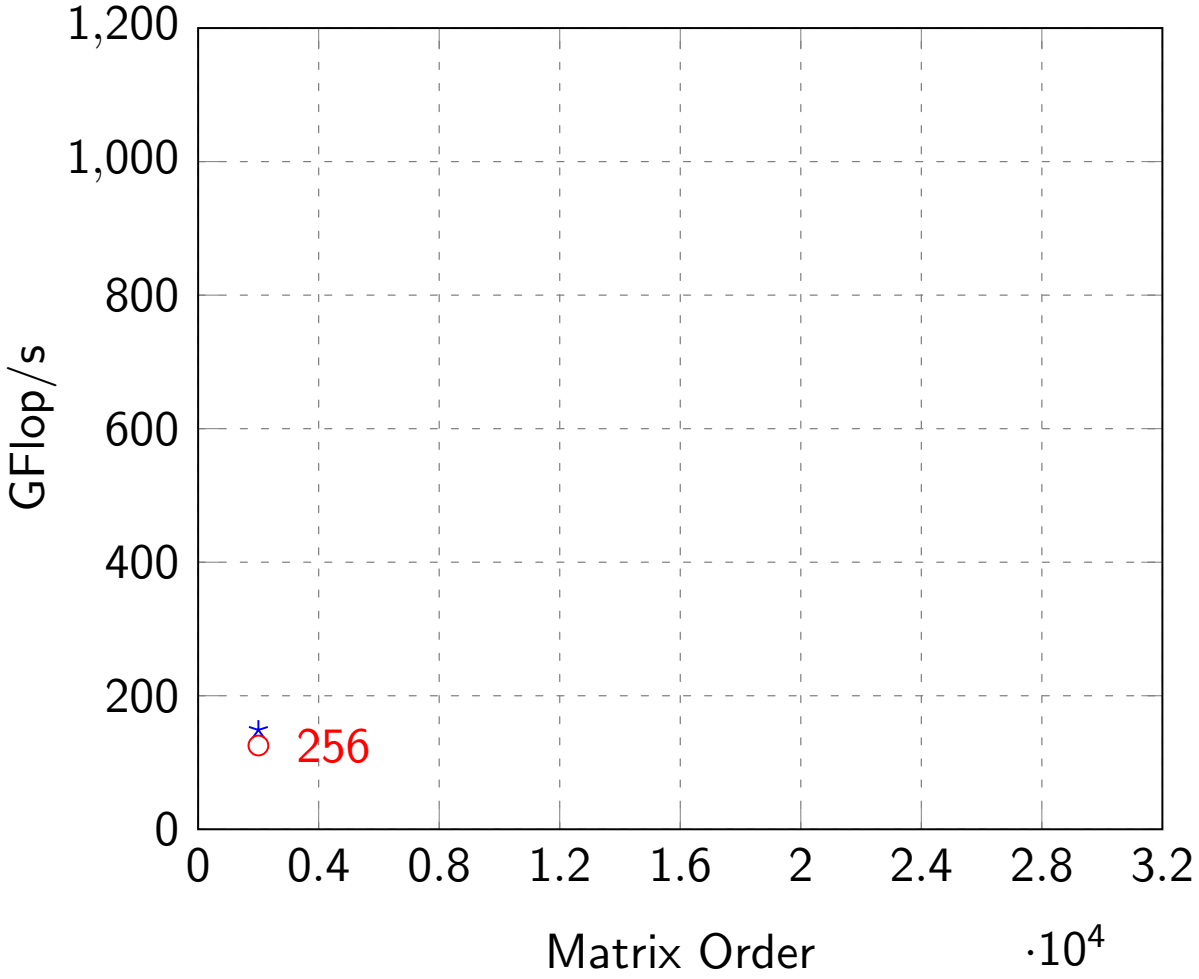


Experimental Time: 30 minutes

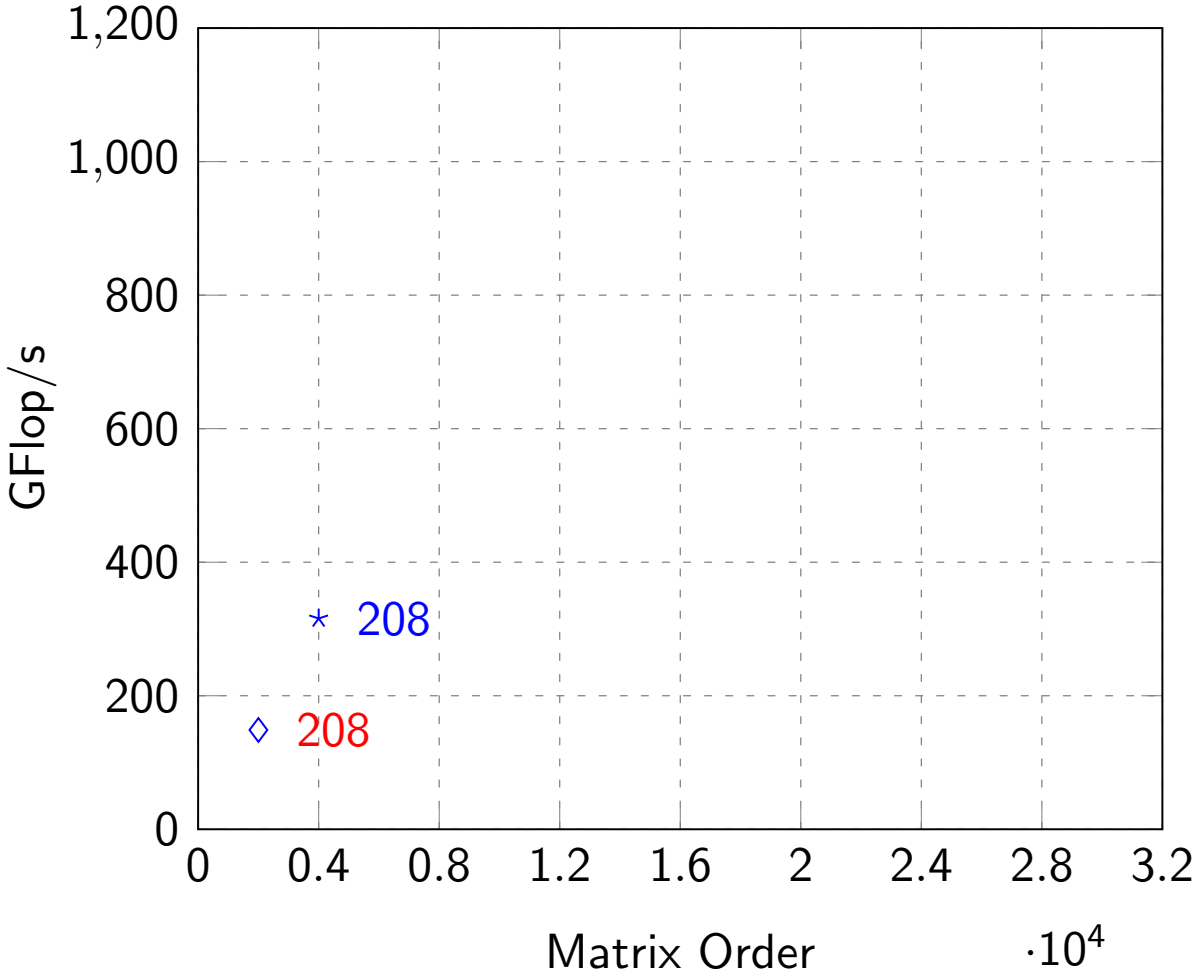
S2: Empirical+Pruned (Cholesky)



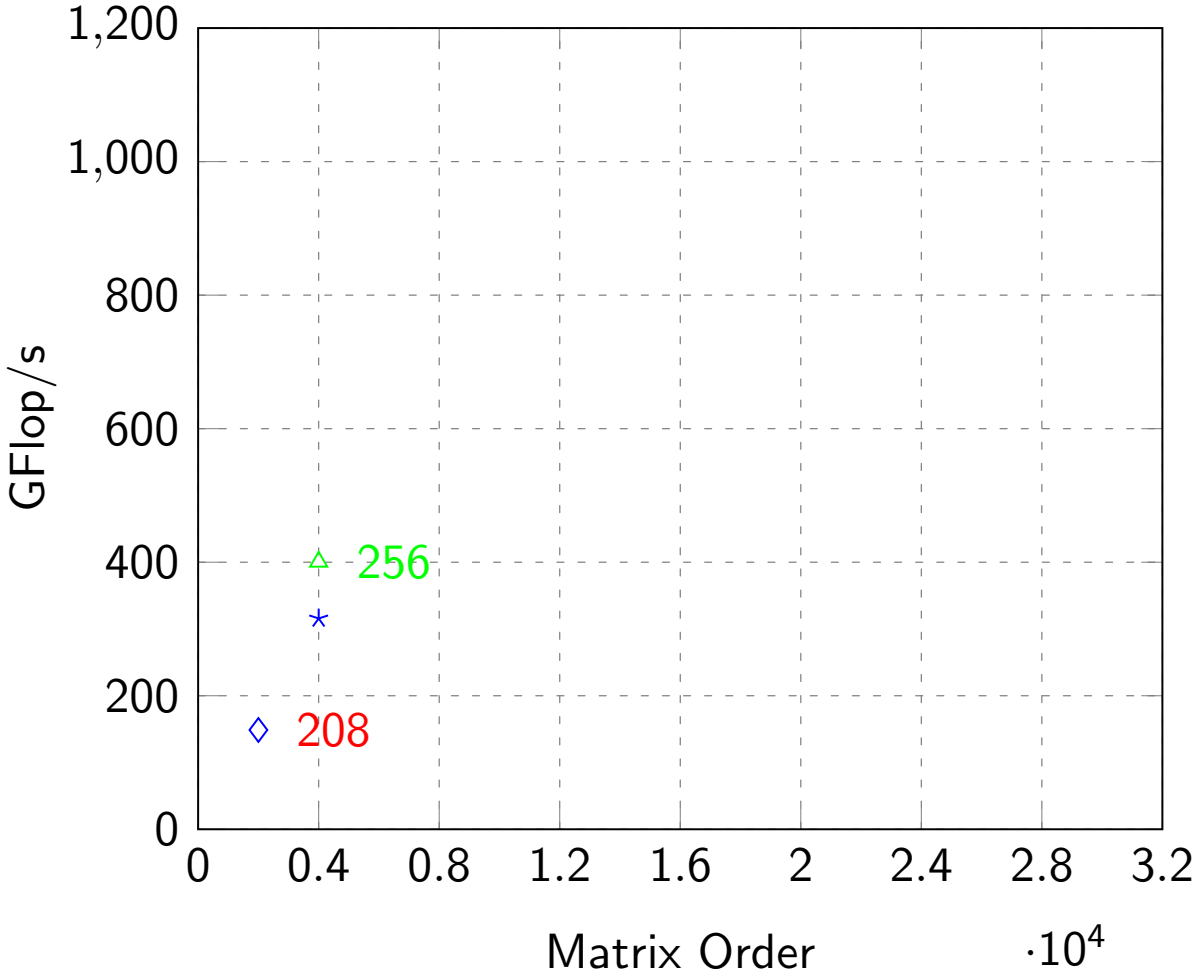
S2: Empirical+Pruned (Cholesky)



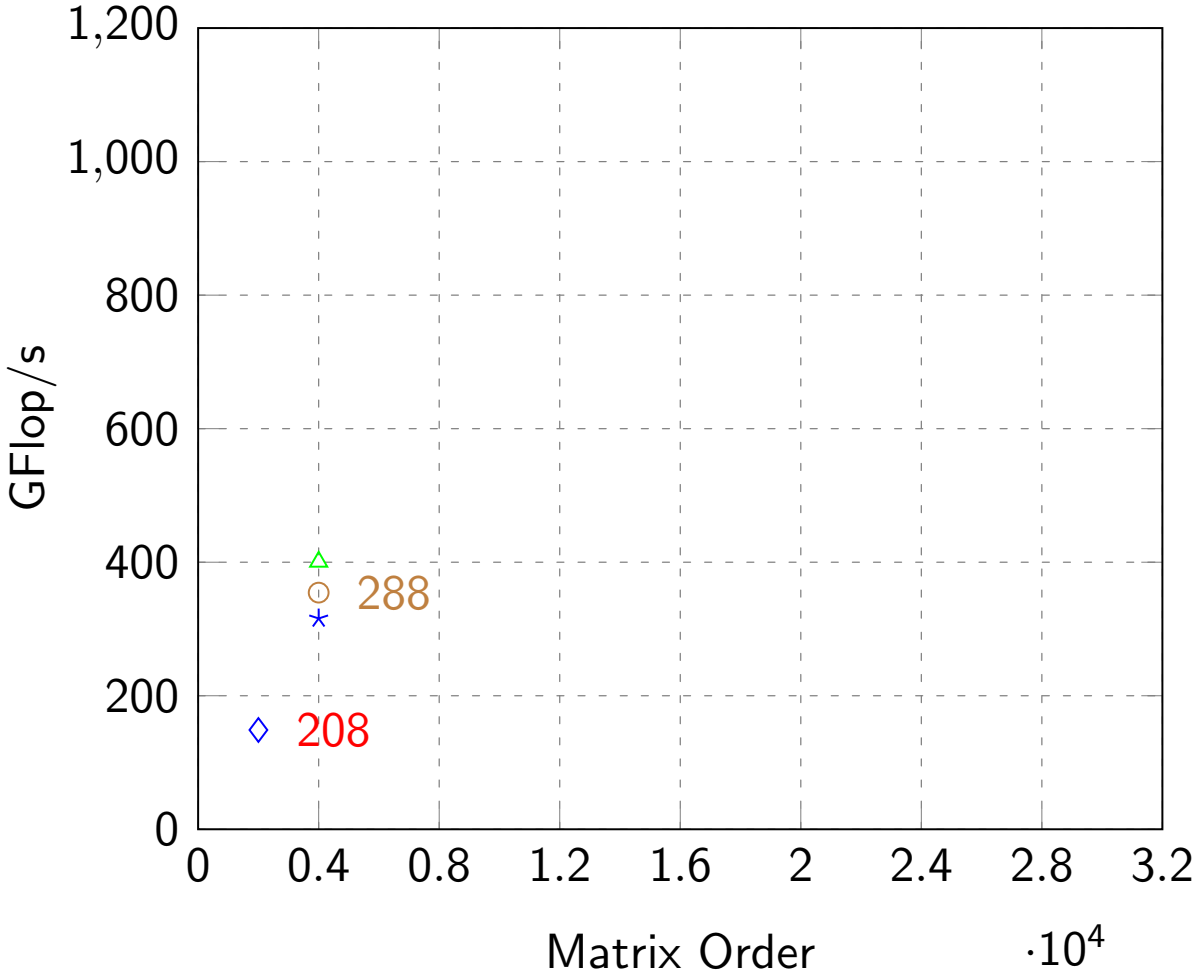
S2: Empirical+Pruned (Cholesky)



