Co-funded by the Horizon 2020 Framework Programme of the European Union

Grant agreement No. 671555

ExCAPE
Exascale Compound Activity Prediction Engine

Future and Emerging Technologies (FET)

Call: H2020-FETHPC-2014
Topic: FETHPC-1-2014
Type of action: RIA

Deliverable D2.10

Report: Simulation report 2

SMURFF compute performance analysis

Due date of deliverable: 31.08.2017
Actual submission date: 03.09.2018

Start date of Project: 1.9.2015 Duration: 36 months

Responsible Consortium Partner:	Intel
Contributing Consortium Partners:	Intel
Name of author(s) and contributor(s):	Yves Vandriessche (Intel)
Internal Reviewer(s):	Tom Vander Aa (IMEC), Jiří Dvorský (IT4I)
Revision:	V2

Project co-funded by the European Commission within Horizon 2020 Framework Programme (2014-2020)		
Dissemination Level		
PU	Public	PU
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Document revision tracking

This page is used to follow the deliverable production from its first version until it has been reviewed by the assessment team. Please give details in the table below about successive releases.

Release number	Date	Reason of this release and/or validation	Dissemination of this release (task level, WP/ST level, Project Office Manager, Industrial Steering Committee, etc)
0	05.05.2018	First draft, first SMURFF performance results	CO
1	26.07.2018	second draft, performance post-SMURFF changes	CO
2	03.09.2018	Incorporated reviewer feedback, fixing front matter	PU

Glossary

No glossary is given.

Link to Tasks

Task number	Work from task carried out	Deviations from task technical content, with motivation and summary of impact
2.4.3	Changes were made based on the results of this report under *Task 2.4.3 Optimization of machine learning programs to benefit from specific HPC resources*	None
2.5	This document reports on the work and results of *Task 2.5 Detailed Performance Analysis of Implementations*	None

Contents

1 Introduction **5**

2 Performance analysis setup **5**

2.1 Software Setup 5

©ExCAPE

2.2 Hardware Setup 6

3 Performance results and feedback **6**

3.1 Death by a thousand cuts 6

3.2 Hotspots 7

4 Improved SMURFF impact and results **7**

4.1 Benchmark results 7

4.2 Advanced and Latest SMURFF performance evolution 8

5 Conclusion **8**

©ExCAPE

Executive Summary

We present a node-level performance benchmark of the SMURFF framework and an analysis of the performance defects it helped detect. Here we show the immediate results from fixing the defects, but also show a longer term performance evolution. We end with a software engineering recommendation to help maintain and improve compute performance as a feature.

The submission of this report was held back in order to include benchmark performance results of more recent versions of SMURFF.

Dissemination and Exploitation

This report is public. The results of this report have not been published or otherwise communicated other than in this report.

©ExCAPE

1 Introduction

The purpose of this document is twofold. First it is a report on the performance issues and bottlenecks of the SMURFF software framework, as described in deliverable D2.7 [1]. These issues are the result of a SMURFF performance analysis using commonly available performance analysis tools, but also using the SNIPER proprietary architectural simulator. This performance analysis has been communicated and discussed in-depth with the SMURFF developer (Tom Vander Aa, WP2, IMEC), who subsequently addressed the uncovered issues in later versions.

The second purpose of this document is to give an overview of the performance evolution of SMURFF during the latter part of its continuous development life cycle. Chiefly, we focus on the impact of the above performance issue report, comparing SMURFF before and after the related changes. To this end, we compare the results of four different SMURFF versions to the same performance benchmark.

This report is structured as follows. In Section 2 we describe the performance setup. The results of the analysis is described in Section 3. Section 4 describes the effect on performance of SMURFF after addressing the performance issues uncovered by the anlysis. Section 5 formulates the overall and longer term conclusions.

2 Performance analysis setup

In this section we describe what hardware and software setup we used to do the performance analysis.

