A Hierarchical Approach for Autotuning Linear Algebra Routines on Heterogeneous Clusters

J. Cámara, J. Cuenca, L-P. García and D. Giménez
University of Murcia, Spain

18th International Conference on
Computational and Mathematical Methods
in Science and Engineering
July, 2018

Contents

Contents

Introduction

- Most of the HPC platforms are composed of several nodes with different architecture and computational capacity.
- Nodes are connected through a high-speed communication network and, in general, each one consists of a *multicore* CPU and one or more coprocessors (GPU, MIC or other).
- New optimization techniques are required to efficiently use these platforms.
- This work presents a hierarchical *bottom-up* approach for autotuning linear algebra routines on heterogeneous platforms.

Introduction

- The hierarchical approach allows to install and auto-optimize the routines by levels, reducing the installation time at each level of the hierarchy.

- The proposed methodology uses an autotuning engine to decide at each level of the hierarchy and for each problem size, the best configuration to use for each computational element.

- Decisions are taken based on the performance information stored in the previous hierarchy level for each computational element for the closest size.

Contents

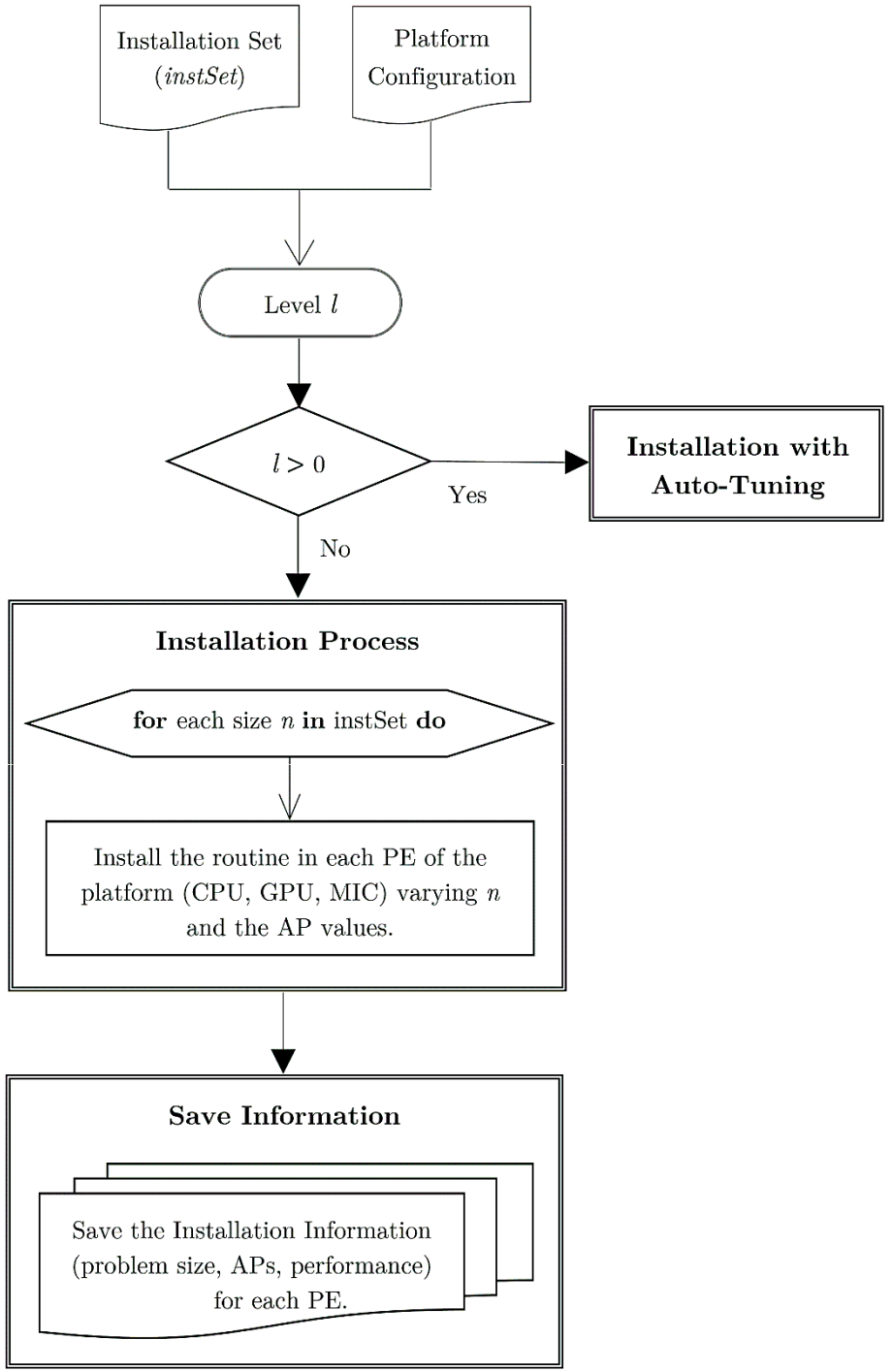
Hierarchical Autotuning Methodology

- Each level of the hierarchy is associated to a HW level of the heterogeneous platform:
  - Level 0: processing elements (CPU, GPU and MIC).
  - Level 1: hybrid nodes, each one composed of several processing elements of Level 0.
  - Level 2: cluster, composed of a set of nodes of Level 1.

- At each level and for each problem size, the installation process experimentally searches the best values for a set of adjustable parameters (AP) for the computational elements of such level and stores the selected configuration and the performance achieved.
  - Level 0: *AP* = number of threads in CPU or MIC (no parameters are considered for GPU)
  - Level 1: *AP* = amount of work to assign to each processing element (CPU, GPU and MIC)
  - Level 2: *AP* = amount of work to assign to each node of the cluster.