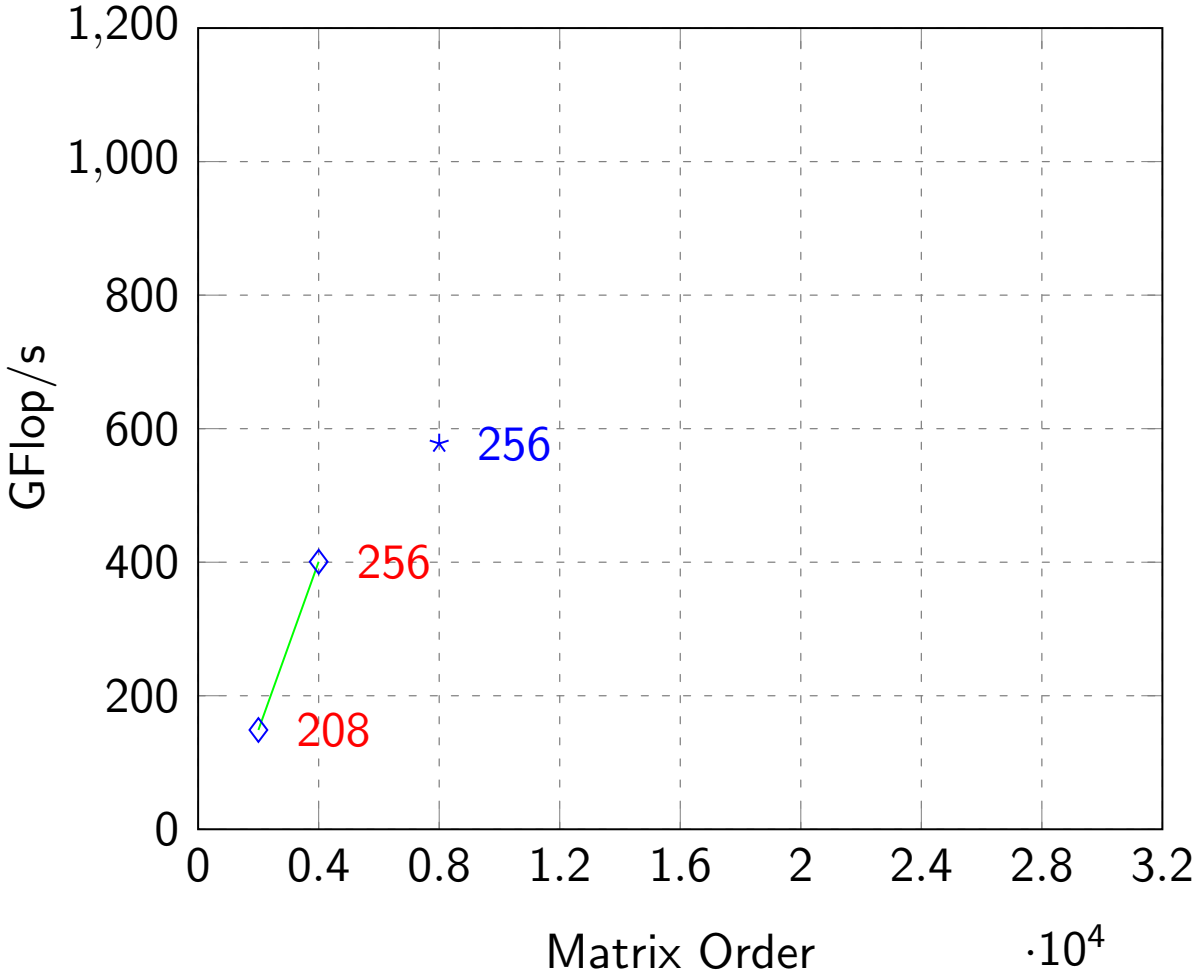
S2: Empirical+Pruned (Cholesky)



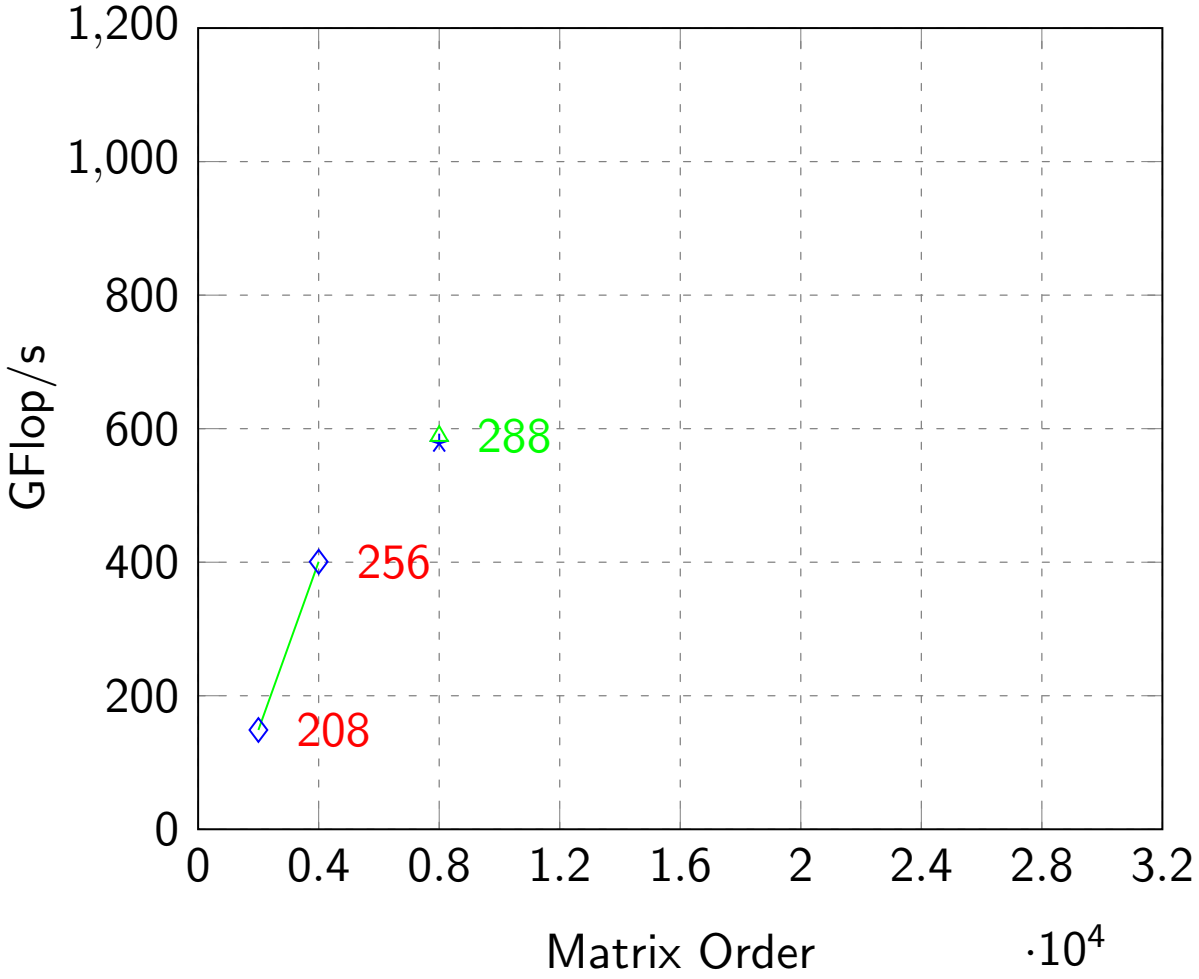
S2: Empirical+Pruned (Cholesky)



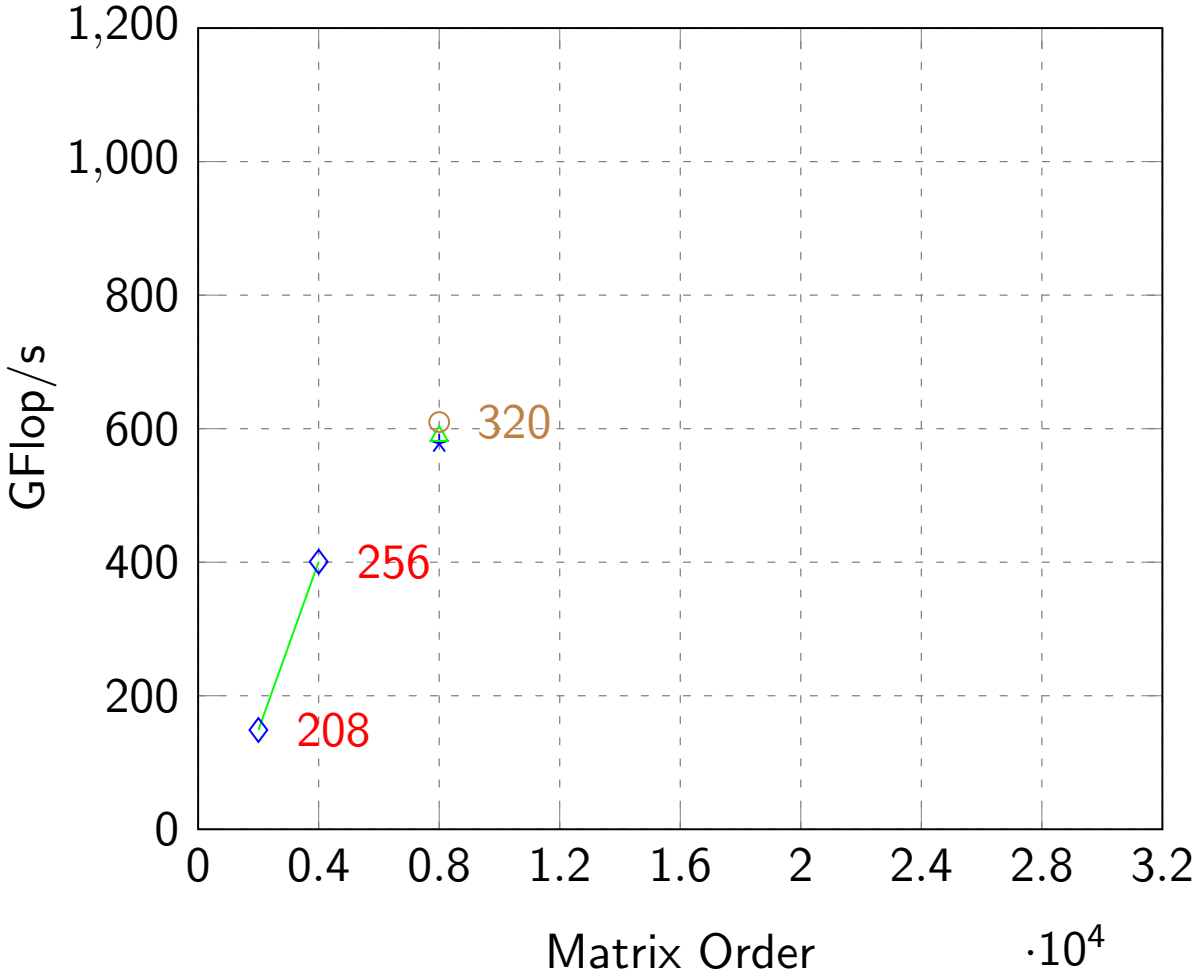
S2: Empirical+Pruned (Cholesky)



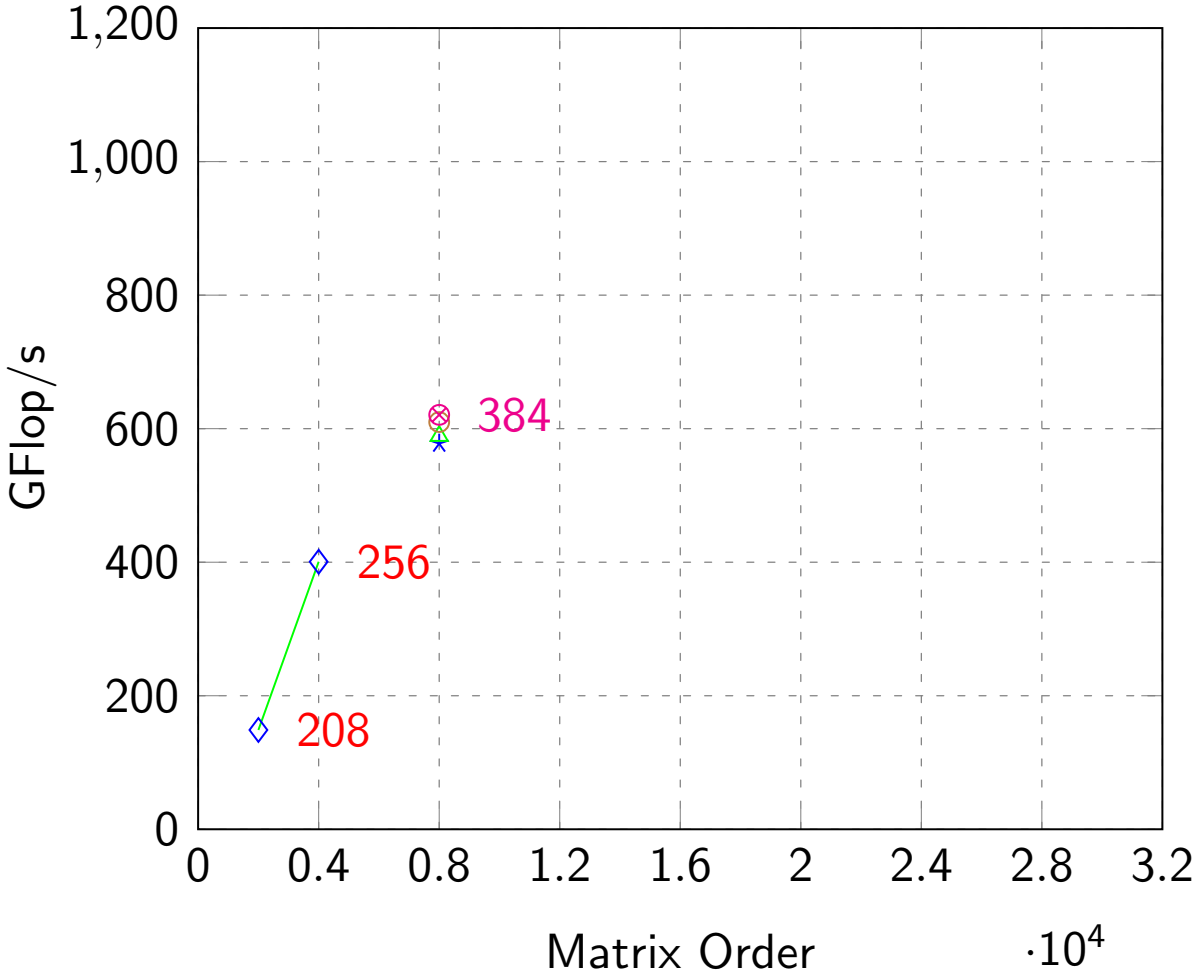
S2: Empirical+Pruned (Cholesky)



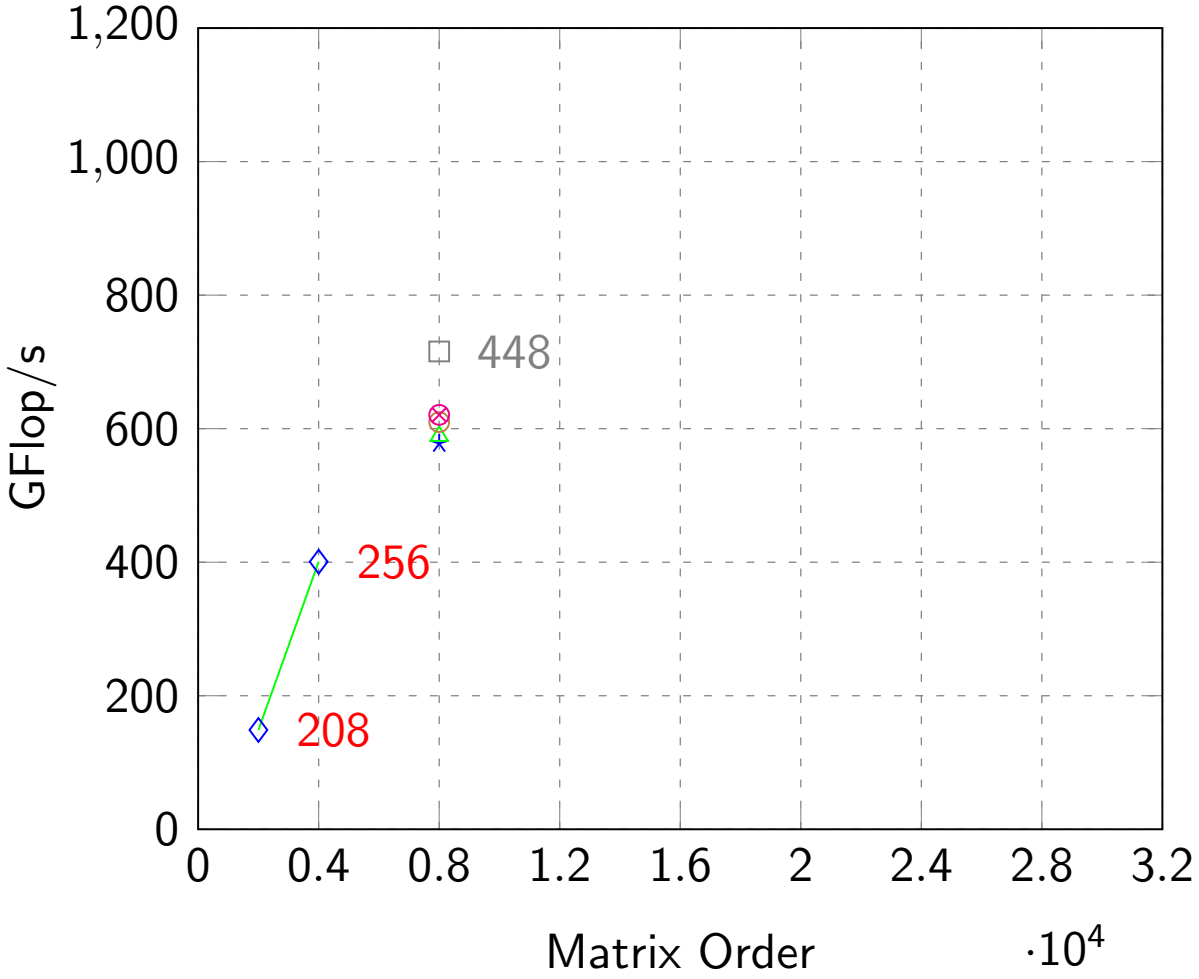
S2: Empirical+Pruned (Cholesky)