2.1 Software Setup

To cover the entire SMURFF framework, we set up the benchmark to contain a set of different methods, datasets and impactful parameters. Our SMURFF performance benchmark consists of the BPMF and MACAU methods, with the MACAU methods being subdivided into the direct- and CG-solver variants. For the MACAU methods, we also varied the side-information datasets: both sparse and dense ECFP6, and a dense Chem2Vec descriptor. All data originates from the ExCAPE project. This yielded the following five SMURFF configurations for our benchmark:

- *c2v* : MACAU with direct solver and dense Chem2Vec data.
- *sparse-cg* : MACAU with CG-solver and sparse ECFP data.
- *sparse-direct* : MACAU with direct solver and sparse ECFP data.
- *dense-direct* : MACAU with direct solver and dense ECFP data.
- *bpmf* : BPMF method, no side information.

Four different versions of SMURFF were benchmarked.

- *Round 1:* Baseline, for compute performance analysis (`v0.11`).
- *Round 2:* Improved, post-fix (`v0.11`).
- *Round 3:* Advanced, next point release (`v0.12`).

©ExCAPE

- *Round 4:* Latest, most recent (`v0.13`).

The difference between The Baseline and Improved version focuses on the changes made after the performance analysis. The Advanced and Latest point releases contain high number and wide ranging changes: 90 committed changes between Improved and Advanced, 320 between Advanced and Latest. Each SMURFF version retrieved from the ExCAPE source code repository history. Configuration files could not be shared across point releases. Instead, verbose SMURFF output and small configuration changes were used to ensure equivalent workloads across all four rounds and five configurations.

2.2 Hardware Setup

The benchmarking platform was an Intel Xeon 8170 (Skylake) platform. This platform is representative for current-generation compute nodes found in both cloud- and HPC-compute environments. An individual Xeon 8170 has 26 compute cores and 36MB of combined L2 cache. Counting all cores including HyperThreading yields 52 hardware threads.

Note that we deliberately avoid Non-Uniform Memory Architecture (NUMA) performance issues by restricting execution to a single processor. In practice, compute nodes often carry two processors. Launching SMURFF across such a system could give rise to detrimental NUMA effects. Here we limit our benchmarking platform to a single platform to avoid NUMA effects from hiding other performance effects.

3 Performance results and feedback

Speaking in broad terms, we found two categories of performance defects: *hotspots* and *death by a thousand cuts*. A hotspot is found easily with simple profiling: a hotspot function or code fragment dominates the execution time, pointing to a possible performance problem. A *death by a thousand cuts* defect adds an execution penalty across the entire program execution and acts as a multiplier. We first summarize the latter category of defects found in the Baseline SMURFF variant.

3.1 Death by a thousand cuts

There are two small performance defects that can be found across all code paths of both Macau and BPMF methods, although the magnitude of their effect varies. The first and simplest defect is the frequent use of a datastructure access method with bounds checking: the C++ STL `at()` method. As this function is used in the inner loops of the SMURFF methods, it is invoked extremely frequently and the bounds checking has a measurable effect. The Improved version of SMURFF moved where appropriately to a `operator[]` access method, without bounds checking.

The second *death by a thousand cuts* defect is caused the combination of multi-threading and an overuse of managed pointers, specifically the C++ STL `shared_ptr`. Passing such pointer through a chain of function invocations adds a small overhead. This overhead is caused by reference counting bookkeeping and very small, usually a simple increment or decrement. When passing such reference-counted pointer in a inner-loop call chains (e.g. `Model::U()`), this bookkeeping becomes significant. While not an important performance problem on its own, the fact that many threads are involved in SMURFF

©ExCAPE

that turns these managed pointers into a performance bottleneck. C++ STL protects the managed pointers from races by using atomic operations: `atomic_add`, `atomic_exchange`, etc. We detected a high amount of contention on these operations, threads were spending a great deal of time trying to concurrently access and change these managed pointers. The Improved SMURFF version changed where appropriate to the use of regular *flat* pointers.

3.2 Hotspots

The hotspots differ across SMURFF methods and side information data types. However, the underlying reason for each is the same: a sequential bottleneck.

