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 D2.6

Report: Simulation Report 1

Due date of deliverable: 28.02.2017
Actual submission date: 12.04.2017

Start date of Project: 1.9.2015 Duration: 36 months

Responsible Consortium Partner: INTEL
Contributing Consortium Partners: INTEL, IMEC
Name of author(s) and contributor(s): Yves Vandriessche (INTEL), Tom Vander Aa (IMEC)
Internal Reviewer(s): Kateřina Janurová (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	13.3.2017	First rough draft	CO
1	07.4.2017	Second draft, mostly text complete	CO
2	10.4.2017	Interally reviewed version	PU

Glossary

CG	Conjugate Gradient
COO	COOrdinate list sparse matrix format
CSR	Compressed Sparse Rows matrix format
ECFP	Extended Connectivity FingerPrint
HPC	High Performance Computing
ML	Machine Learning
SpMV	Sparse Matrix-Vector multiplication

Link to Tasks

Task number	Work from task carried out	Deviations from task technical content, with motivation and summary of impact
2.1.2	*Analysis of new machine learning algorithms* was required to link performance insight to elements of the algorithm.	None
2.1.1	*Analysis of standard machine learning algorithms* was required to performance insight to elements of the algorithm.	None
2.5.2	*Node performance simulation* results are reported throughout this report.	None
2.5.1	Node-level *scalability benchmarking* was used to analyse thread-level parallel performance	None
2.4.3	*Accelerator-specific optimizations* for SpMV operations were explored.	This work will appear in the *Simulation Report 2* deliverable.

2.4.2	Our SpMV and TensorFlow work are part of a future *reference implementation of new machine learning algorithms*	None
2.4.1	Our SpMV implementations are part of an existing *implementation of selected standard machine-learning algorithms*	None

Contents

Executive Summary

This document reports on the compute performance analysis activities of the ExCAPE project. Benchmarking and performance analysis tools are used to get insights on implementation efficiency and performance on CPU compute hardware. A proprietary processor simulator is used where deep and detailed performance metrics are required.

We report on the results and conclusions from continuous fast performance feedback of implementation prototypes from WP1 and WP2. Next, we deliver in-depth performance analysis and implementation exploration of two ExCAPE implementation domains: Sparse Matrix operations and Deep Learning.

The most prominent results are that:

- sparse matrix operations form the most important bottleneck across all ExCAPE algorithm implementations, ExCAPE-specialized routines have a significant positive impact,
- the Binet Deep Learning framework does not make sufficient use of available CPU resources,
- whereas the TensorFlow framework performs four times better on the ExCAPE learning pipeline, making it WP2's choice of high-performance Deep Learning implementation platform.

This deliverable was finished late because we wanted to take into account the latest simulation results.

Dissemination and Exploitation

The workloads and performance simulation results were disseminated within the Intel group with the aim of benchmarking future hardware and software solutions. There is a direct exploitation of the optimized binary sparse matrix-vector multiplication routines developed for the project and as discussed in Section 3, binary SpMV is in the process of being included in the Intel® MKL software [14].

1 Introduction

The purpose of this report is to document and elaborate on the various performance analysis work performed by WP2 during the project. The primary tool used here for performance analysis is the Sniper [4] simulator, an Intel-internal processor architecture simulator.

This document is organized into a fast feedback and performance analysis section. These two sections draw a line between two types of performance reporting. Performance analysis can be thorough and extensive, leveraging slow but accurate simulation techniques and multiple investigations to pinpoint problem areas. Performance analysis can also be preliminary, exploratory, highly contextual, or steer an implementation in progress. We therefor use the first section as an activities report of our fast-feedback activities, documenting various results obtained while in a close feedback-loop collaboration with the other partners of the project. We do not go in great detail in the first section, as many of the conclusions and results no longer hold, due to the fast and changing nature of their subject.

In the second section, we deal with performance results of implementations that are more stable and established, such that the conclusions can be safely reported in detail here. In the performance analysis section, the focus is on a more detailed and methodical analysis.

2 Fast Feedback

The subsections below document the fast-feedback optimization performed on the BPMF, SGD and CG codes in ExCAPE.

2.1 BPMF

Bayesian Probabilistic Matrix Factorization or BPMF was the first high-performance algorithm implementation available in the project [20]. This was due to its initial development in the related EC FP7 project: EXA2CT [1]. We gave preliminary performance results on this implementation during the first months of the project to the partner in charge of the project: IMEC.

We provided performance feedback results of matrix factorization on preliminary bioactivity data, for both a multi-core server CPU architecture (Xeon) and a next-generation many-core processor (*Xeon Phi x200*, formally known as Knight's Landing).

