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

**Task-Based Libraries** = 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**
  ```cpp
  for (j = 0; j < N; j++)
    Task(A[j]);
  ```
- **Direct Acyclic Graph**
- **Runtime Systems (Scheduling of Tasks)**
- **Heterogeneous Platforms**
  - **StarPU**
  - **Optimized Kernels**
    - MKL, cuBLAS, ...

**Empirical and Simulation-Based Autotuning**

Seattle, Washington, February 2020
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.
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 ($nb$)
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?
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.
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

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

$n_b, i_b$

Goal
For each problem size $n$:
Optimization Strategies

**Goal**

For each problem size $n$:
- Find the best values for $nb$ and $ib$.
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

**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

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

**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

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

```c
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

```c
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]);
        }
    }
}

Jesús Cámara (University of Murcia)   Empirical and Simulation-Based Autotuning   Seattle, Washington, February 2020
The Cholesky Decomposition

```c
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

```c
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]);
        }
    }
}
```
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]);
        }
    }
}
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]);
        }
    }
}
```
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

```c
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]);
        }
    }
}

GEMM
SYRK
TRSM
POTRF

POTRF
TRSM
SYRK
GEMM
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]);
        }
    }
}

GEMM
SYRK
TRSM
POTRF

Jesús Cámara (University of Murcia)
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?

In principle any.
An arbitrary set is considered for $nb$:

- 208, 256, 288, ...

ib = 16, 32, 64, ...
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

- **Marte**: 1 AMD Phenom II X6 1075T (800 MHz) (1 x 6 cores)
  - 4 GB RAM
  - L1: 04 KB, L2: 32 KB, L3: 6 MB

- **Mercurio**: 1 AMD Phenom II X6 1075T (800 MHz) (1 x 6 cores)
  - 16 GB RAM
  - L1: 04 KB, L2: 512 KB, L3: 6 MB

- **Saturno**: 4 Intel Xeon EP2660 (1.87 GHz) (4 x 5 cores)
  - 32 GB RAM
  - L1: 32 KB, L2: 256 KB, L3: 15 MB

- **Jupiter**: 2 Intel Xeon E5-2620 (2.00 GHz) (2 x 6 cores)
  - 32 GB RAM
  - L1: 32 KB, L2: 256 KB, L3: 15 MB

- **Venus**: 2 Intel Xeon E5-2620 v3 (2.60 GHz) (2 x 6 cores)
  - 64 GB RAM
  - L1: 32 KB, L2: 256 KB, L3: 15 MB

- **GPU**
  - Geforce GTX 480 (Fermi)
  - 480 cores - 1.5 GB

- **GPU**
  - Geforce GTX 460 (Fermi)
  - 480 cores - 1.5 GB

- **GPU**
  - Tesla C2075 (Kepler)
  - 2496 cores - 5 GB

- **GPU**
  - Tesla C2050 (Fermi)
  - 448 cores - 6 GB

- **GPU**
  - Geforce GTX 560 (Fermi)
  - 512 cores - 1.5 GB

- **GPU**
  - Geforce GTX 560 (Fermi)
  - 512 cores - 1.5 GB

- **GPU**
  - Geforce GTX 560 (Fermi)
  - 512 cores - 1.5 GB

- **GPU**
  - Geforce GTX 560 (Fermi)
  - 512 cores - 1.5 GB

- **GPU**
  - Geforce GTX 560 (Fermi)
  - 512 cores - 1.5 GB

- **GPU**
  - Geforce GTX 560 (Fermi)
  - 512 cores - 1.5 GB

- **MIC**
  - Xeon Phi 3120A KNL
  - 57 cores (1.1 GHz)
  - 6 GB - 28.5 MB L2
Heterogeneous Platform

Jupiter

- 2 Intel Xeon E5-2620 Hexa-Core (12 CPU Cores).
- 4 GPU NVIDIA GeForce GTX590.
- 2 GPU NVIDIA Tesla C2075.
Performance significantly depends on the value of the Tile Size ($n_b$).