The *dense-direct* benchmark has a significant sequential initialization time due to a large matrix-multiplication routine A^tA being performed sequentially though Eigen. This routine shows up again during the critical `sample_latents` routine, which injects a sequential part limiting the effectiveness of the parallel loop it is contained in.

The same `sample_latents` for `c2v` has a sequential section introduced by the `SolveInPlace` routine, which also uses Eigen and which is not parallelized. Another source of sequential parts is the construction and copying of large matrices by Eigen inside a parallel loop, such as for the construction of the K matrix in `sample_beta_direct` ($K = F^tF$).

The problem with large sequential parts is that even when invoked in a parallel loop, they introduce a significant load imbalance. Breaking large sequential parts up into smaller parallel tasks engages hardware threads that are otherwise idle. Several such hotspots were fixed in the Improved version through the use of the Intel MKL library where possible, and configuring Eigen to use MKL where appropriate. MKL internally parallelizes larger routines, which will provide a better load balance and thus CPU efficiency. The superfluous movement or construction of matrices is solved by moving them where possible outside inner loops to a higher enclosing scope.

4 Improved SMURFF impact and results

In this section we look at later versions of SMURFF and how the performance bottlenecks have been fixed in these later versions.

4.1 Benchmark results

The performance benchmark results in Figure 1 shows the impact of fixing the various performance defects reported in the previous section. The leftmost bars per benchmark shows the relative performance of the Improved version, which is the version that only includes changes made to the Baseline version to fix the performance issues mentioned above. In the case of the BPMF method, this relative performance gain is high, offering a $\times 4$ speedup. This is mostly attributed to the `shared_ptr` and `at()` changes; the BPMF method queries and updates its model very frequently, which multiplies any efficiency changes to that model access. The other models are more impacted by the method-specific hotspot changes, chiefly the changes to the Cholesky decomposition and avoiding superfluous movement of large matrices (specifically, the K matrix in the direct method). Interestingly, the same Cholesky changes that caused a good speedup for the *sparse-direct* benchmark caused a performance regression for *dense-direct*.