The simulated execution and execution profiling showed good vectorization, instruction-level parallelism, cache behavior and low idling or lock-contention. However, we did identify a disproportionate amount of time being spent in the Mersenne-twister Random Number Generator (RNG). Fixing the RNG-use yielded an 8x speedup.

Once the random number bottleneck was removed, the majority of CPU cycles were spent inside the Eigen C++ linear algebra library. Eigen is an open source high-performance linear algebra library, and has seen significant performance improvements by its community. As such, the current BPMF implementation has no easy performance low-hanging fruits left. The performance analysis and improvement on Eigen has, at this point, a low return on investment.

In conclusion, the BPMF demonstrates highly effective use of modern multi- and many-core processors. The recommendation is to focus further optimization efforts to

domain-level and library-level changes.

2.2 SGD

Stochastic Gradient Descent [16] is a ubiquitous optimization algorithm in modern Machine Learning. As such, we benchmarked a reference SGD implementation to guide future implementation efforts. The gold standard reference code for SGD that we tested was written by Léon Bottou, the academic author that originally presented SGD for Machine Learning [3].

Simple profiling quickly revealed that 95% of the execution time was spent in an unoptimized custom vector-math library. This library's low performance was readily apparent.

Our offered conclusion is to not use this code base for experimenting with performance improvements for SGD. Any algorithmic or performance improvements would be hidden by the poor vector-library implementation.

2.3 CG

MACAU [17], a Bayesian factorization algorithm related to BPMF, uses a Conjugate Gradient (CG) solver to find latent variables, before handing these to the learner algorithm. The CG solver forms the main computational bottleneck of MACAU, taking up a ration 50:1 execution time compared to the learners.

Our preliminary profiling results showed that the main culprit was the repeated sparse matrix-vector multiplications (SpMV) performed at each CG iteration step. The MACAU author isolated the SpMV routines in a separate library, which is the focus of a more in-depth analysis in Section 3. Early simulations revealed the extreme sparsity of the data to be the most important factor. The distribution of non-zeroes in the (6M by 2M) ECFP matrix defeats the caching and prefetching mechanisms in the CPU by accessing the input vector using large and highly random strides.

An interesting observation was that for the ECFP data matrix, the next-generation many-core processor *x200 Xeon Phi* could not leverage it's higher memory bandwidth into higher SpMV performance. Comparing simulated execution on both architectures revealed that both were showing a high degree of L1- and L2-cache misses due to the irregular access pattern. However, whereas the Xeon Phi had to fetch every L2-miss from its in-package DRAM, roughly half of the L2-misses on the Xeon processor could be found in its large L3-cache. As L3-cache provides a higher bandwidth, the Xeon remained competitive with the Xeon Phi. Lowering the size of the ECFP data matrix, viz. through feature selection, can make the input vector fit completely in L2-cache.

The conclusion of the initial MACAU-experiment was to suggest usage-specific custom SpMV routine improvements. Another feedback was related to the data organization. Given that the non-zero structure of the ECFP matrix dictates the access pattern, functionally-equivalent permutations exist of that matrix that offers a better access pattern. Note that this feedback is true for any algorithm using the ECFP data, not just MACAU. We have suggested to WP3 partners the use of matrix-partitioning algorithms from literature (Metis [15], Mondriaan [21]). This trades off paying a large up-front repartitioning cost with faster handling of the sparse matrix algebra across all learners.

3 Analysis: Extremely sparse linear algebra

The in-depth performance analysis effort started with the optimization of a class of routines that is ubiquitous for all involved Machine Learning algorithms on ExCAPE-like data: sparse linear algebra. The importance of sparse matrix operations quickly became clear during the initial analysis of the new and existing and ML algorithms [5]. In many cases, these operations dominate the runtime of the computation, which means that even small optimizations translate into large efficiency and performance gains. Here, we report on an exploration of various SpMV routine implementations, which we developed and benchmarked specifically for the ExCAPE data and algorithms. A detailed performance-impact analysis on the new ExCAPE ML algorithms is left for the second performance report deliverable.

Much effort has been put towards optimizing dense linear algebra routines, leading to highly tuned libraries such as CBLAS [7], Intel MKL [14] or Eigen [11]. Fewer such optimized sparse linear algebra libraries exist, e.g. Intel MKL SparseBLAS, SuiteSparse. In practice, high-performance codes often rely on custom implementations, in order to take advantage of the (domain-)specific sparsity structure of the data. The importance of such sparse algebra codes is only increasing, as the gap between compute and memory speeds, the "memory wall", continues to widen. This increased importance is reflected in the HPC community with the rise of the HPCG [6] benchmark as a complement to LINPACK [8] to rank supercomputers.