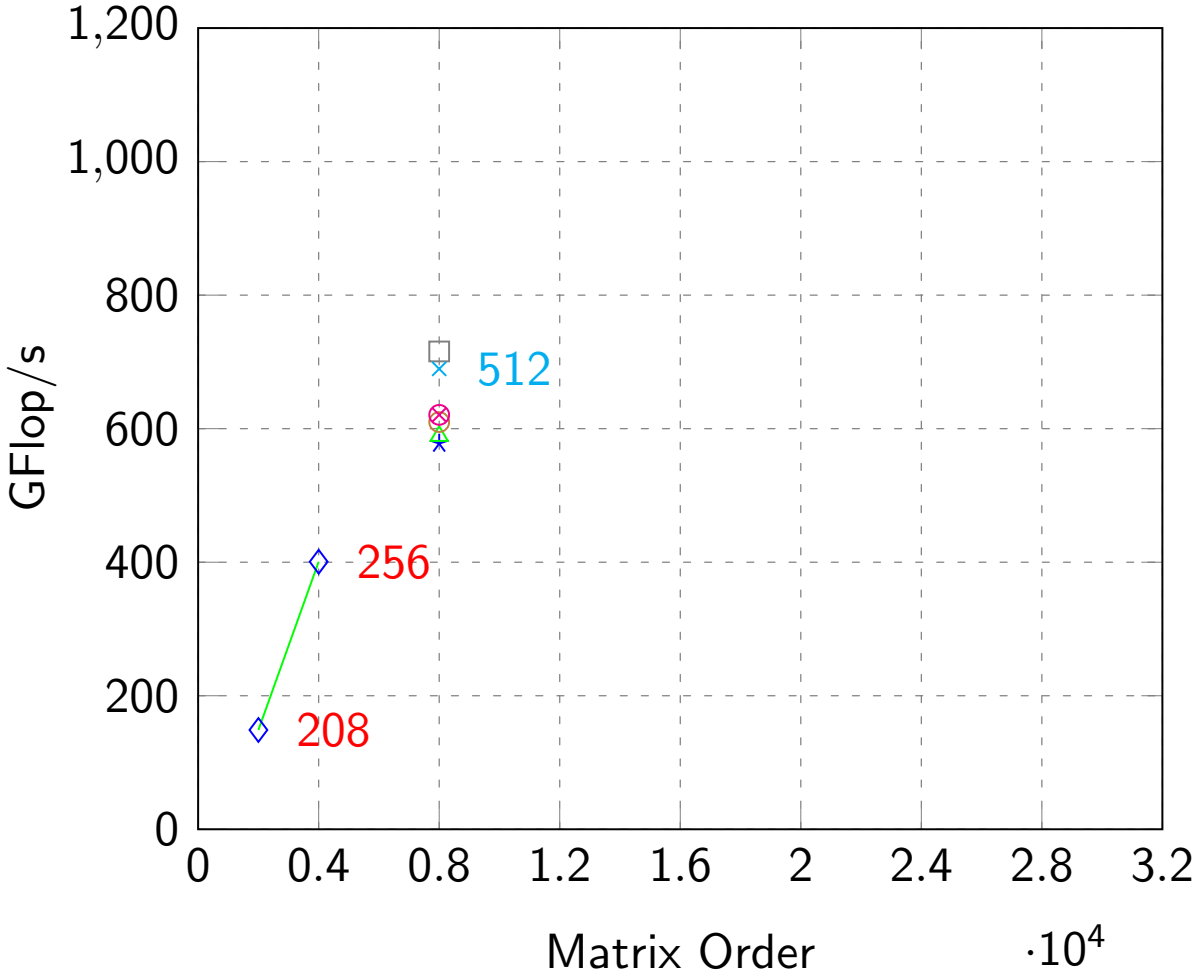
S2: Empirical+Pruned (Cholesky)



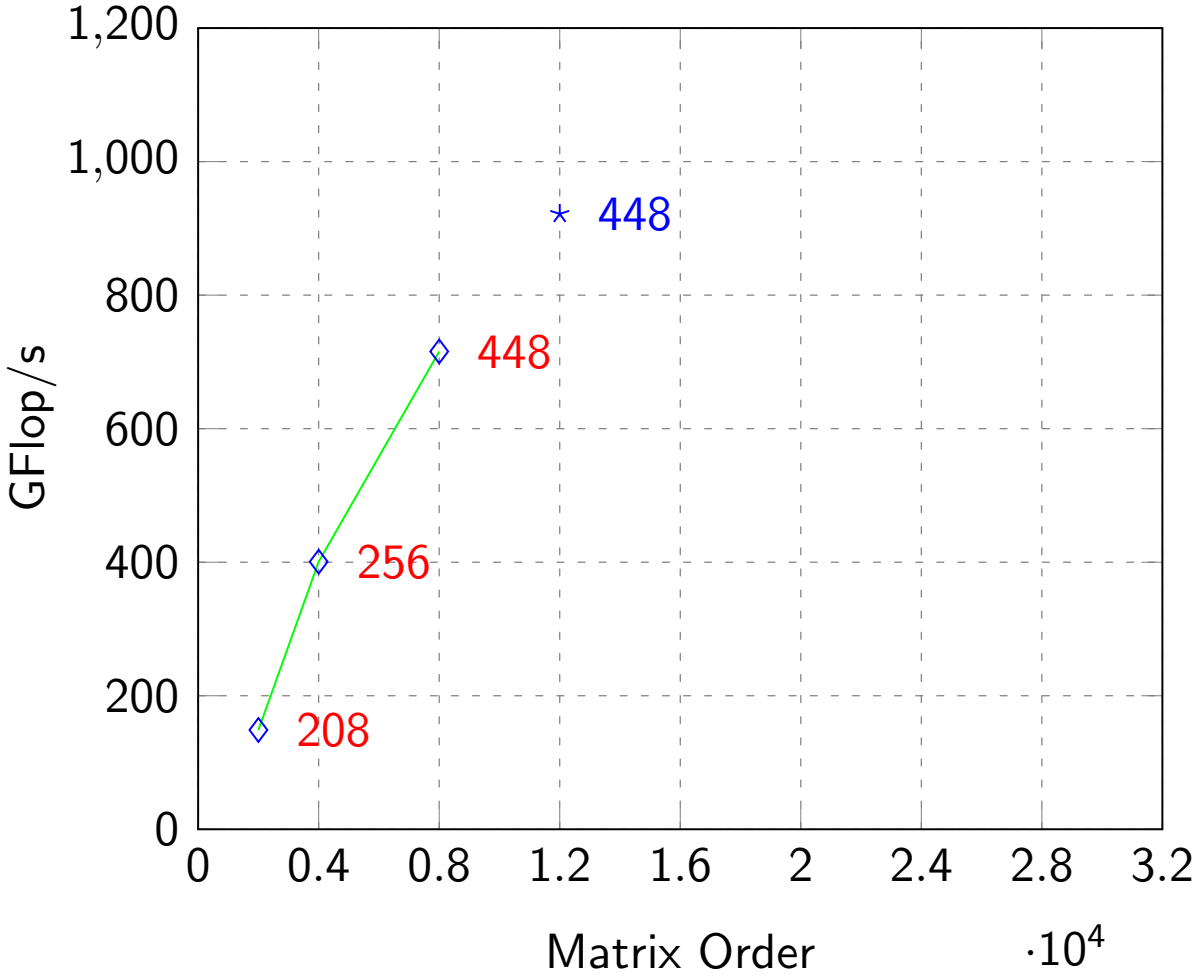
S2: Empirical+Pruned (Cholesky)



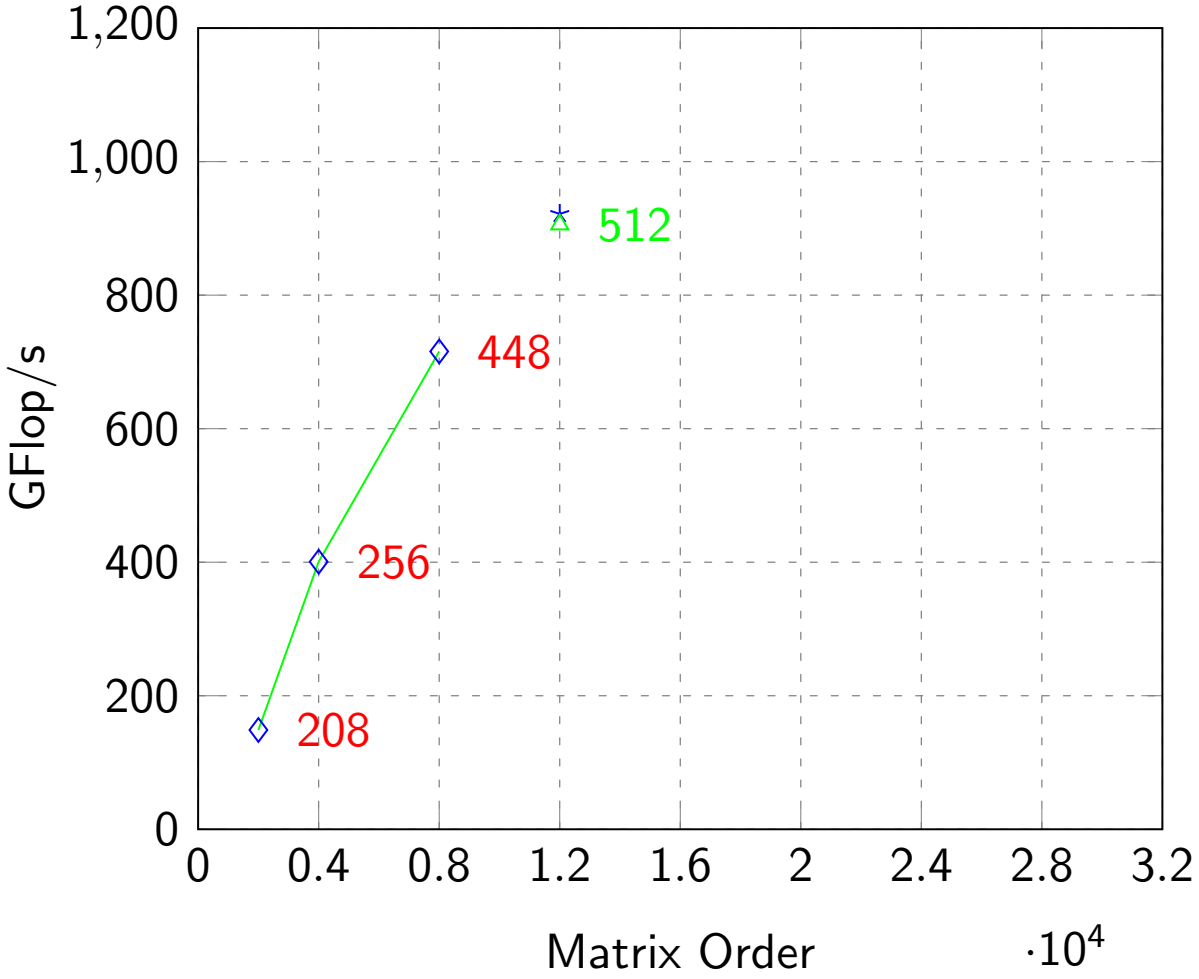
S2: Empirical+Pruned (Cholesky)



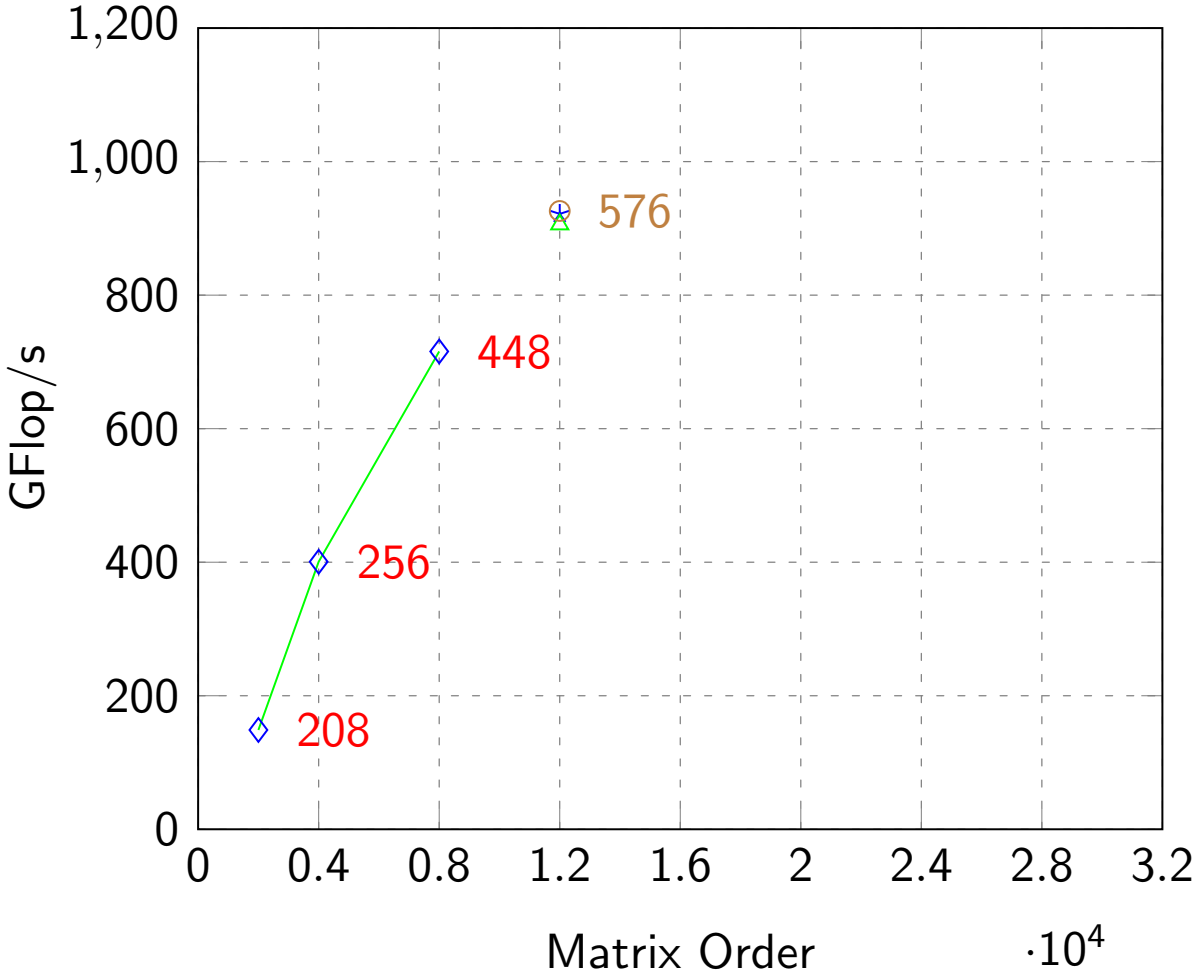
S2: Empirical+Pruned (Cholesky)