©ExCAPE

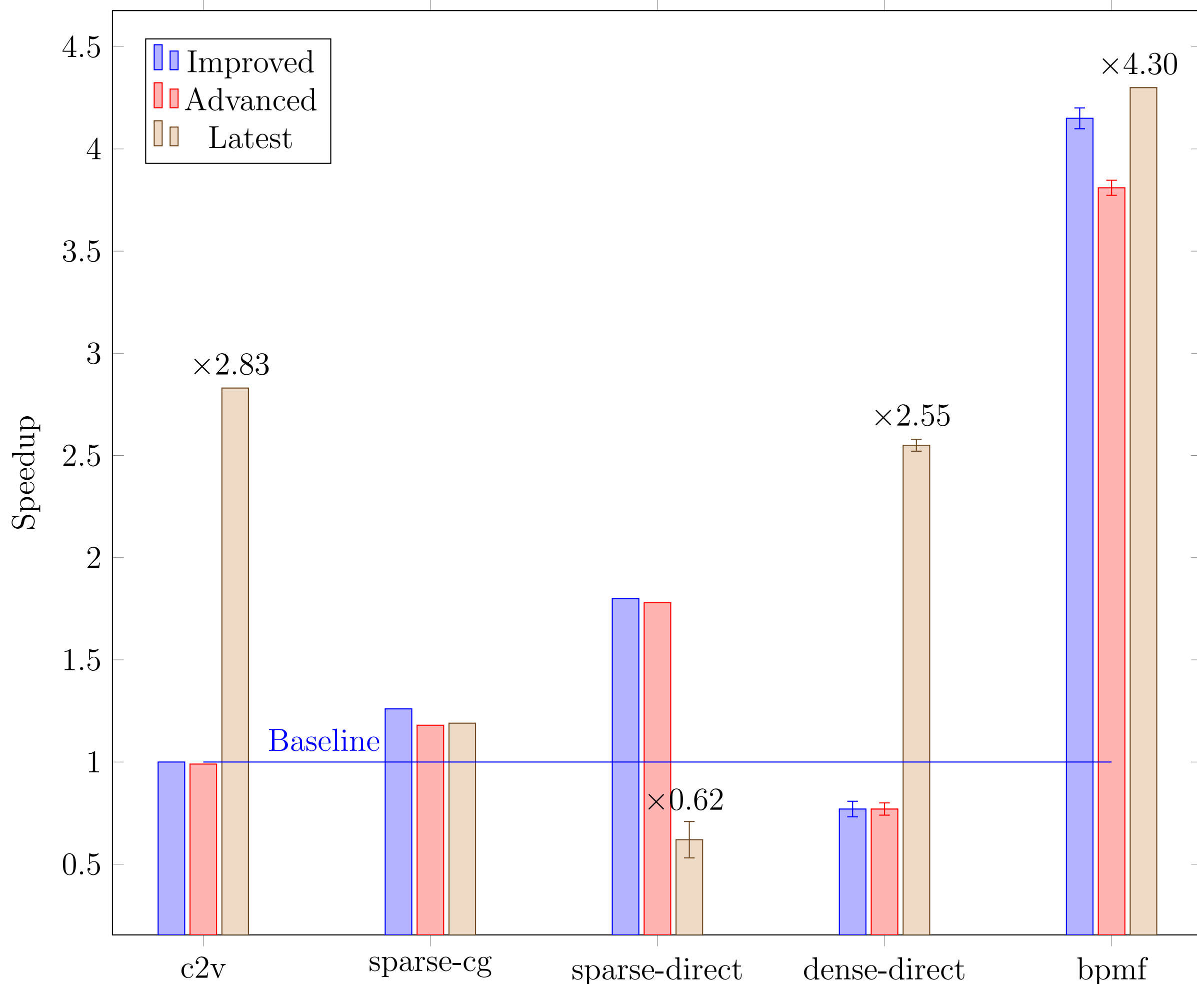

Figure 1: Average runtimes of the SMURFF benchmarks across multiple versions. Each bar shows the performance of a benchmark version relative to its Baseline version. Error bars show standard deviation over 2%.

4.2 Advanced and Latest SMURFF performance evolution

The results for the Advanced and Latest variants show the longer term performance evolution of the same benchmarks. Here we see that SMURFF Advanced (`v0.12`), stays close to the Improved version, but that changes made in the Latest (`v0.13`) version have have a great impact on the direct method. Closer analysis shows that the introduction of the Eigen LLT decomposition significantly sped up the dense side-effect benchmarks, *c2v* and *dense-direct*, but introduced a significant performance regression in the sparse side-effect dataset benchmark *sparse-direct*.

5 Conclusion

Performance analysis is similar to *bug analysis*. In this analogy, a programmer does not write code to only have it tested for bugs by a bug expert at the end of the project. Ideally, it is a continuous task of every involved developer, who continuously tests and fixes defects as they occur in their code changes. As we have seen here, small changes down the line can completely undo or obsolete conclusions of such analysis. A in-depth

©ExCAPE

performance analysis report is only a time-slice of a project that in practice continuously evolves.

The more changes happen between benchmark versions, the harder it is to detect the cause of performance problems. Performance analysis tools have become more powerful and easy to use, but such analysis remains time consuming.

Overall, SMURFF makes effective use of modern CPUs by making good use of parallel tasking facilities and optimized compute library routines. As has been shown, performance defects are easily introduced when evolving the code base. A history of frequent performance benchmark results helps to detect such performance regressions. More importantly, it allows to deduce what specific changes to the codebase introduced the performance regression. Our recommendation is to set up an automatic performance benchmark framework, analogous to a testing framework, to frequently and continuously check for changes to performance.

References

[1] Tom Vander Aa (Imec). ExCAPE deliverable D2.7: Other: Implementation 2. Technical report, 2017.

©ExCAPE