The ML algorithm implementations in ExCAPE, as are nearly all scientific codes, rely on a bedrock of low-level linear algebra routines (e.g. BLAS implementations or Eigen). Much effort has been put towards optimizing and tuning these ubiquitous routines. However, there is an important difference between sparse and dense linear algebra on modern hardware. The performance of sparse routines is determined by the nature of the data, which makes it impossible to make a routine that is optimal in general.

Fundamentally, both dense and sparse routines implement the same mathematical operations, however they differ greatly in how data is treated. In modern computing hardware, performance is increasingly determined by data access pattern, viz. the "memory wall", rather than raw floating point operations (FLOPS) potential. Dense routines have fixed and predictable data-flow, which plays well with modern deep cache hierarchies and hardware prefetchers. In contrast, the data-flow on sparse data is completely determined by the nature and properties of the data itself. For instance, a matrix-vector multiplication involving a large matrix with a uniform spread of non-zeroes will be a magnitude slower compared to a matrix with the same non-zeroes on the diagonal[1].

Sparse matrix-vector was identified as a key bottleneck low-level operation is the ExCAPE ML-algorithms, first identified by analysis [5] and later corroborated by profiling the available codes. The two main workloads we are analyzing are matrix-factorization and neural network training on ExCAPE's activity and ECFP fingerprint data.

3.1 Sparse matrix representation

The core difference between a sparse and dense matrix lies in its representation. A sparse matrix is represented by a sequence of $(\text{row}, \text{col}, \text{value})$ tuples, whereas a dense matrix

[1] Take A a 1024×1024 identity-matrix. Multiplication with a random vector using NumPy on a Xeon E5-2699 clocks a best time over 1000 executions of 0.16 sec. A random permutation of the same matrix clocks in at 7.36 sec.

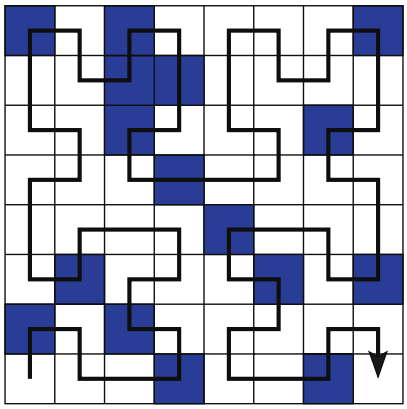

Figure 1: Hilbert-curve ordering of matrix non-zeroes. Figure reproduced from *Parallelism Pearls* [22].

lays out the values in memory in order of appearance in the matrix. By only storing the non-zero values, a sparse matrix can save storage and compute. The storage and compute requirement of a sparse matrix are in the order of the number of non-zero elements: $\mathcal{O}(\text{nnz})$.

The straightforward sparse matrix representation is a simple list of the (row, col, value) triples, the *list of list* representation. This format is often used during matrix construction, but it is not suitable for fast operations. A slightly faster basic representation is the Coordinate list (COO) sparse matrix. The COO representation uses three arrays to store the tuple elements as a structure-of-arrays, such that for a given non-zero index `i`, `rows[i]` and `cols[i]` contain the row and column index of the matrix element with value `values[i]`. COO does not improve on storage, but significant speedup can be gained by reordering the non-zeroes. This speedup is gained from improving the predictability and scope of the data access pattern. We will look at the SpMV performance of three different orderings: row-, column- and Hilbert-ordering. Hilbert-ordering is a linear ordering of the matrix elements by following a Hilbert space-filling curve over the matrix. In contrast to row- and column-ordering, Hilbert-ordering, visualized in Figure 1, improves access locality across *both* dimension [22].

Other sparse matrix formats attempt to leverage some property of the sparsity structure to improve storage and access pattern. Avoiding storing and accessing indices is an important optimization for routines that are memory-bound. CSR (shown in Figure 2) improves on COO by ordering the matrix entries by row and only storing at which index of the `values` array the next row starts. The more non-zeroes sit on the same row, the higher the compression factor. For SpMV access pattern, CSR enables a predictable access pattern on the destination vector (writes), but has an unpredictable access pattern on the input vector (reads). To improve on the input vector access pattern, we also selected the Column-blocked CSR (BCSR) representation for exploration. BCSR applies CSR to column-sliced sub-matrices. This improves cache-locality by limiting the access range to the input to a small column-range at a time.