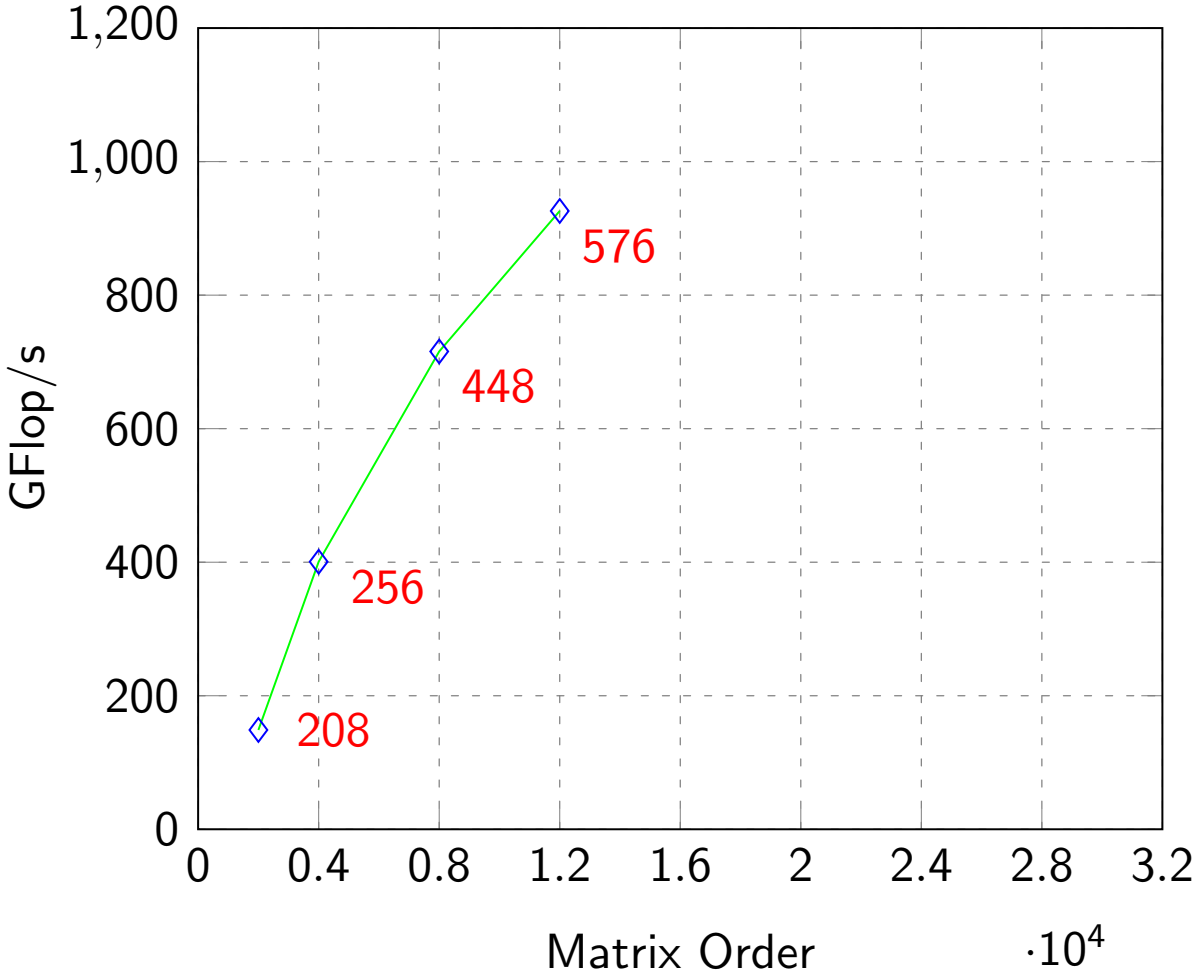
S2: Empirical+Pruned (Cholesky)



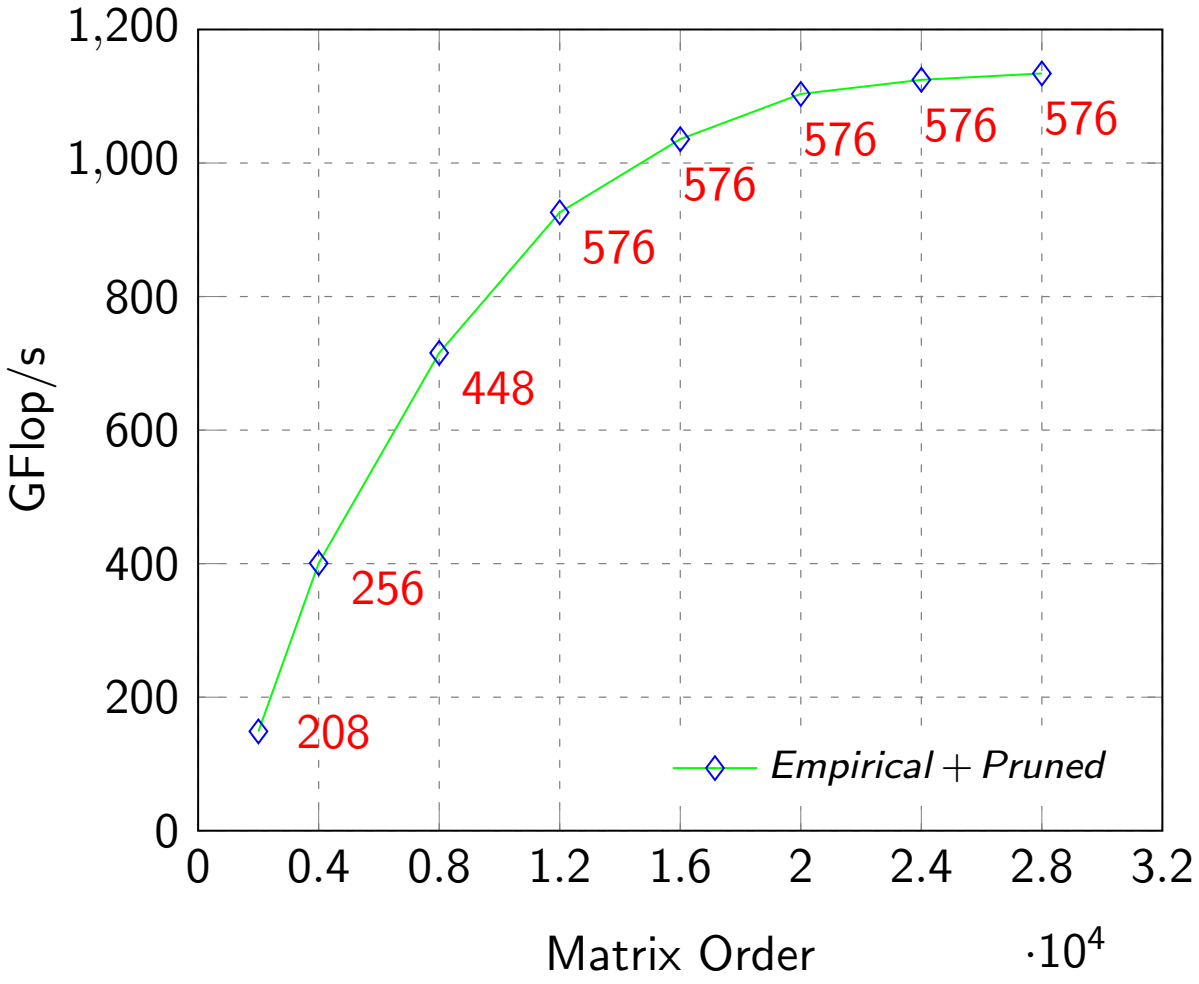
S2: Empirical+Pruned (Cholesky)



S2: Empirical+Pruned (Cholesky)



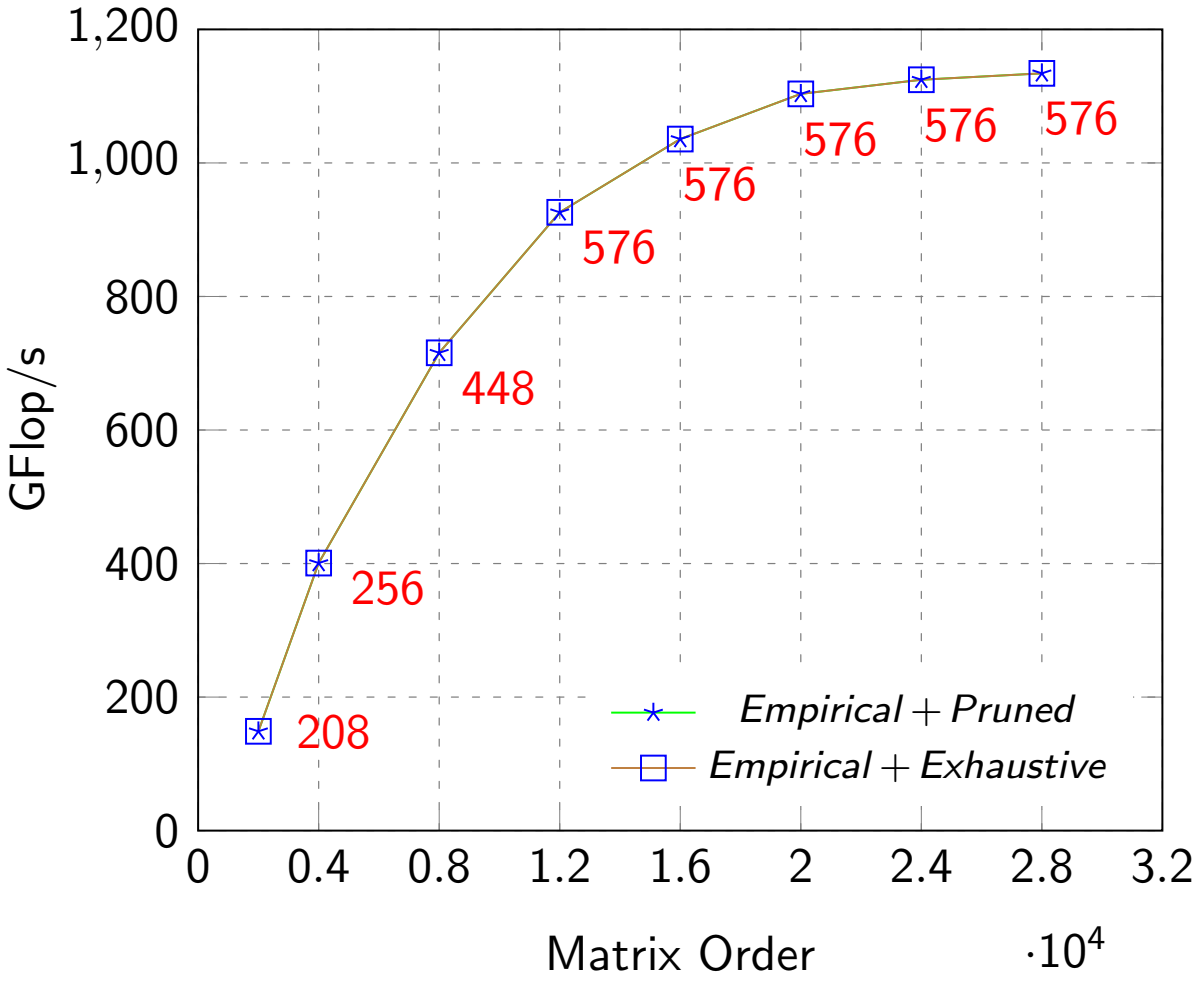
S2: Empirical+Pruned (Cholesky)



Experimental Time: 154 seconds

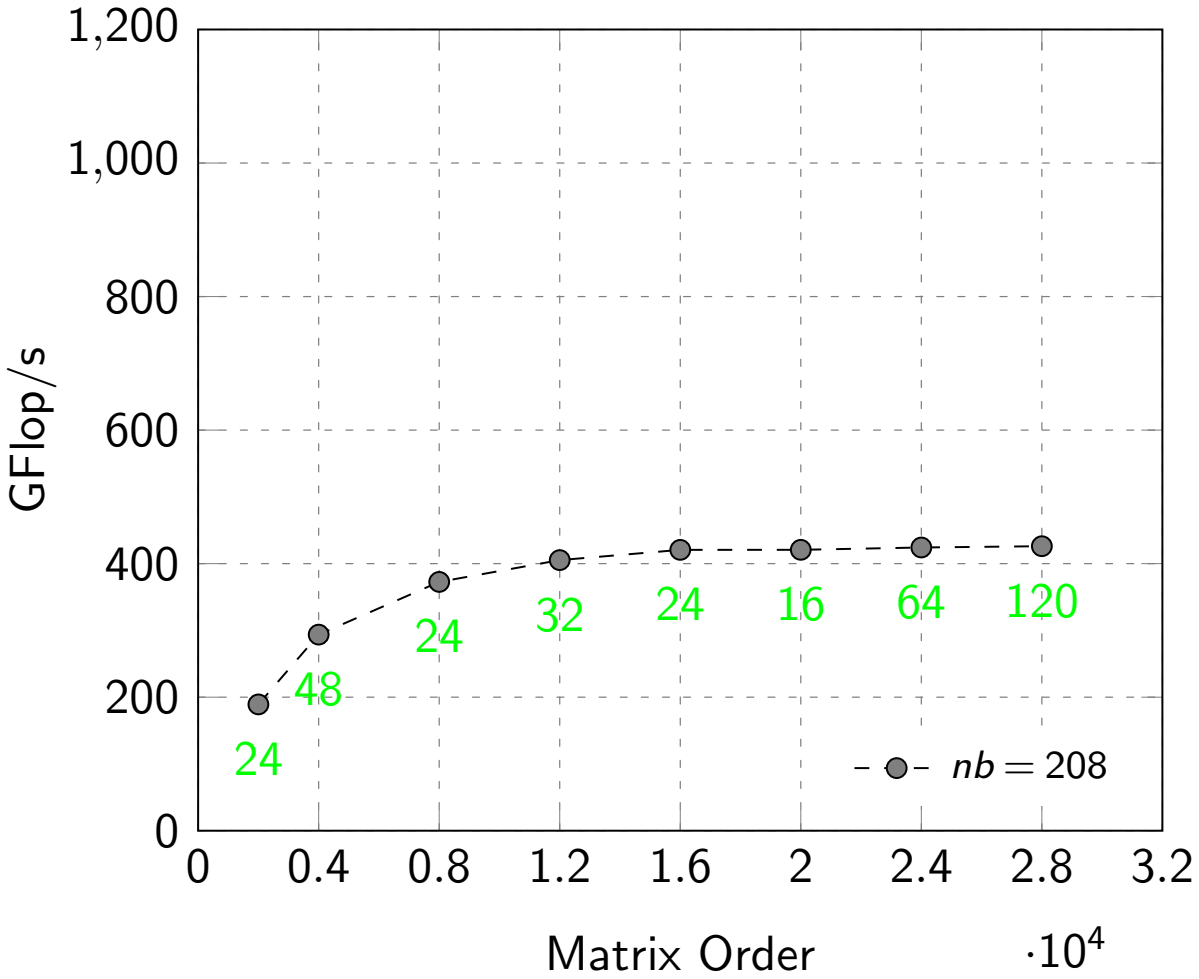
S1 & S2 (Cholesky)