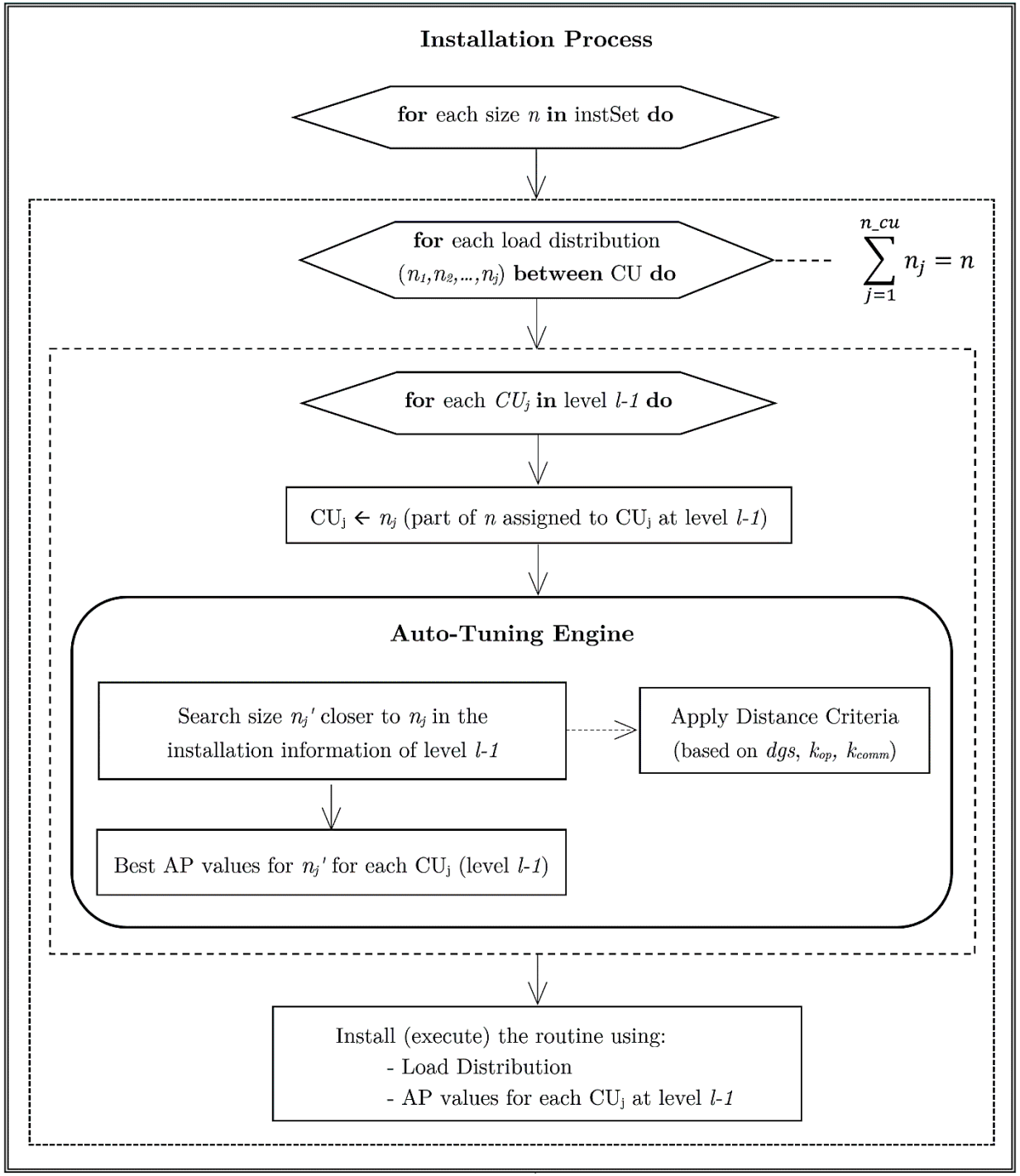
Hierarchical Autotuning Methodology

- The installation of the routines is carried out by levels, starting at the lowest level of the platform and going upward to higher levels by applying an autotuning process.



Hierarchical Autotuning Methodology

- When $level > 0$, for each problem size, the autotuning engine searches in the installation information stored for the previous level, the closest size to the load assigned to each computing unit and selects the best *AP* values.