Many more sparse matrix storage formats exist, differing in row- or column-ordered index compression or blocking strategies. For instance, the unique architecture of GPUs has given rise to a family of GPU-optimized packed and padded sparse matrix formats (e.g. ELLPACK [10]). Most of the research on sparse matrix algebra has been in the domain of scientific and engineering simulations: sparse solvers (e.g. in finite element analysis). As such, the type of matrices had an exploitable underlying structure, such as non-zeroes

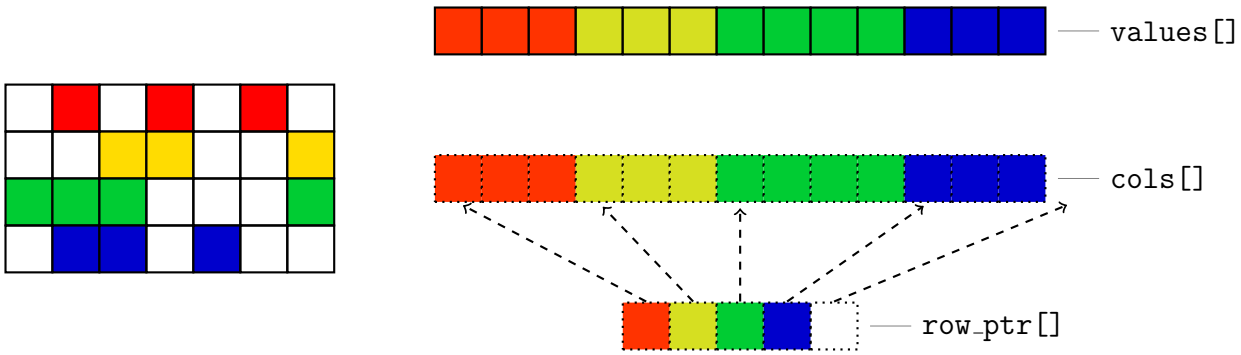

Figure 2: Visual representation of Compressed Sparse Row (CSR) matrix format.

appearing in clear clusters or blocks, or in diagonal bands across the matrix. Exploring the sparsity structure of the ExCAPE dataset, it quickly becomes clear that pharmaceutical real-world data does not exhibit such regular structures. ExCAPE-db data is both extremely sparse (less than 1 entry in 10000) and irregular, defeating sophisticated sparse matrix formats. For example, a state of the art blocked-matrix representation from literature yields blocks that on average contain only slightly more one non-zero element.

We make one important optimization in the above mentioned sparse matrix representation formats: we remove the `values` vector. ExCAPE-db's data can be represented as pattern- or binary-matrices, viz. where the non-zeroes have a Boolean value. As the stored values vector only contains '1' or '`True`' values in this case, it can be omitted altogether. The important effect of this optimization is to remove the fetch and multiplication of the matrix entry's value. We explore the effect of this binary-matrix optimization below.

3.2 SpMV performance exploration

Sparse matrix-vector routines are interesting architecture investigation test-bed. First, these routines are important; when they are used, they usually form the chief bottleneck. A small architectural change to improve SpMV can thus improve a number of valuable workloads. Secondly, the routines are small, straightforward and easy to understand; it is easier to link low-level hardware operations to the semantics in the high level code. Next, SpMV stresses the (micro-)architecture of the processor; structural hazards form a bottleneck (e.g. number of outstanding memory move instruction). And finally, there is no silver-bullet or optimal solution, a wide range of effective approaches, techniques and trade-offs are viable in different situations: cache blocking, prefetching, gather/scatter, vectorization, partitioning, etc. We have therefore analyzed the SpMV routines using our proprietary hardware simulator to reveal detailed effects otherwise not attainable using regular performance analysis tools.

The performance numbers shown for these experiments have been generated on an Intel Xeon E5-2699 v4 (Broadwell) system, using a single CPU with 22 cores (44 hardware threads). We have multithreaded these experiments where possible, but limit ourselves to one CPU socket to limit NUMA-effects in these micro-benchmark. All performance experiments calculate the Sparse Matrix-Vector multiplication

$$y = y + Ax \tag{1}$$

with A an ECFP sparse binary matrix of 169,674,021 non-zeroes with dimensions 1,807,972 (compounds) by 4,231,803 (features). The data type for the input and output vectors x

and y are doubles, making x 32MB fit in the processor's L3 cache (56MB), but not in its L2 cache (256KB). Note that performance will differ significantly from the results reported here in cases where x fits in L2 cache.