Same *nb* value for each problem size *n*



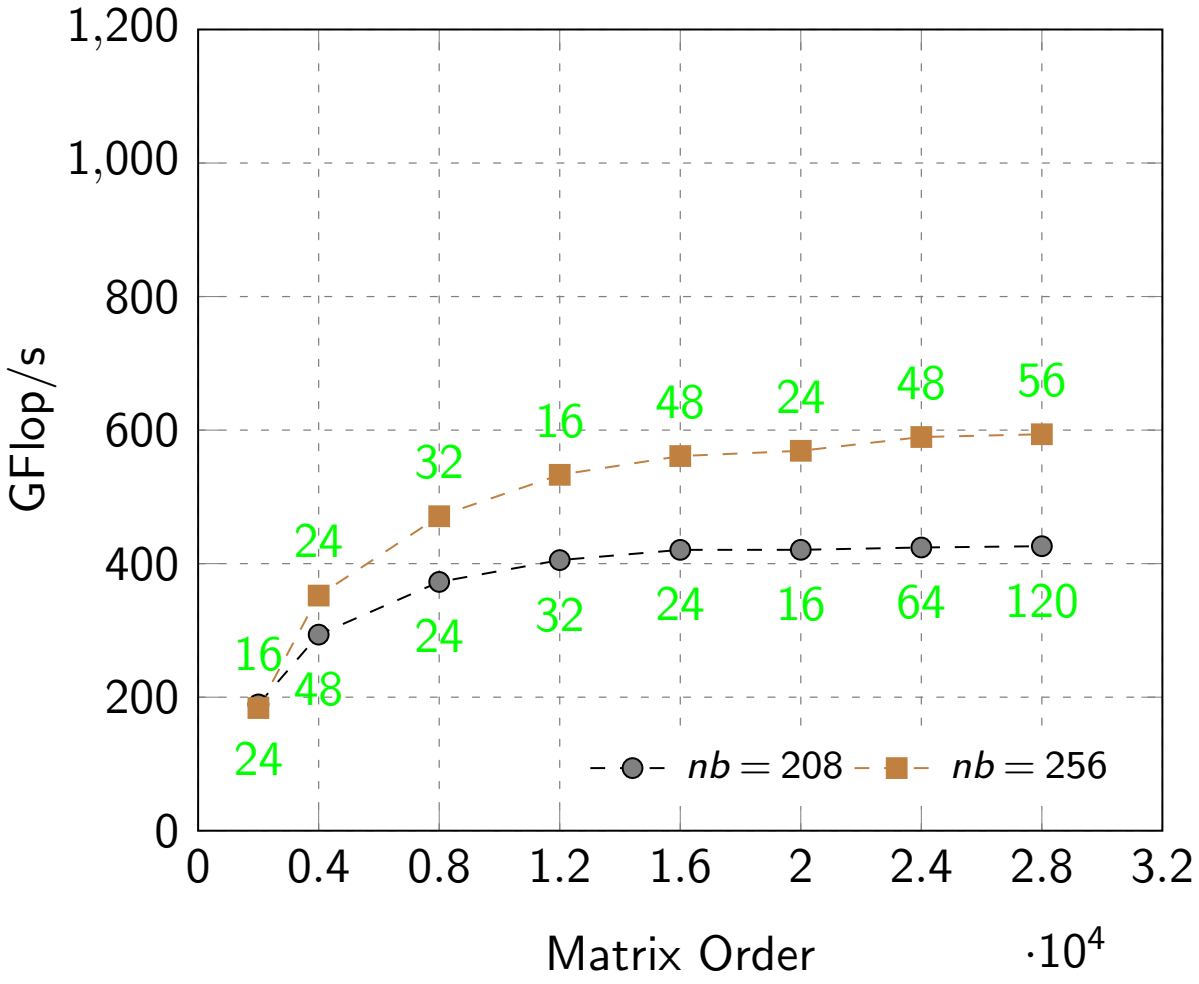
S1: 30 min; **S2**: 154 seconds

S1: Empirical+Exhaustive (LU)



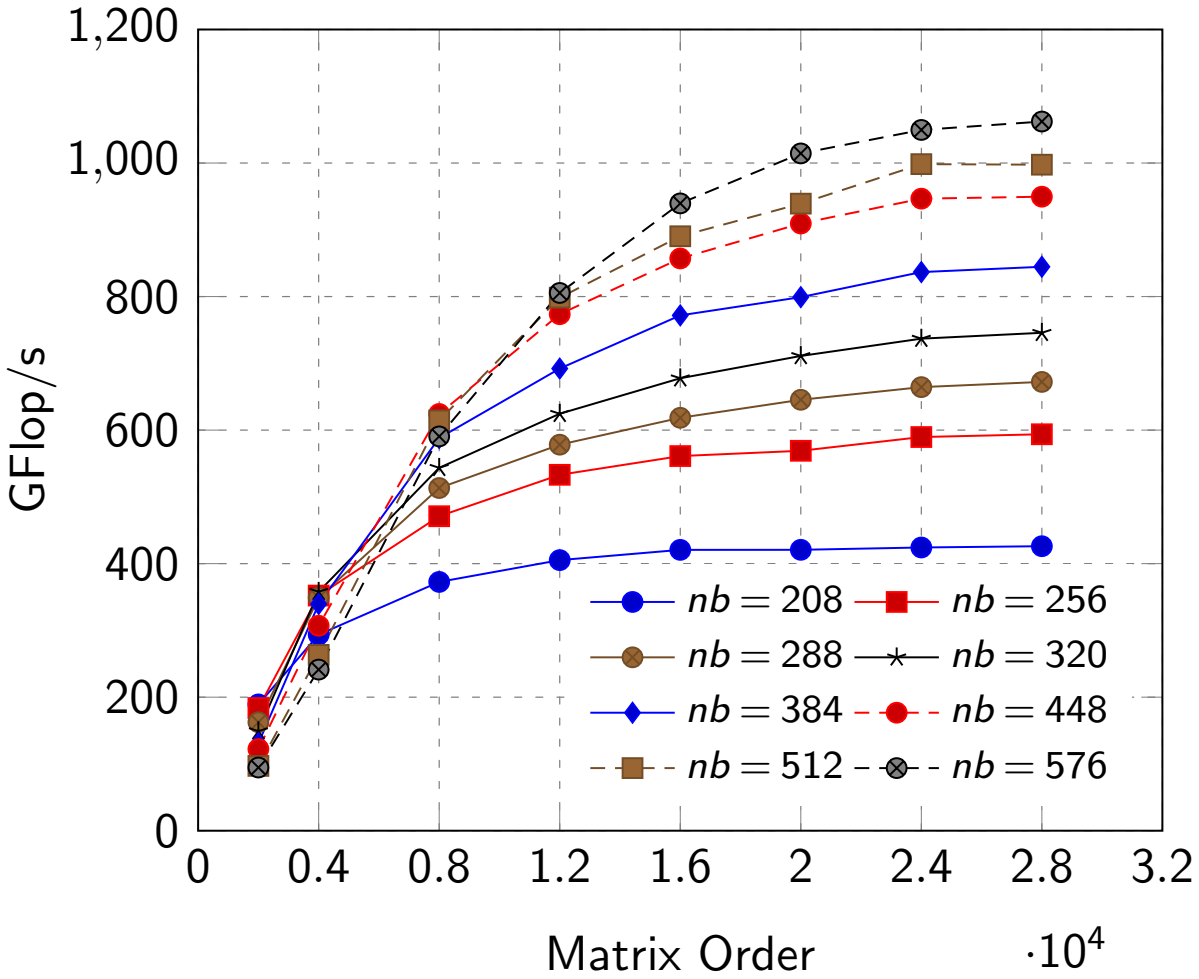
Best value for *ib* with current *nb* for each problem size *n*.

S1: Empirical+Exhaustive (LU)



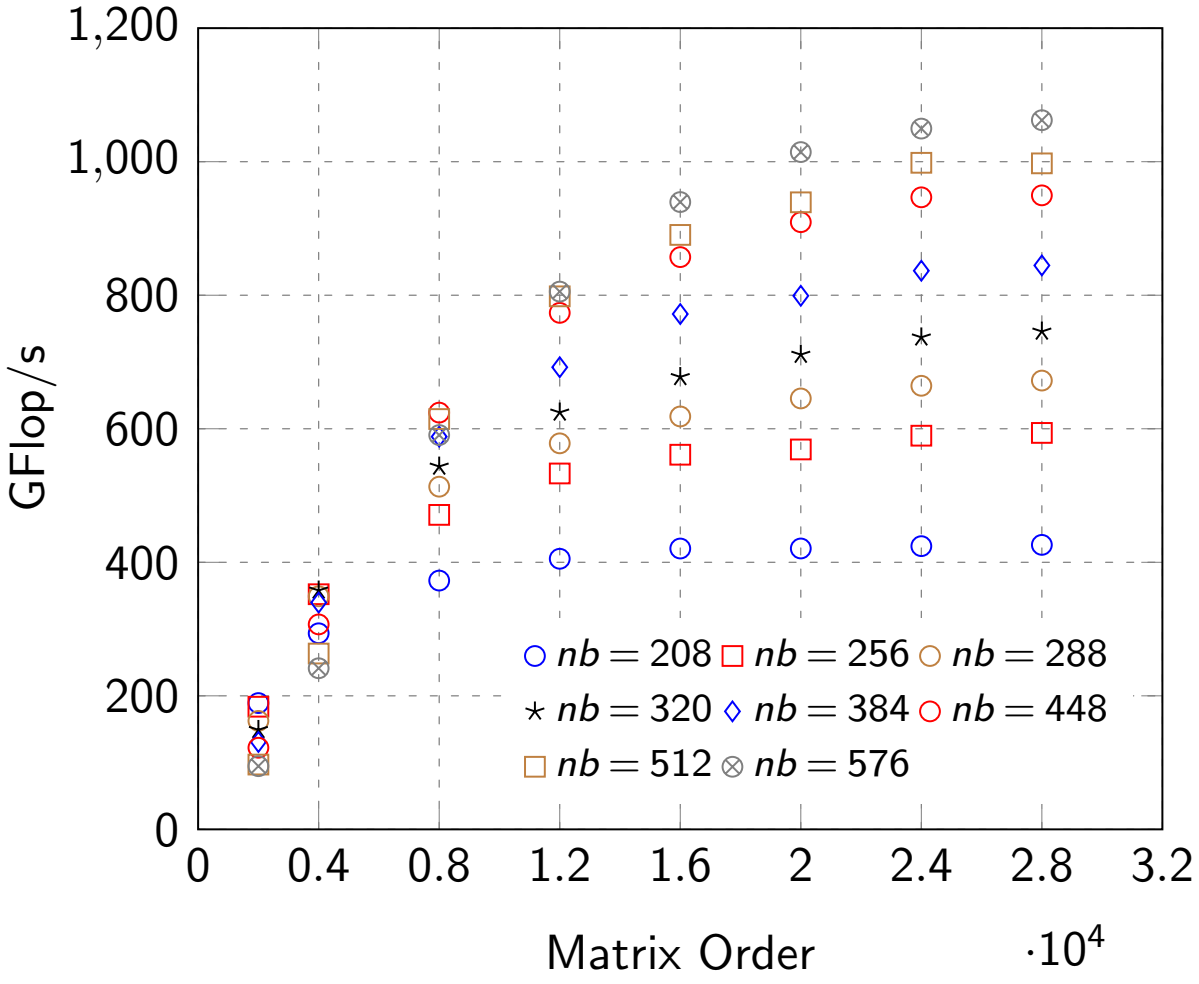
ib is incremented from 8 to *nb* in steps of 8.

S1: Empirical+Exhaustive (LU)



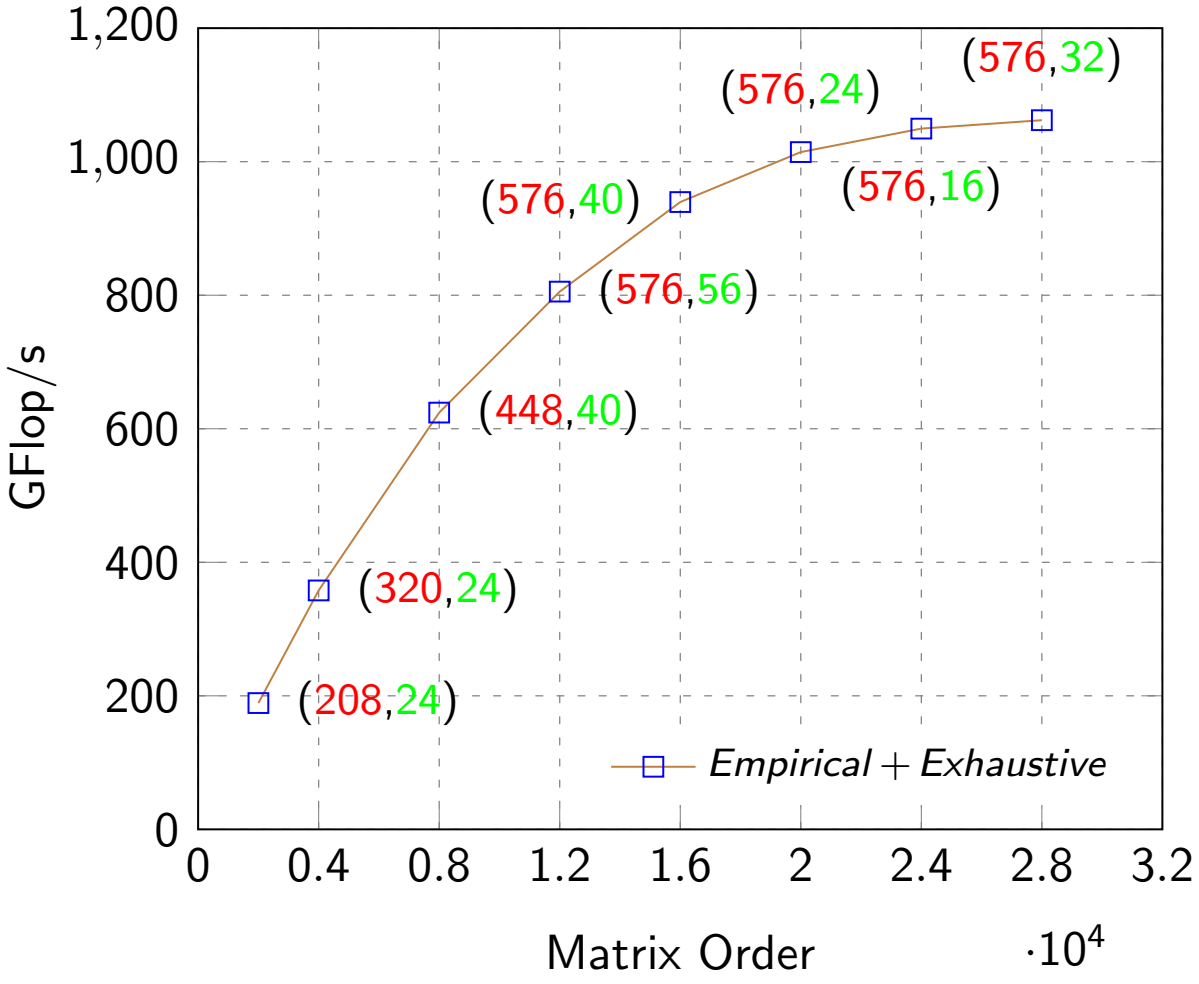
Performance significantly depends on the value of *nb* and *ib*

S1: Empirical+Exhaustive (LU)



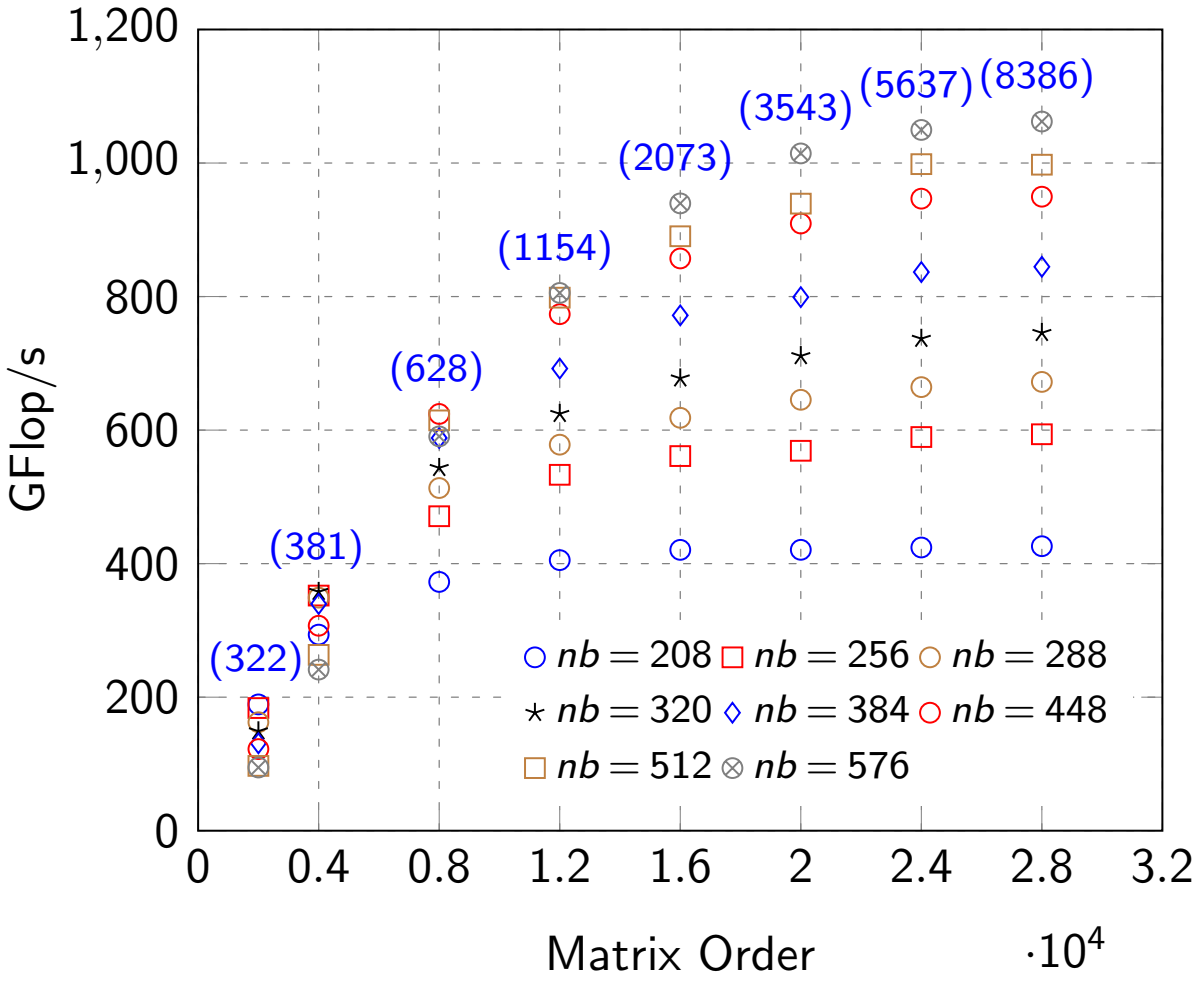
Which (*nb*, *ib*) to choose for a *n* given at runtime?