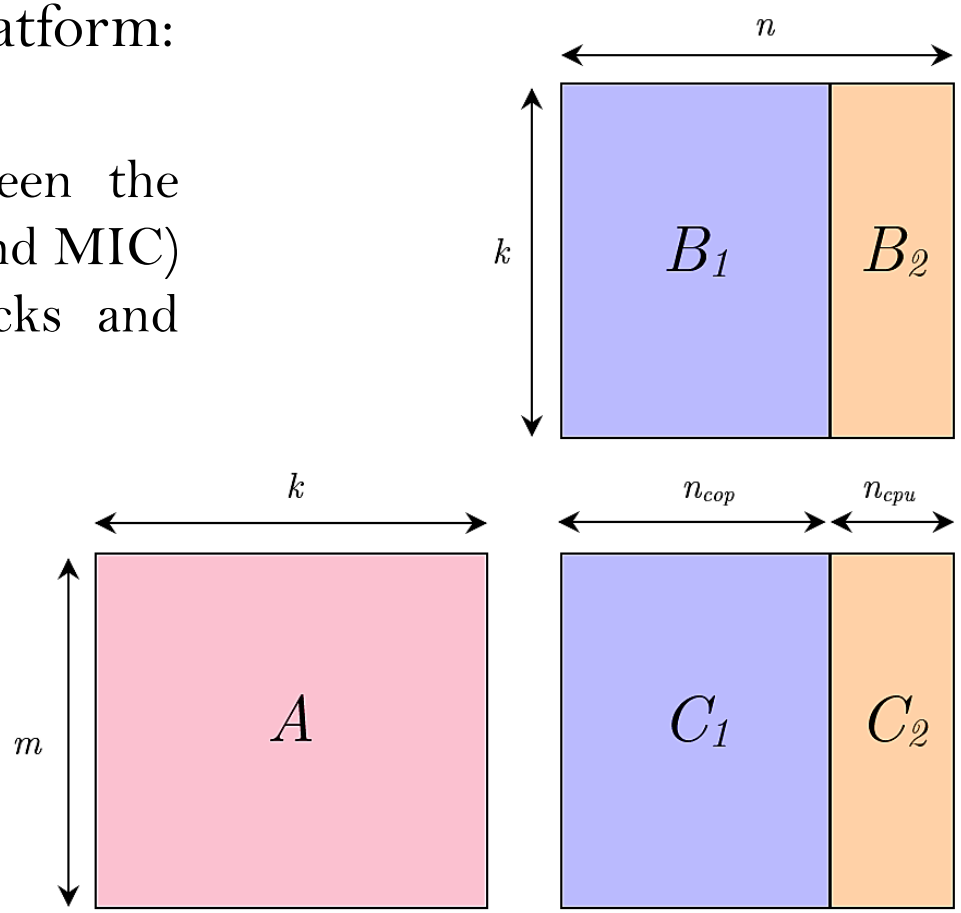
Contents

Parallel Matrix Multiplication

- Matrix multiplication has been redefined in order to obtain a balanced load distribution between the computational elements of the platform.
- Our aim is not to optimize the matrix multiplication, but to show how the autotuning methodology works on the heterogeneous platforms considered.
- Optimized implementations of BLAS are used to exploit the multithreaded parallelism: Intel MKL for CPU and MIC and cuBLAS for GPU.
- The proposed methodology can be applied in the same way with other linear algebra libraries.

Parallel Matrix Multiplication

- Matrix multiplication is carried out by distributing the workload to each computing element of the heterogeneous platform:
  - At node level, matrix A is replicated between the processing elements of the node (CPU, GPU and MIC) and matrix B is partitioned in column blocks and distributed between them.
  - At platform level, matrix A is replicated between nodes and matrix B is distributed between them, where each coprocessor j of node i calculates a block of size $n_{i,j}$ of the submatrix B assigned to the node and the CPU calculates a block of size $n_{i,cpu}$



Contents

Experimental Results

- Results are shown for **Heterosolar**, a heterogeneous cluster composed of 5 compute nodes connected through a Gigabit Ethernet network.
- Nodes are of 4 types: each of them includes a *multicore* CPU with one or more GPU of different types, and one also includes 2 MIC.
- Experiments are focused on showing that *AP* values selected with the autotuning engine at each hierarchy level provides times close to the experimental optimum.
- In order to reduce the amount of executions during the installation process for the different combinations of the *AP* values, a value of 10% has been used to set the distribution grain size (*DGS*) of matrix *B* between the computing elements.