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?
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)

Matrix Order vs. GFlop/s graph with a marker at (208, 0).

Jesús Cámara (University of Murcia) - Empirical and Simulation-Based Autotuning - Seattle, Washington, February 2020
S2: Empirical+Pruned (Cholesky)

Matrix Order

GFlop/s

0 0.4 0.8 1.2 1.6 2 2.4 2.8 3.2

10^4

256
S2: Empirical+Pruned (Cholesky)

![Graph showing the relationship between matrix order and GFlop/s for empirical and pruned Cholesky methods. The graph includes a marker at (208, 208) indicating a specific point of interest.]
S2: Empirical+Pruned (Cholesky)
S2: Empirical+Pruned (Cholesky)

Matrix Order vs. GFlop/s for various matrix orders. The plot shows performance (GFlop/s) on the y-axis and matrix order on the x-axis. Key points include:

- Matrix Order: 208, GFlop/s: 256
- Matrix Order: 256, GFlop/s: 600
- Matrix Order: 320, GFlop/s: 1200

Jesús Cámara (University of Murcia) Empirical and Simulation-Based Autotuning Seattle, Washington, February 2020
S2: Empirical+Pruned (Cholesky)

Matrix Order vs. GFlop/s

- 208
- 256
- 288

Jesús Cámara (University of Murcia)  Empirical and Simulation-Based Autotuning  Seattle, Washington, February 2020
S2: Empirical+Pruned (Cholesky)

Matrix Order vs GFlop/s plot:
- Points at 208, 256, and 320 GFlop/s
- Matrix Orders: 0, 0.4, 0.8, 1.2, 1.6, 2, 2.4, 2.8, 3.2
S2: Empirical + Pruned (Cholesky)

Matrix Order vs. GFlop/s graph with points at matrix orders 208, 256, and 384.
S2: Empirical+Pruned (Cholesky)

![Graph showing performance (GFlop/s) vs. matrix order for empirical and pruned Cholesky implementations. Data points for matrix orders 208, 256, and 512 are marked.](image-url)
S2: Empirical+Pruned (Cholesky)

Matrix Order vs. GFlop/s for different matrix orders and GFlop rates. The graph shows the performance of empirical and pruned Cholesky algorithms for various matrix orders.
S2: Empirical + Pruned (Cholesky)

![Graph showing performance of Empirical + Pruned (Cholesky) for different matrix orders. The x-axis represents the matrix order, and the y-axis represents GFlop/s. The graph includes data points for matrix orders of 208, 256, 448, and 512.]
S2: Empirical + Pruned (Cholesky)
S2: Empirical + Pruned (Cholesky)

Matrix Order vs. GFlop/s graph showing performance of the empirical + pruned Cholesky method with different matrix orders. The graph plots matrix order on the x-axis and GFlop/s on the y-axis. Key points include:

- Matrix Order 208 with GFlop/s of 208
- Matrix Order 256 with GFlop/s of 448
- Matrix Order 576 with GFlop/s of 576

The graph illustrates the performance scaling with increasing matrix order.
S2: Empirical + Pruned (Cholesky)

Experimental Time: 154 seconds
S1 & S2 (Cholesky)

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

![Graph showing GFlop/s vs Matrix Order for Empirical + Pruned and Empirical + Exhaustive methods]

**S1**: 30 min; **S2**: 154 seconds
Best value for $ib$ with current $nb$ for each problem size $n$. 

Matrix Order

GFlop/s

$n = \cdot 10^4$

$n = 208$
$ib$ is incremented from 8 to $nb$ in steps of 8.
Performance significantly depends on the value of $nb$ and $ib$. 

$n b = 208$  
$n b = 256$  
$n b = 288$  
$n b = 320$  
$n b = 384$  
$n b = 448$  
$n b = 512$  
$n b = 576$
Which \((nb, ib)\) to choose for a \(n\) given at runtime?
(nb, ib) combination with a higher performance for each problem size n
Experimental Time: 6 h 10 min
Search in $ib$ dimension with threshold of 2% to avoid local minima.

Experimental Time: 32 min
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 dynamically 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 dynamically 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)**