3.2.1 COO routines

SpMV has a straightforward sequential implementation for the Binary COO-matrix representation,

```
for (int i=0; i < nnz; i++)
  y[rows[i]] += x[cols[i]];
```

and a best-performing parallel implementation

```
#pragma omp parallel for schedule(guided,256)
  for (int i=0; i < nnz; i++)
#pragma omp atomic update
    y[rows[i]] += x[cols[i]];
```

which both work for whichever way the non-zero entries are sorted. We benchmark four different orderings: row- and column-index sorted, Hilbert-order and repartitioning-order. The first three are discussed earlier in this section. The repartitioning-order follows the non-zero reordering produces by the Mondriaan [21] matrix-repartitioning tool, which effectively reduces the aggregate distance between indices. The benchmark results, presented in Figure 3, show the Hilbert-ordering to be superior for both the sequential and parallel cases. This demonstrates the importance of improving data access locality. An interesting result is the row-ordering performing better for the parallel routine than the column-ordering, whereas in the sequential case they are reversed.

The parallel implementation of the COO-SpMV routine uses atomic operations to avoid read/write conflicts across threads. Atomic operations incur some small fixed overhead, but also a dynamic overhead depending on the presence of a read/write-conflicts. As can be seen on Figure 3, the performance of the COO-SpMV routines vary wildly depending on the ordering of non-zero elements. For instance, a row-sorted COO will cause a thread to see a smaller number of `rows[i]` values on average, which means that the read-update-write memory operation are more often local to the processor core hosting the thread. The col-sorted variant in contrast writes potentially across the entire range of the `y` vector, causing more frequent load-store conflicts across threads, with each conflict incurring expensive cache-coherency traffic across cores.

3.2.2 CSR routines

The Binary CSR-matrix implementation of SpMV,

```
#pragma omp parallel for schedule(guided, 64)
  for (int row = 0; row < nrow; row++)
    for (int i = row_ptr[row]; i < row_ptr[row + 1]; i++)
      y[row] += x[cols[i]];
```

removes the indirect access in the rows-direction, iterating instead explicitly over row index. By parallelizing over rows, we can eliminate the expensive store conflicts of the COO-variant. CSR also enables a further optimization,

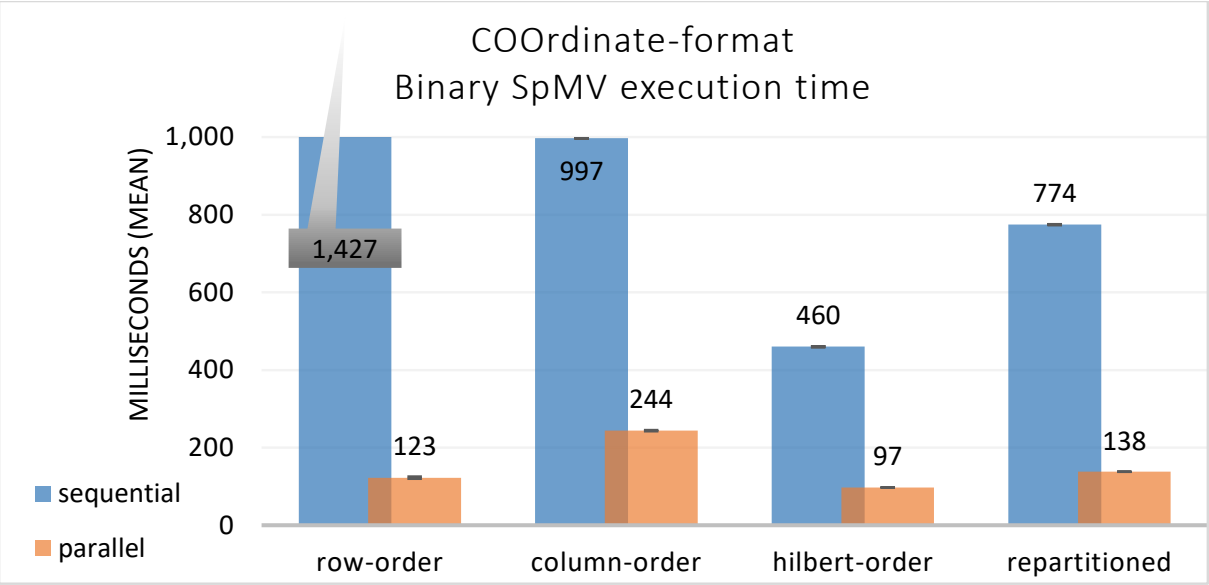

Figure 3: Side by side execution times of the sequential and parallel SpMV routines for COO-format matrix representations, using four different ordering of non-zero elements. A confidence interval of 95% is shown.

```
#pragma omp parallel for schedule(guided, 64)
  for (int row = 0; row < nrow; row++) {
    double sum = y[row];
#pragma omp simd reduction(+:sum)
    for (int i = row_ptr[row]; i < row_ptr[row + 1]; i++)
      sum += x[cols[i]];
    y[row] = sum;
```

which uses a local variable to accumulate the row sum. This ensures that a register can be used for the sum and guarantees that each write to an element of the `y` vector happens only once. The reduction pragma gives an additional hint to the compiler that enables vectorization of the entire loop[2]. As performance baseline, we use the SparseBLAS standard (non-binary) CSR general matrix-vector multiplication routine provided by Intel MKL: `mkl_dcsrmv()`. Comparing the benchmark results in Figure 4 reveals a 2.26× improvement of our best binary CSR routine over the MKL baseline.