Experimental Results

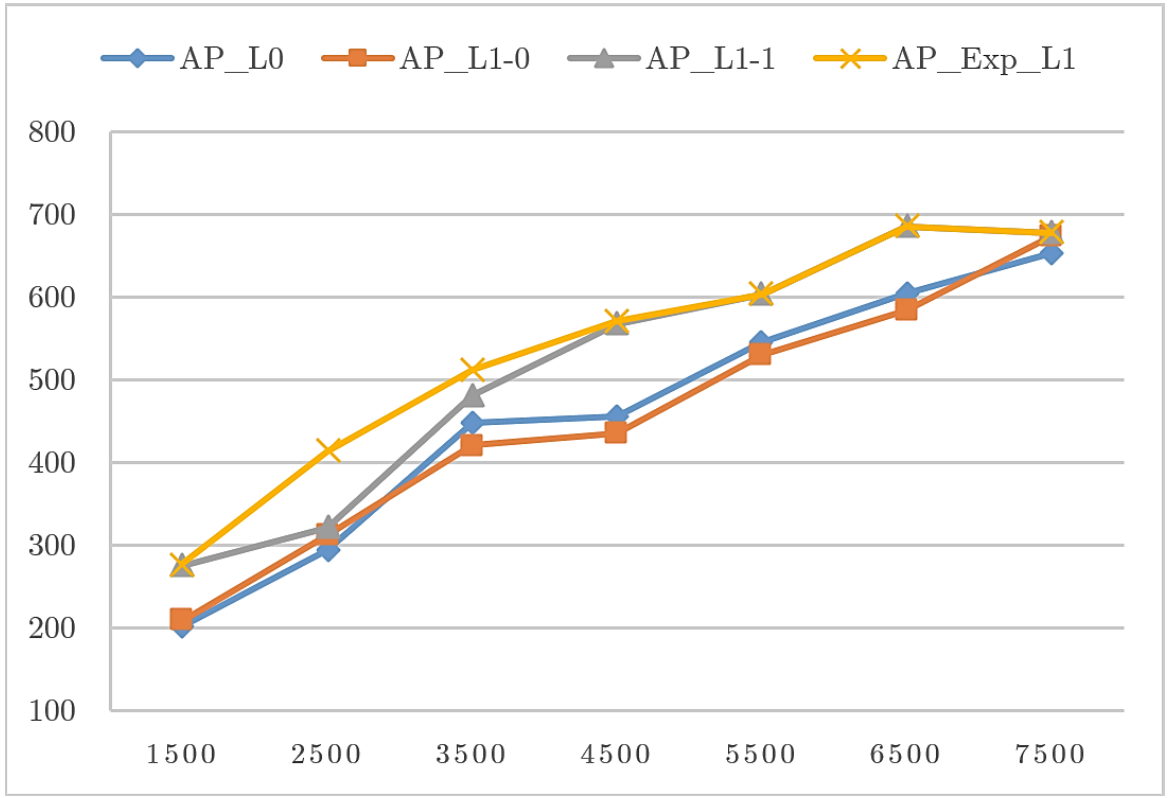
- Results are shown for the most heterogeneous nodes of **Heterosolar**:
  - **12CtwoC2075fourGTX590**: 1 CPU with 12 cores, 2 GPU NVIDIA Tesla C2075 Fermi and 4 GPU NVIDIA GeForce GTX 590 Fermi.
  - **12ConeGT640KtwoMIC**: 1 CPU with 12 cores, 1 GPU NVIDIA GeForce GT 640 Kepler and 2 Intel Xeon Phi 3120A KNC.
- Experiments has been carried out with problem sizes {1500, 2500, ...}, an example of a set of sizes that a user could use to test the methodology.
- For each problem size n, the autotuning engine will select the AP values stored in the previous level for the problem size closest to n and will run the routine with those values to solve the problem.