S1: Empirical+Exhaustive (LU)



(nb, ib) combination with a higher performance for each problem size n

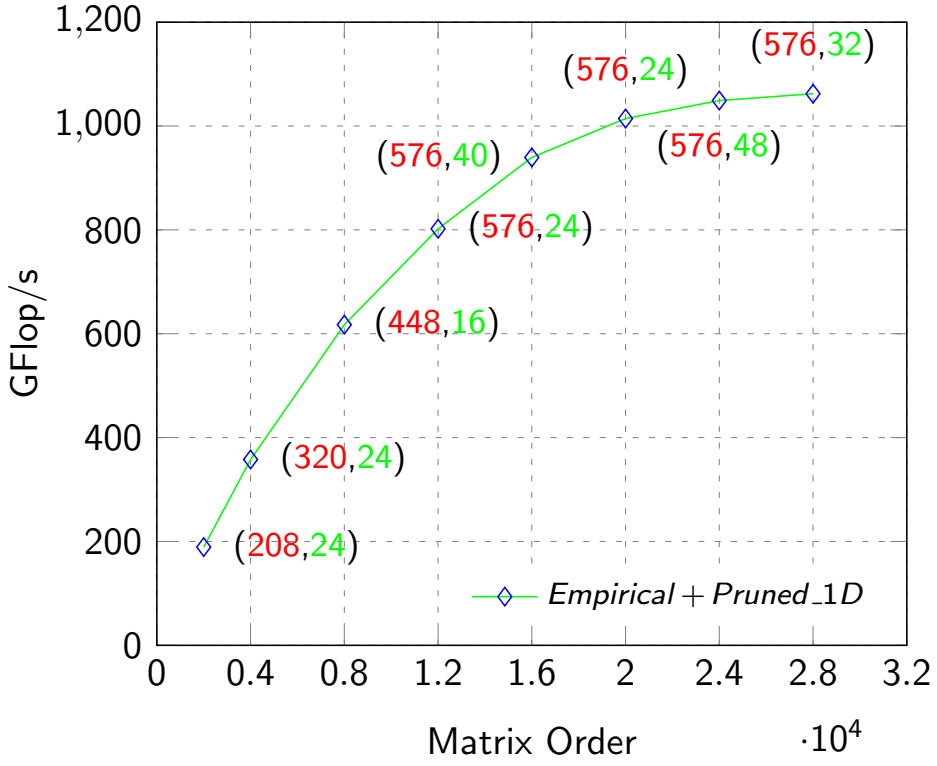
S1: Empirical+Exhaustive (LU)

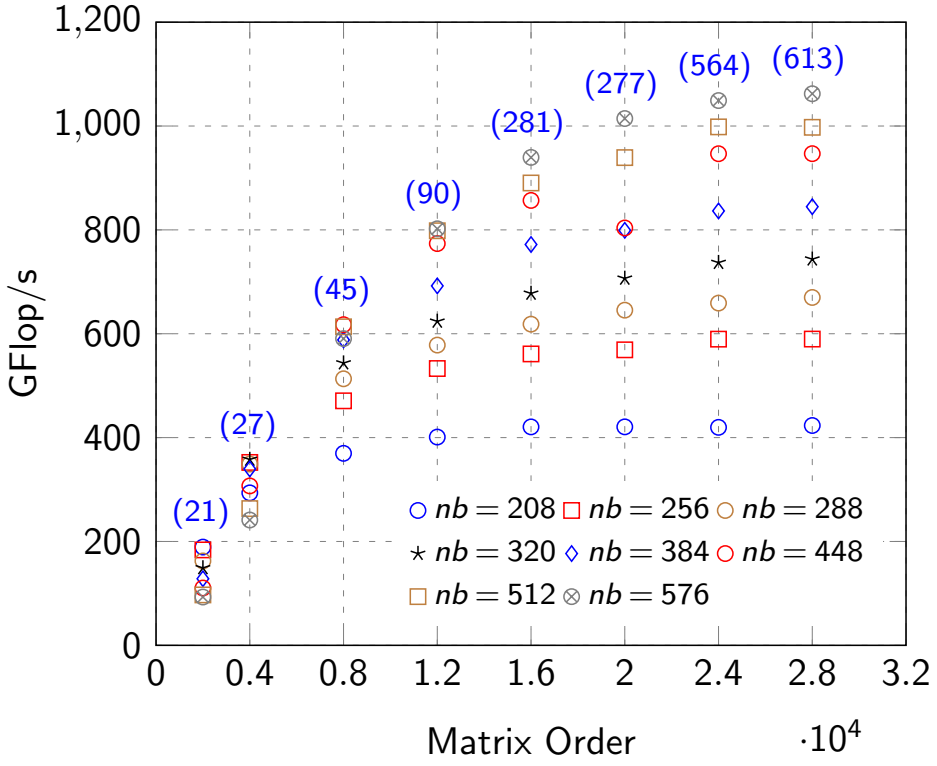


Experimental Time: 6 h 10 min

S2: Empirical+Pruned_1D (LU)

Search in *ib* dimension with threshold of 2% to avoid local minima.

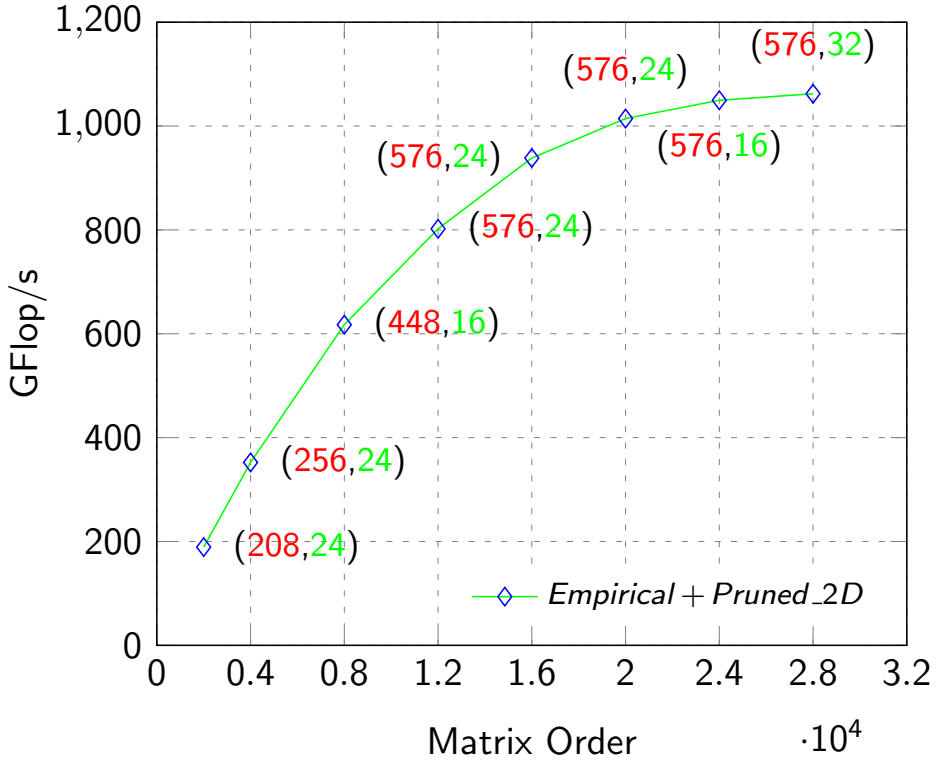


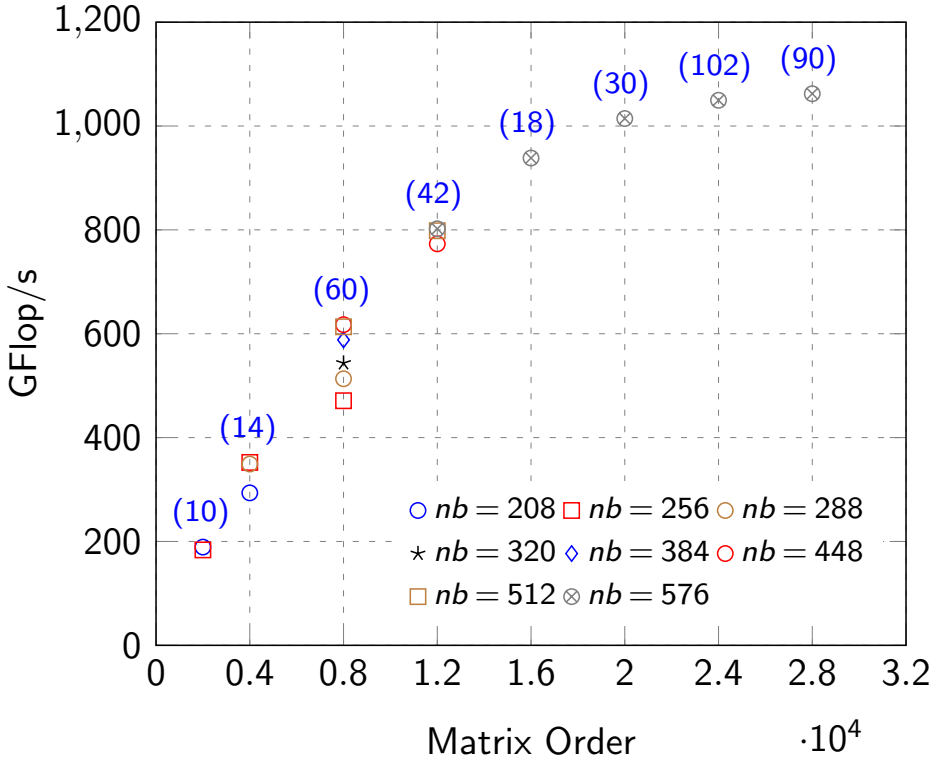


Experimental Time: 32 min

S2: Empirical+Pruned_2D (LU)

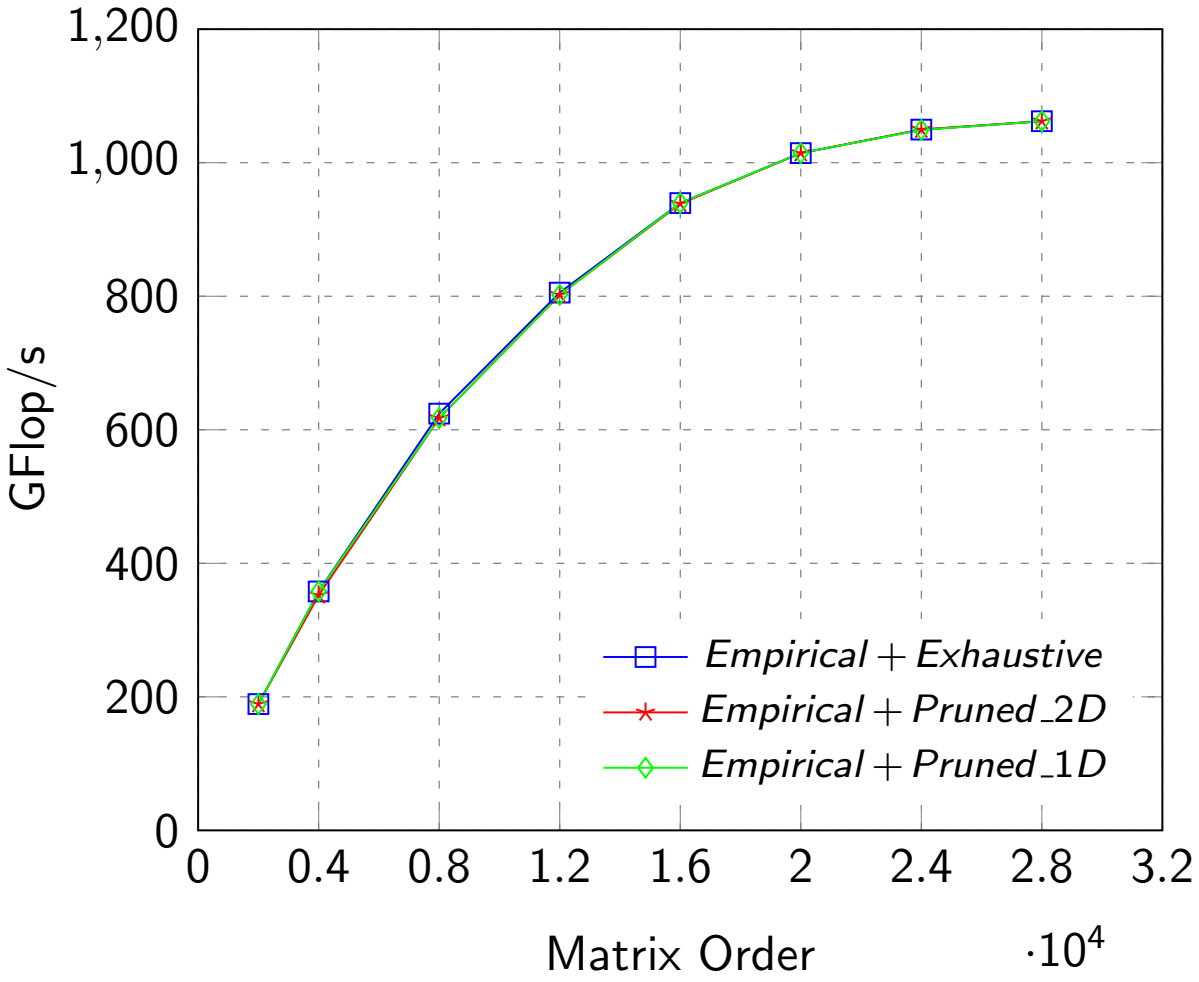
Search in both dimensions (*nb* and *ib*) with threshold in *ib* dimension.