Both COO and CSR variants of the SpMV are best parallelized using an automatic task-scheduling technique. The ExCAPE-db data has an extremely uneven non-zero distribution. Using a static scheduling technique, viz. omitting the `schedule` term in `parallel for`, distributes the work unevenly among threads, causing a large imbalance that reduces parallel efficiency. A dynamic distribution of work is crucial when parallelizing operations on ExCAPE's dataset[3]. All SpMV experiments here were replicated using CilkPlus and IntelTBB, which are both built around workstealing scheduling. We only report the OpenMP results here as its scheduling overhead for work stealing (viz.

[2]This optimization has a greater effect on AVX-512pf enabled processors, such as future Xeons (Skylake) and Xeon Phi (KNL), as it enables the use of the vectorized gather and prefetching instructions (VPGATHER).

[3]The same observation on the importance of dynamic scheduling on the ExCAPE-db data was reported in the Matrix Factorization implementation [13].

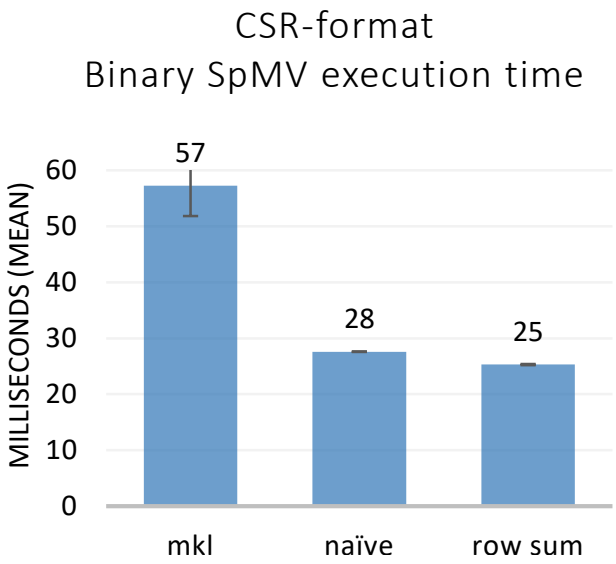

Figure 4: Execution times of our binary-optimized CSR SpMV routines compared to the baseline MKL `mkl_dcsrmv()` routine. A confidence interval of 95% is shown.

`schedule(guided)`) are measurably –albeit slightly– lower.

Company-internal performance simulations reveal the main bottleneck of the above CSR implementations to be the read-access operations on the input vector. As can be seen from the CSR code listing, each row-iteration leaps through column elements. The access pattern is indirect, meaning that no hardware prefetching can happen. Due to the non-zero distribution pattern, the access pattern also contains large strides, which is shown in simulations to cause a low (20%) cache efficiency.

3.2.3 Column-blocked CSR routines

Further, we also tested a column-blocked CSR variant to attempt to improve cache-locality and to increase the degree of parallelization. In column-blocked CSR, the original matrix is split into a set of matrices that each contain a subset of the original columns. Each such sub-matrix is stored in CSR format by concatenating their array representation. An extra `cell` array of offset indices is added to distinguish between sub-matrices in the blocked representation. The column-blocked CSR (BCSR) routine labeled `blocked naive` adds to the original an extra parallel iteration over the sub-matrices:

```
#pragma omp parallel {
  double ytmp[nrow] = {0};  // thread–local array
#pragma omp for schedule(guided) nowait
  for (int block = 0; block < nblocks; block++)
    for (int row = 0; row < nrow; row++) {
      int cell = block * nrow + row;
      double tmp = 0;
      for (int i = row_ptr[cell], end = row_ptr[cell+1]; i < end; i++)
        tmp += x[cols[i]];
      ytmp[row] += tmp;
    }
#pragma omp critical
  for (int row = 0; row < nrow; row++)
```

```
        y[row] += ytmp[row];
    }
```