Experimental Results

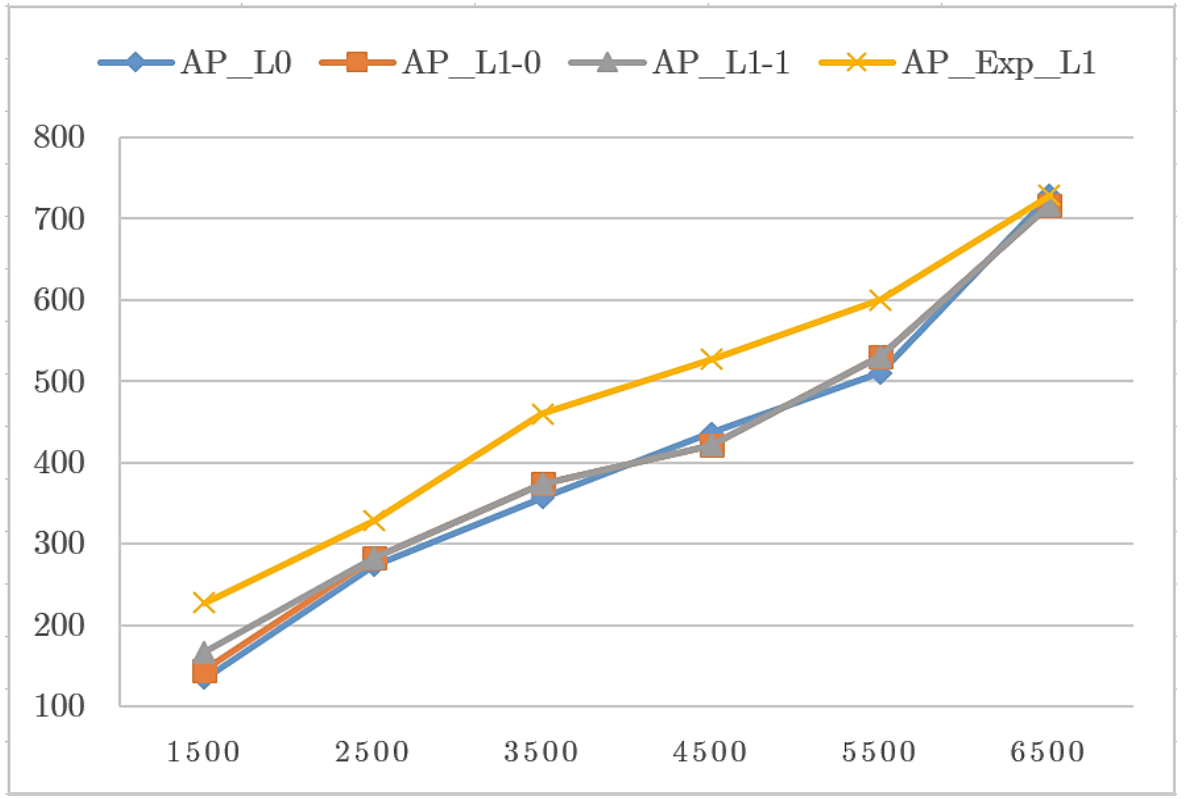
- We consider 3 types of auto-optimization at Level 1, depending on the way in which the autotuning engine uses the information stored at Level 0:
  - **AT-L0**: uses the performance information stored at L0 for each processing element to determine the load distribution between them.
  - **AT-L1-0**: performs a L1 installation determining the workload distribution between processing elements, using the best L0 *AP* values stored for them.
  - **AT-L1-1**: performs a L1 installation by searching the *AP* values for this level, using for each one the *AP* values stored at L0 for each processing element.
- Each of the 3 types offer similar installation times and performance results close to the experimental optimum.

Experimental Results

12CtwoC2075fourGTX590



12ConeGT640KtwoMIC



AP values selected by the AT-L0 auto-optimization type

n	(thr_{CPU}, n_{CPU})	n_{GPU_0}	n_{GPU_1}	n_{GPU_2}	n_{GPU_3}	n_{GPU_4}	n_{GPU_5}	GFLOPS
1500	(22,150)	150	300	150	150	150	450	208
3500	(11,350)	350	700	350	350	350	1050	420
5500	(22,550)	550	1100	550	550	550	1650	541
7500	(13,750)	750	2250	750	750	750	1500	528

AP values selected by the AT-L1-0 auto-optimization type

n	(thr_{CPU}, n_{CPU})	n_{GPU_0}	n_{GPU_1}	n_{GPU_2}	n_{GPU_3}	n_{GPU_4}	n_{GPU_5}	GFLOPS
1500	(22,150)	150	300	150	150	150	450	208
3500	(11,350)	350	700	350	350	350	1050	420
5500	(11,550)	550	1100	550	550	550	1650	529
7500	(12,750)	750	2250	750	750	750	1500	673

AP values selected by the AT-L1-1 auto-optimization type

n	(thr_{CPU}, n_{CPU})	n_{GPU_0}	n_{GPU_1}	n_{GPU_2}	n_{GPU_3}	n_{GPU_4}	n_{GPU_5}	GFLOPS
1500	(0,0)	0	450	300	300	0	450	275
3500	(0,0)	350	1050	350	350	350	1050	481
5500	(0,0)	550	1650	550	550	550	1650	603
7500	(12,750)	0	2250	750	750	750	2250	677