Experimental Time: 6 min

S1 & S2 (LU)



S1: 6 h 10 min; **S2_1D**: 32 min **S2_2D**: 6 min

Overview

1. Motivation
2. Application Case
3. Search of the AP values
4. Selection of Computing Units
5. Selection of Scheduling Policy
6. Ongoing Work
7. Conclusions

Selection of Computing Units

StarPU

Assigns dinamically tasks to the CUs using a scheduling policy, but it does not take into account the computational power of the CUs.

Selection of Computing Units

StarPU

Assigns dinamically tasks to the CUs using a scheduling policy, but it does not take into account the computational power of the CUs.

Goal

Decide the best number of CUs (number of CPU cores and GPUs) to use for a given problem size n.

Selection of Computing Units

StarPU

Assigns dinamically tasks to the CUs using a scheduling policy, but it does not take into account the computational power of the CUs.

Goal

Decide the best number of CUs (number of CPU cores and GPUs) to use for a given problem size n.

Tuning Technique

Selective Searching: successively adds CUs in increasing power order using as starting point the best value for nb for the previous problem size n.

Selection of Computing Units

- Routine: Cholesky.
- Scheduling Policy: lws (by default)
- Strategy: Selective Searching.

				Performance (GFlops)		
n	*nb*	*CPU_Cores*	*GPUs*	*Tuned*	*Chameleon*	%
1000	112	12	$\{-\}$	76	46	40
2000	192	9	$\{1,5,0\}$	164	117	29
3000	192	8	$\{1,5,0,2\}$	285	196	31
4000	240	7	$\{1,5,0,2,3\}$	412	352	15
5000	256	6	$\{1,5,0,2,3,4\}$	545	465	15
6000	256	6	$\{1,5,0,2,3,4\}$	626	554	12
7000	320	6	$\{1,5,0,2,3,4\}$	687	608	12
8000	320	6	$\{1,5,0,2,3,4\}$	753	682	10

Overview

1. Motivation
2. Application Case
3. Search of the AP values
4. Selection of Computing Units
5. Selection of Scheduling Policy
6. Ongoing Work
7. Conclusions

Runtime System

StarPU

- Handles scheduling of tasks to the available CUs in the system.

Runtime System

StarPU

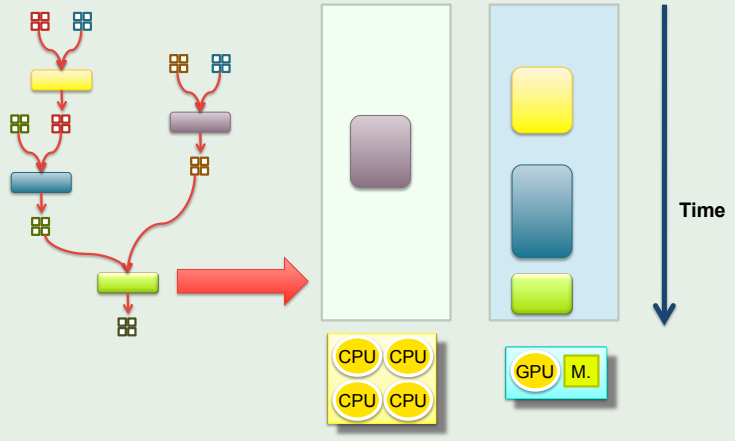
- Handles scheduling of tasks to the available CUs in the system.
- Execute tasks using optimized kernels of linear algebra libraries.

Runtime System

StarPU

- Handles scheduling of tasks to the available CUs in the system.
- Execute tasks using optimized kernels of linear algebra libraries.

Scheduling of Tasks



StarPU

Optimized Kernels

MKL, cuBLAS, . . .

Scheduling Policies

Performance

Varies depending on the scheduling policy selected.

Scheduling Policies

Performance

Varies depending on the scheduling policy selected.

Policies - Non-Performance Modelling

- **eager**: uses a central task queue from which workers draw tasks.
- **random**: uses a queue per worker, and distributes tasks randomly according to assumed worker overall performance.
- **ws** (work stealing): uses a queue per worker, and schedules a task on the worker which released it by default. When a worker becomes idle, it steals a task from the most loaded worker.
- **lws** (locality work stealing): similar to *ws*, but steals a task from neighbour workers. It also takes into account priorities.

Scheduling Policies

Performance

Varies depending on the scheduling policy selected.

Policies - Non-Performance Modelling

- **eager**: uses a central task queue from which workers draw tasks.
- **random**: uses a queue per worker, and distributes tasks randomly according to assumed worker overall performance.
- **ws** (work stealing): uses a queue per worker, and schedules a task on the worker which released it by default. When a worker becomes idle, it steals a task from the most loaded worker.
- **lws** (locality work stealing): similar to *ws*, but steals a task from neighbour workers. It also takes into account priorities.

Scheduling Policies

Performance

Varies depending on the scheduling policy selected.

Policies - Non-Performance Modelling

- **eager**: uses a central task queue from which workers draw tasks.
- **random**: uses a queue per worker, and distributes tasks randomly according to assumed worker overall performance.
- **ws** (work stealing): uses a queue per worker, and schedules a task on the worker which released it by default. When a worker becomes idle, it steals a task from the most loaded worker.
- **lws** (locality work stealing): similar to *ws*, but steals a task from neighbour workers. It also takes into account priorities.

Scheduling Policies

Performance

Varies depending on the scheduling policy selected.

Policies - Non-Performance Modelling

- **eager**: uses a central task queue from which workers draw tasks.
- **random**: uses a queue per worker, and distributes tasks randomly according to assumed worker overall performance.
- **ws** (work stealing): uses a queue per worker, and schedules a task on the worker which released it by default. When a worker becomes idle, it steals a task from the most loaded worker.
- **lws** (locality work stealing): similar to *ws*, but steals a task from neighbour workers. It also takes into account priorities.

Scheduling Policies

Performance

Varies depending on the scheduling policy selected.

Policies - Performance Model-Based

- **dm** (deque model): schedules tasks in the order they become available, without taking into account priorities.
- **dmda** (deque model data aware): similar to *dm*, but it also takes into account data transfer time.

Scheduling Policies

Performance

Varies depending on the scheduling policy selected.

Policies - Performance Model-Based

- **dm** (deque model): schedules tasks in the order they become available, without taking into account priorities.
- **dmda** (deque model data aware): similar to *dm*, but it also takes into account data transfer time.

Scheduling Policies

Performance

Varies depending on the scheduling policy selected.

Policies - Performance Model-Based

- **dm** (deque model): schedules tasks in the order they become available, without taking into account priorities.
- **dmda** (deque model data aware): similar to *dm*, but it also takes into account data transfer time.

Goal

Scheduling Policies

Performance

Varies depending on the scheduling policy selected.

Policies - Performance Model-Based

- **dm** (deque model): schedules tasks in the order they become available, without taking into account priorities.
- **dmda** (deque model data aware): similar to *dm*, but it also takes into account data transfer time.

Goal

- Analyze the impact on performance when using different policies.

Scheduling Policies

Performance

Varies depending on the scheduling policy selected.

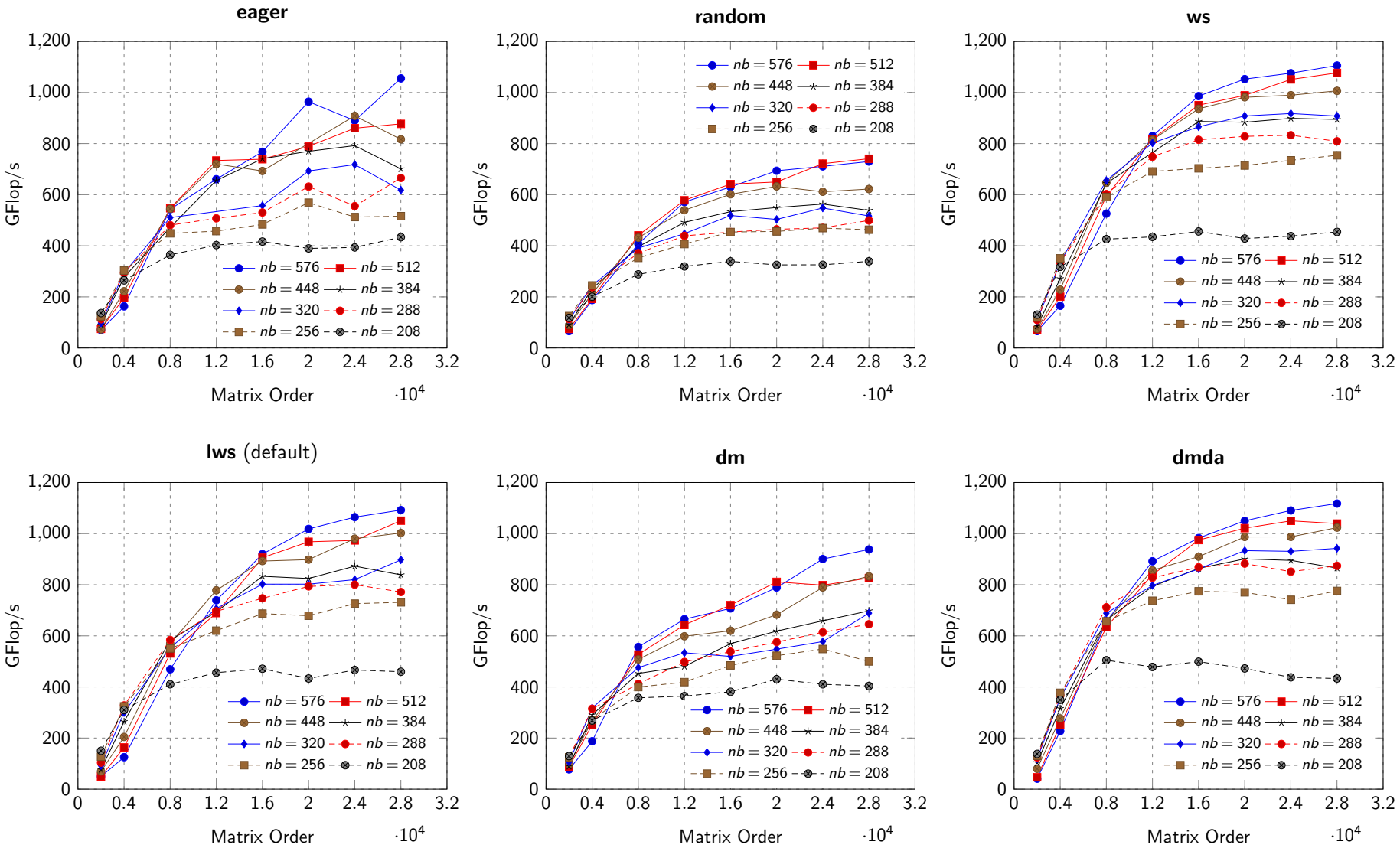
Policies - Performance Model-Based

- **dm** (deque model): schedules tasks in the order they become available, without taking into account priorities.
- **dmda** (deque model data aware): similar to *dm*, but it also takes into account data transfer time.

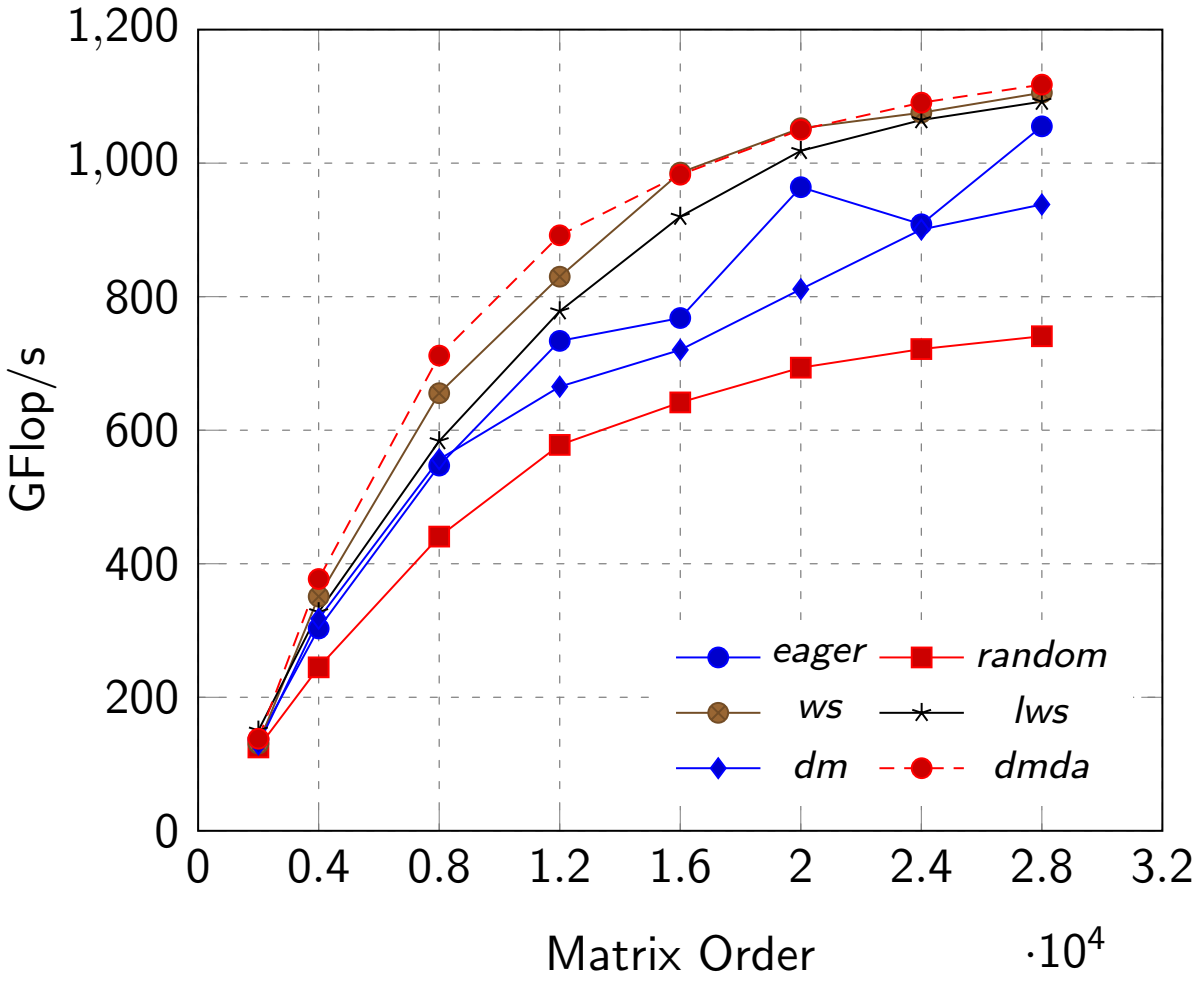
Goal

- Analyze the impact on performance when using different policies.
- Select the best scheduling policy for each problem size n.

Scheduling Policies (Cholesky)



Scheduling Policies (Cholesky)



Performance achieved by each scheduling policy using the best *nb* value for each problem size *n*.