such that each block-iteration is guaranteed to access only a limited range of the input vector, which limits the unpredictable read-accesses of `x[cols[i]]` in the fast L2 cache. However, this comes at the cost of each thread having to store a local temporary output vector `ytmp`. The `atomic` variant of this routine avoids the local vector in favor of atomic updates on the output vector. An improvement on this routine is to use thread-aware memory management, to avoid allocating and reducing a full `ytmp` array for each worker thread. For this, we use Intel TBB and Cilk++ Reducer objects [9]. In the above OpenMP implementation, no assumptions about the distribution of task over worker threads can be made, we thus need to conservatively assume the worst and allocate a full temporary output vector. Using a Reducer, in contrast, the necessary range of the `ytmp` array is created only when a worker thread steals the task that produces that range. We benchmark two Reducer strategies: a 2D-blocked Reducer in which tasks can steal in both row and column ranges and a row-blocked Reducer that only steals over rows. The 2D-blocked variant increases the amount of potential parallelism, as we parallelize potentially over both row- and column-dimensions. The row-blocked variant exhibits a lower overhead. From the benchmark results shown in Figure 5 can be concluded that

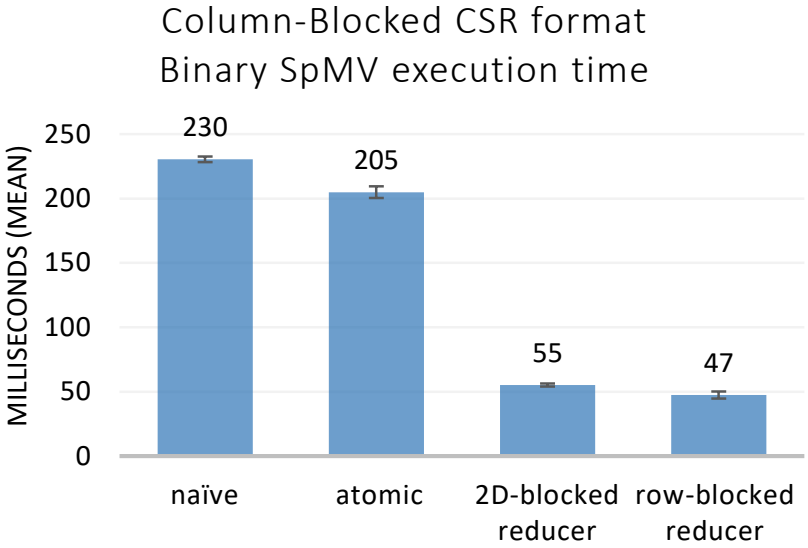

Figure 5: Execution times of the column-blocked CSR SpMV routines, showing with a 95% confidence interval.

the overhead of allocating, clearing and recombining a local output vector in the *naive* variant outweighs any cache-locality benefits from the column-blocked representation. The Reducer variants offer a better trade-off, with the 2D-blocking not being worth the overhead compared to only parallelizing over rows. However, the best column-blocked routine is still slower than the best routine for non-blocked CSR representation. Internal performance simulation reveals that blocking incurs a branch-prediction overhead at the processor front-end, which outweighs the improved cache efficiency. This is also explained by analysis of the non-zero distribution pattern; with a non-zero density of 10^{-6}, column block sizes that fit into L2-cache (e.g. 4096) contain no non-zeroes on most rows and only a handful in a few, highly populated rows are extremely scarce.

3.2.4 Batched & vectorized SpMV

Finally, we show that for certain use-cases we can gain another 2.36× amortized speedup over the best binary-CSR SpMV routine. In cases where the same matrix is multiplied many times with different input vectors, one can more efficiently calculate multiple SpMV operations simultaneously. This can be seen, for instance, in MACAU's CG-solver iterations (see Section 2.3). The vectorized routine, implemented as

```
#pragma omp parallel for schedule(guided, 64)
  for(int row_idx = 0; row_idx < nrow; row_idx++) {
    double sum[N] = {0};
    int row = row_idx * N;
#pragma omp reduction(+:sum[:])
    for (int i = row_ptr[row_idx]; i < row_ptr[row_idx + 1]; i++) {
      int col = col_idx[i] * N;
      sum[:] += X[col:N]; // Cilk++ array notation compiler extension
    }
    Y[row:N] = sum[:];
  }
```

"batches" the CSR-SpMV routine. In essence, this performs a matrix to thin-matrix multiplication routine. The benchmarked runtime for $N = 8$, on eight different vectors, yields a mean runtime of $85\,646\mu s$. Amortized, this results in a mean execution time of $10,705\mu s$, or 2.36× faster than our best CSR-SpMV routine.

The improved efficiency is not attributed to the enabled use of vectorization instructions. Indeed, simulations show that the CSR-SpMV variant already gets automatically vectorized by modern compilers on processors that support the AVX2 or AVX512 instructions. The improvement is attributed to an improved cache efficiency.