<table>
<thead>
<tr>
<th>$n$</th>
<th>$nb$</th>
<th>CPU_Cores</th>
<th>GPUs</th>
<th>Tuned</th>
<th>Chameleon</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>112</td>
<td>12</td>
<td>${-}$</td>
<td>76</td>
<td>46</td>
<td>40</td>
</tr>
<tr>
<td>2000</td>
<td>192</td>
<td>9</td>
<td>${1,5,0}$</td>
<td>164</td>
<td>117</td>
<td>29</td>
</tr>
<tr>
<td>3000</td>
<td>192</td>
<td>8</td>
<td>${1,5,0,2}$</td>
<td>285</td>
<td>196</td>
<td>31</td>
</tr>
<tr>
<td>4000</td>
<td>240</td>
<td>7</td>
<td>${1,5,0,2,3}$</td>
<td>412</td>
<td>352</td>
<td>15</td>
</tr>
<tr>
<td>5000</td>
<td>256</td>
<td>6</td>
<td>${1,5,0,2,3,4}$</td>
<td>545</td>
<td>465</td>
<td>15</td>
</tr>
<tr>
<td>6000</td>
<td>256</td>
<td>6</td>
<td>${1,5,0,2,3,4}$</td>
<td>626</td>
<td>554</td>
<td>12</td>
</tr>
<tr>
<td>7000</td>
<td>320</td>
<td>6</td>
<td>${1,5,0,2,3,4}$</td>
<td>687</td>
<td>608</td>
<td>12</td>
</tr>
<tr>
<td>8000</td>
<td>320</td>
<td>6</td>
<td>${1,5,0,2,3,4}$</td>
<td>753</td>
<td>682</td>
<td>10</td>
</tr>
</tbody>
</table>
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
StarPU

- Handles scheduling of tasks to the available CUs in the 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

Optimized Kernels

MKL, cuBLAS, ...
Performance

Varies depending on the scheduling policy selected.
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.
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.
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.
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 $n$. 

Jesús Cámara (University of Murcia) Empirical and Simulation-Based Autotuning Seattle, Washington, February 2020
Scheduling Policies (Cholesky)

- **eager**
- **random**
- **ws**
- **lws** (default)
- **dm**
- **dmda**

Matrix Order vs. GFlop/s

**Jesús Cámara** (University of Murcia)  
Empirical and Simulation-Based Autotuning  
Seattle, Washington, February 2020
Scheduling Policies (Cholesky)

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

- **eager**
- **random**
- **ws**
- **lws**
- **dm**
- **dmda**
## Scheduling Policies (Cholesky)

### Best Configuration

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

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?

Calibration

Simulation

Valid in a wide range of settings

StarPU

SimGrid

Many simulations at low cost!

<table>
<thead>
<tr>
<th>qr_mumps</th>
<th>Cores</th>
<th>RAM</th>
<th>Evaluation</th>
<th>Makespan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>40</td>
<td>58.0GiB</td>
<td>157s</td>
<td>141s</td>
</tr>
<tr>
<td>SimGrid</td>
<td>1</td>
<td>1.5GiB</td>
<td>57s</td>
<td>142s</td>
</tr>
</tbody>
</table>

StarPU

Performance profiles

Platform description

Run once!
Exhaustive vs. Pruned (Cholesky)

![Graph showing GFlop/s vs. Matrix Order for Exhaustive and Pruned methods.](image)

- **Exhaustive**: 30 min
- **Pruned**: 84 sec
**S1 & S2 & Simulated (Cholesky)**

The graph shows the performance of different algorithms in terms of GFlop/s (Giga Floating Point Operations per Second) against the matrix order. The algorithms compared are:

- **Empirical + Exhaustive**
- **Empirical + Pruned**
- **Simulated + Exhaustive**
- **Simulated + Pruned**

The key metrics are:

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

The graph indicates a clear trend in performance across different matrix orders for each methodology.
S1 & S2 & Simulated (LU)

\[\begin{array}{ccc}
0.4 & 0.8 & 1.2 \\
0.8 & 1.2 & 1.6 \\
1.6 & 2 & 2.4 \\
2.4 & 2.8 & 3.2 \\
\end{array}\]

- S1: 6 h 10 min
- S2 2D: 6 min
- Sim + Pruned 2D: 4 min

Jesús Cámara (University of Murcia)
Empirical and Simulation-Based Autotuning
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
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!