Scheduling Policies (Cholesky)

Best Configuration

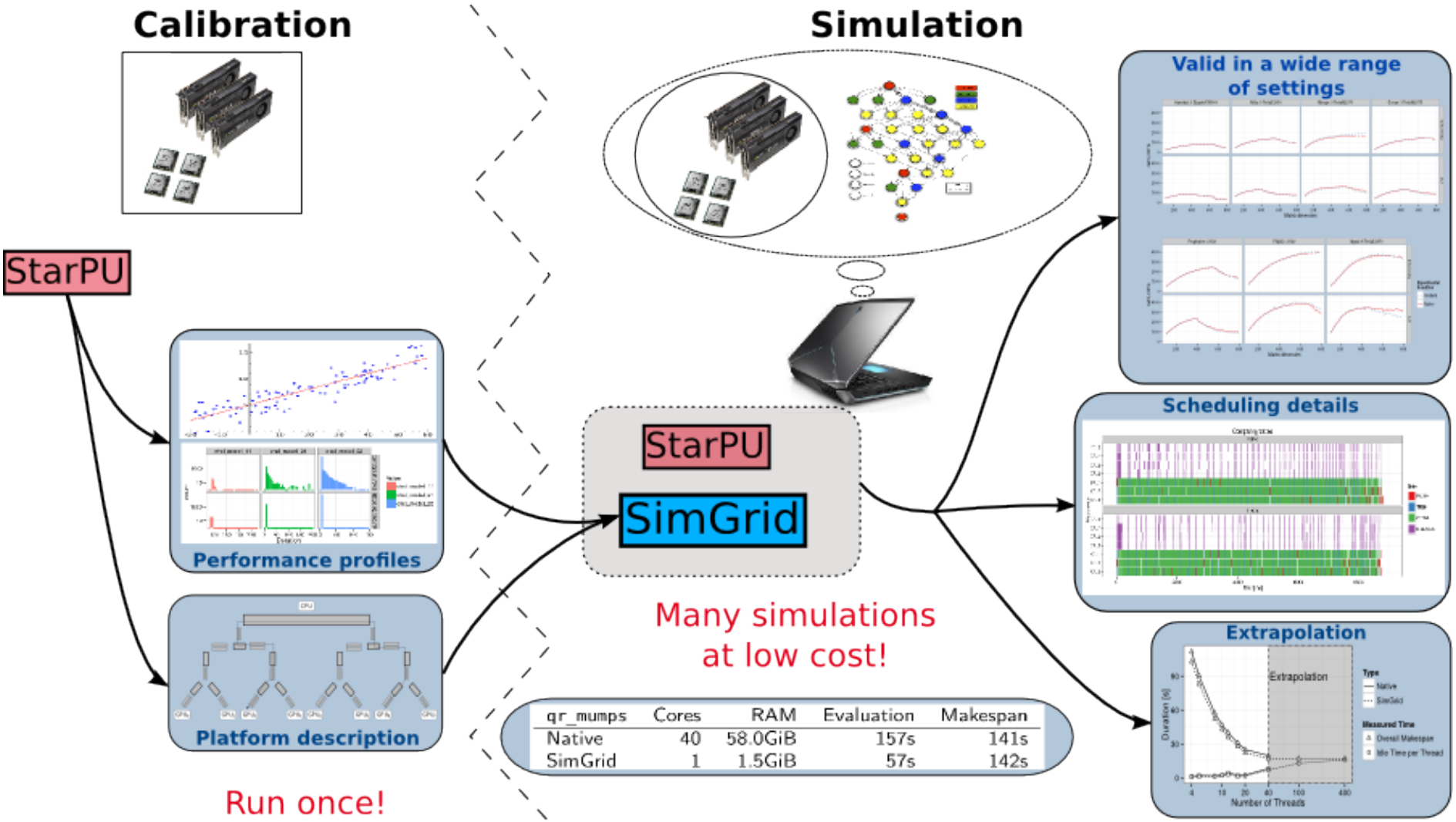
n	*nb*	*CPU_Cores*	*GPUs*	*Policy*	*GFlops*
2000	224	10	{1,5}	dm	169
4000	256	7	{1,5,0,2,3}	dmda	403
8000	320	6	{1,5,0,2,3,4}	dmda	719
12000	576	6	{1,5,0,2,3,4}	dmda	896
16000	576	6	{1,5,0,2,3,4}	dmda	1010
20000	672	6	{1,5,0,2,3,4}	ws	1051
24000	672	6	{1,5,0,2,3,4}	ws	1120
28000	672	6	{1,5,0,2,3,4}	ws	1150

Performance obtained when using the best scheduling policy with the best AP values (*nb*) and *CUs* (*CPU_Cores*, *GPUs*) for each problem size *n*.

Overview

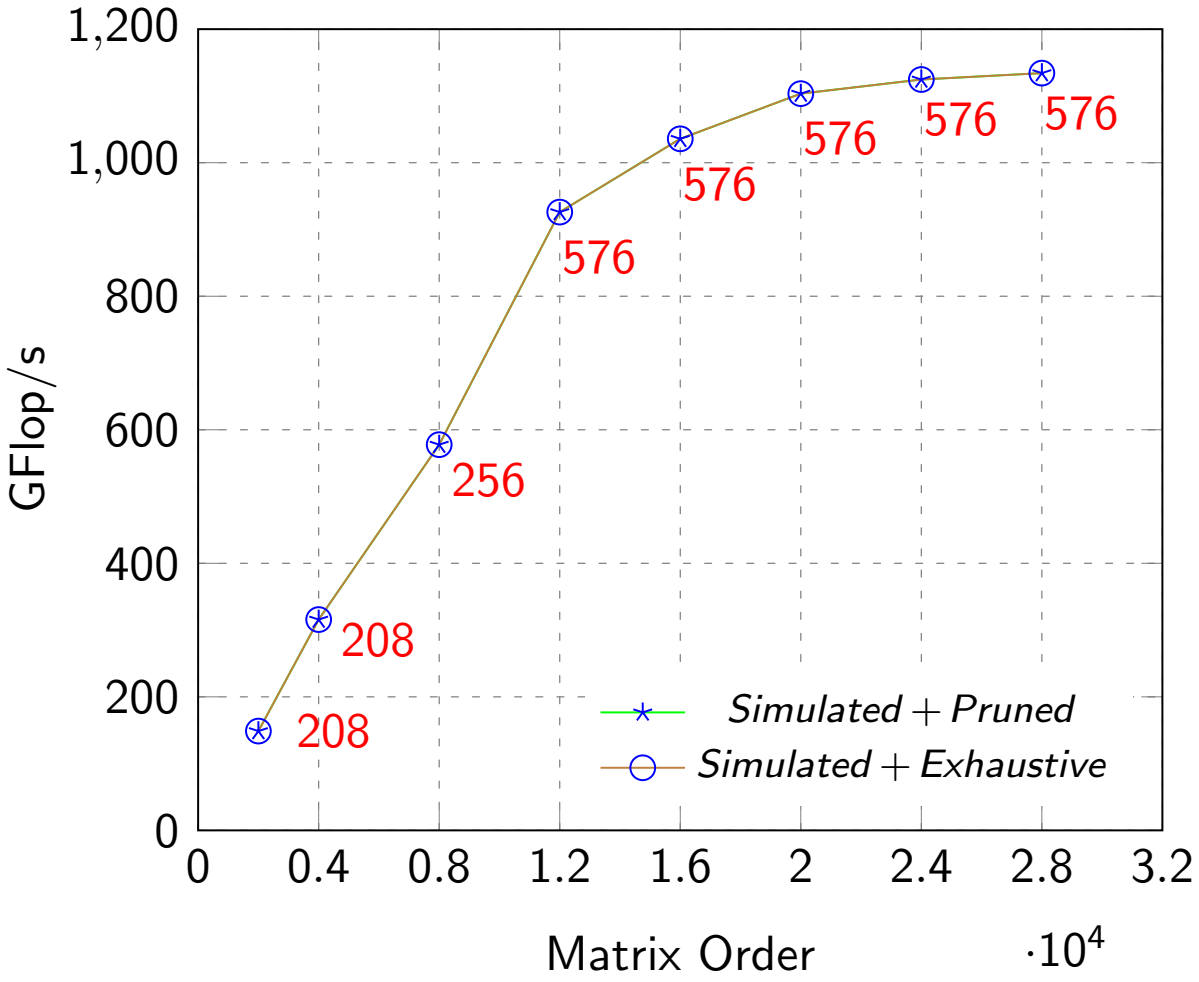
1. Motivation
2. Application Case
3. Search of the AP values
4. Selection of Computing Units
5. Selection of Scheduling Policy
6. Ongoing Work
7. Conclusions

How about using Simulation?



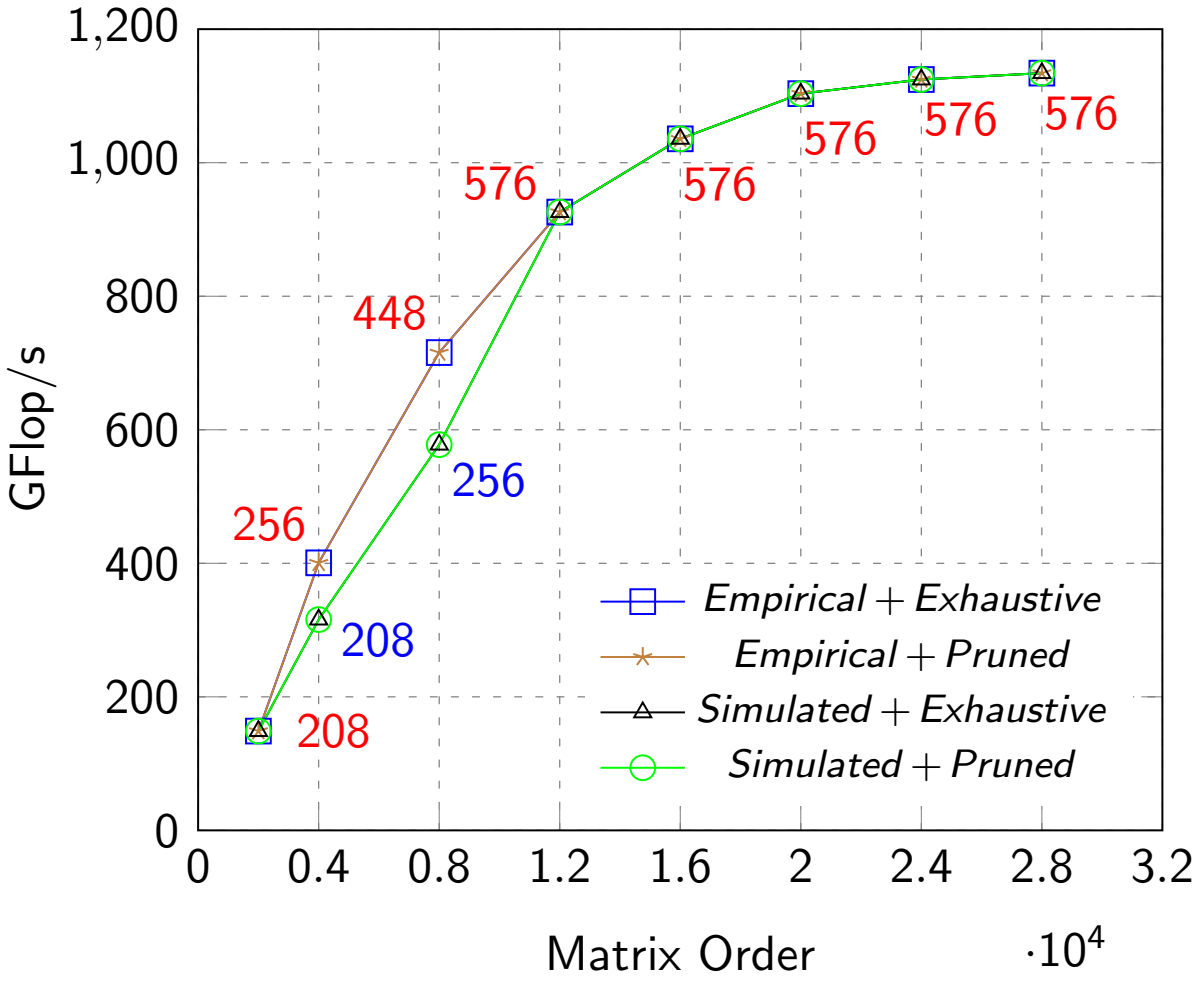
qr_mumps	Cores	RAM	Evaluation	Makespan
Native	40	58.0GiB	157s	141s
SimGrid	1	1.5GiB	57s	142s

Exhaustive vs. Pruned (Cholesky)



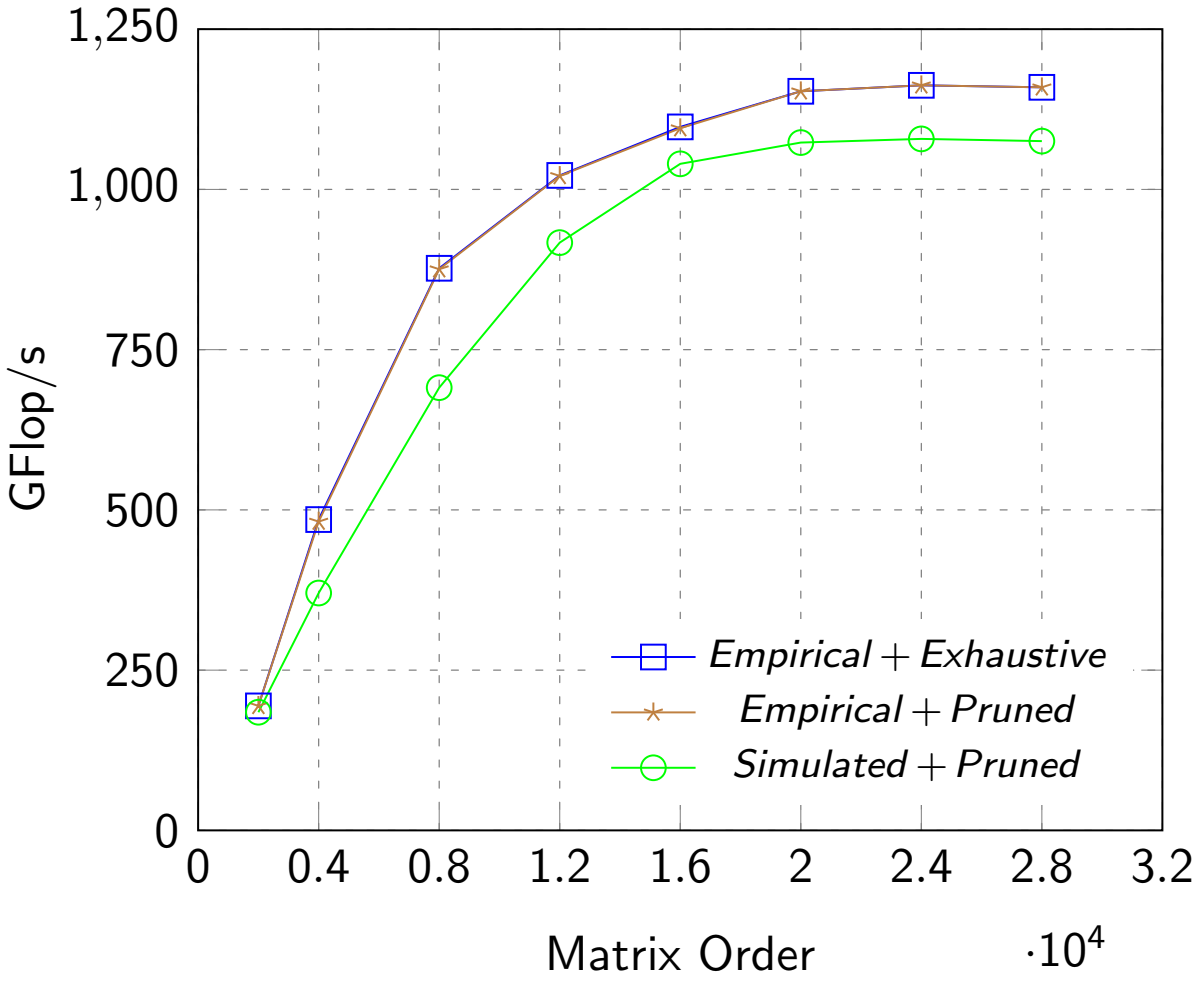
Exhaustive: 30 min; **Pruned**: 84 sec

S1 & S2 & Simulated (Cholesky)



S1: 30 min; **S2**: 154 sec; **Sim+Exh**: 30 min; **Sim+Pruned**: 84 sec

S1 & S2 & Simulated (LU)



S1: 6 h 10 min; **S2_2D**: 6 min; **Sim+Pruned_2D**: 4 min

Overview

1. Motivation
2. Application Case
3. Search of the AP values
4. Selection of Computing Units
5. Selection of Scheduling Policy
6. Ongoing Work
7. Conclusions

Conclusions

- The *pruned* strategy allows to obtain good values for the APs with low experimental times both for the Cholesky and LU routines.
- Performance achieved with the *pruned* strategy overlaps with that achieved by the *empirical* one.
- The *Simulated+Pruned* is a decent tuning strategy to obtain performances close to the *empirical* without (almost) using the system.
- An appropriate selection of the CUs and the Scheduling Policies for each problem size allows to improve the performance with an efficient exploitation of the heterogeneous platform.

Thanks for your Attention!