Experimental *AP* values at L1

n	(thr_{CPU}, n_{CPU})	n_{GPU_0}	n_{GPU_1}	n_{GPU_2}	n_{GPU_3}	n_{GPU_4}	n_{GPU_5}	GFLOPS
1500	(8,300)	0	300	0	300	300	300	277
3500	(0,0)	350	1050	350	700	0	1050	511
5500	(0,0)	550	1650	550	550	550	1650	603
7500	(12,750)	0	2250	750	750	750	2250	677

AP values selected by the AT-L0 auto-optimization type

n	(thr_{CPU}, n_{CPU})	n_{GPU}	(thr_{MIC_0}, n_{MIC_0})	(thr_{MIC_1}, n_{MIC_1})	GFLOPS
1500	(12,450)	0	(112,600)	(108,450)	134
3500	(11,700)	0	(216,1400)	(216,1400)	357
5500	(11,1100)	0	(220,2200)	(212,2200)	510

AP values selected by the AT-L1-0 auto-optimization type

n	(thr_{CPU}, n_{CPU})	n_{GPU}	(thr_{MIC_0}, n_{MIC_0})	(thr_{MIC_1}, n_{MIC_1})	GFLOPS
1500	(12,450)	0	(112,600)	(112,450)	145
3500	(11,700)	0	(220,1400)	(224,1400)	374
5500	(11,1100)	0	(224,2200)	(220,2200)	531

AP values selected by the AT-L1-1 auto-optimization type

n	(thr_{CPU}, n_{CPU})	n_{GPU}	(thr_{MIC_0}, n_{MIC_0})	(thr_{MIC_1}, n_{MIC_1})	GFLOPS
1500	(12,300)	0	(112,600)	(208,600)	167
3500	(11,700)	0	(220,1400)	(224,1400)	374
5500	(11,1100)	0	(224,2200)	(220,2200)	531

Experimental *AP* values at L1

n	(thr_{CPU}, n_{CPU})	n_{GPU}	(thr_{MIC_0}, n_{MIC_0})	(thr_{MIC_1}, n_{MIC_1})	GFLOPS
1500	(12,300)	0	(112,900)	(108,300)	227
3500	(9,700)	0	(216,1750)	(216,1050)	460
5500	(11,1100)	0	(224,2200)	(224,2200)	600

Contents

Conclusions

- Experimental results are satisfactory for the hybrid nodes considered, mainly for large problem sizes, where the performance achieved is close to the experimental optimum.

- The *AP* values obtained at Level 1 by each type of auto-optimization are similar to the experimental optimum, but the installation time used is lower.

- The hierarchical approach allows to extend the autotuning process to other levels of the heterogeneous platform.

Contents

Working Progress…

	Selected *AP* based on L1 installation						
n	*marte*	*mercurio*	*saturno*	*jupiter*	*venus*	*GFLOPS*	*Without_Comms*
1500	300	450	300	300	150	11	19
2500	250	500	500	750	500	16	88
3500	350	350	700	1400	700	24	179
4500	450	450	900	1350	1350	30	349
5500	550	550	1100	1650	1650	38	516
6500	650	650	1300	1950	1950	44	689
7500	750	750	1500	2250	2250	50	765

	Experimental AP values at L2					
n	*marte*	*mercurio*	*saturno*	*jupiter*	*venus*	*GFLOPS*
1500	0	0	1500	0	0	63
2500	0	0	0	2500	0	116
3500	0	0	0	3500	0	161
4500	0	0	0	4500	0	206
5500	0	0	0	5500	0	235
6500	0	0	0	6500	0	285
7500	0	0	0	7500	0	320

Future Research

- Develop empirical modelling techniques for each hierarchy level to model the communication time inside each node and between nodes .
- Use the auto-optimized matrix multiplication routine in higher-level linear algebra routines, such as LU, QR or Cholesky factorizations.
- Apply other installation techniques, such as a guided search for *AP* values, in order to reduce the installation time.
- Extend the autotuning methodology to select the most appropriate linear algebra library (MKL, CUBLAS, MAGMA…) for each processing element.
- Adapt the autotuning methodology to support bi-objective optimization on heterogeneous clusters for performance and energy.

Thanks for your Attention