Each data-word accessed by the processor will cause an entire 64-byte wide cache line to be fetched from memory. Eight FP64-number fit inside such cache line. In the single-SpMV routine, the processor accesses a single element from the x-vector, for each non-zero in a row. Given the sparsity structure of the ECFP-matrix, there is an extremely low chance that multiple accesses into the x-vector are inside the same cache line, as the matrix non-zeroes are mostly non-consecutive. This translates into low cache efficiency and higher bandwidth use: only 1/8-th of the memory moved into cache is effectively used. The vectorized SpMV variant accesses an N-sized vector for each matrix non-zero. Taking $N = 8$ ensures that an entire cache line is accessed and effectively when accessing the x-vector. Note that this does not improve the unpredictable access pattern into the x-vector, but it helps to amortize the memory access cost.

3.3 Conclusion on SPMV

To summarize the optimization results on sparse linear algebra optimizations we can say that:

- Linear algebra optimizations on the CSR format performs best from those formats tested, but
- although the COO format is less efficient, it can still be a good choice in case the matrix contents is changing dynamically.

- When operating on multiple right-hand sides, we can obtain a proportional speedup due to vectorization;
- We can also obtain benefit from the fact that one of the main matrices in ExCAPE is binary.

4 Analysis: Deep Learning platforms

In this section, we report on the initial experiments on the Neural Network pipeline programming model as initially provided by WP1 [18]. We investigate the underlying execution fundamentals, but pay special attention to the performance and scalability of the neural network software framework implementation.

The field of Artificial Neural Networks (ANN) has become an important and permanent pillar of modern machine learning. In the context of ExCAPE, WP1 has been investigating a pipeline that trains a Multi-Layer Perceptron (MLP) neural network architecture on chemical activity data. This pipeline's goal is to predict compound-target activities based on chemical structure side-information (ECFP).

The WP1 research partner, Institute of Bioinformatics at JKU Linz, developed the ANN training pipeline using their purpose-built ANN software framework: Binet [19]. As Binet is purpose-built, it enables it to make several performance tweaks and to cut several corners that a general-purpose ANN framework cannot. Binet, being completely developed in-house in python, also allows them to quickly make changes to the framework's implementation. The first version of the ANN reference pipeline for ExCAPE was provided to WP2 in the form of a Binet implementation.

Last year has seen an intense evolution of software frameworks that support the construction and execution of ANNs. We selected two extra candidate frameworks: Google TensorFlow and Intel DAAL. Intel DAAL, as it is reported as the most efficient DNN execution framework for CPUs. Tensorflow, as it is a most popular and feature-rich framework at the time of this writing. We do not report on Intel DAAL, as during reconstruction of the ExCAPE pipeline revealed critical missing features. We do report here on the TensorFlow re-implementation and performance simulation.

We analyse the training performance of the ExCAPE Deep Learning network architecture on both Binet and TensorFlow platforms. Our setup hardware is a single compute node consisting of two Intel Xeon E5-2699 v4 processors. For a fair framework comparison, we design the workload to be a single training epoch of the ExCAPE network: all mini-batch training iterations for a single fold and a single set of hyper-parameters. Data load and preparation times are excluded, only the mini-batch training iteration times are included in the analysis.

4.1 Binet

Binet was built around the NVIDIA CUDNN framework, but offers a CPU implementation using BLAS-routines. As CPUs form the bulk of the available compute resources in the ExCAPE project, we analyzed Binet's performance on CPU architectures. However, as we will show, Binet's CPU implementation has shortcomings that make it unsuited as Neural Network execution platform for WP2.

Binet is at its core not a parallel program, a Python program drives the execution of the network and farms compute-intensive work in each layer to either the GPU (via

CUDA/CUDNN) or CPU (via BLAS routines). Using IntelPython and Intel MKL, we offered a best-of-class and parallelized implementation of the involved BLAS routines.

The Binet execution of the workload clocks in at a *wallclock time* average of *1,339.1 seconds*, with an aggregated *CPU time* of 45,113s. However, several metrics reveal a very low parallel and CPU efficiency. The fraction of elapsed time spent in parallel regions account only for 233.1s or 14.4%, the rest of the time (1,106.0s) is spent in sequential sections or waiting for other tasks to finish. The Binet runtime is dominated, in terms of Amdahl's Law [2], by its sequential part.

4.2 TensorFlow

Due to the Binet efficiency issues, we re-implemented its neural net learning pipeline using the next candidate framework: Google TensorFlow (TF). TensorFlow is at its core a dataflow execution engine, operating on tensor data types, viz. n-dimensional arrays. This core is written in `C++` for efficiency reasons. The TensorFlow main interface, however, uses the Python language, for usability reasons. In other words, although users use Python to build neural nets in TensorFlow, the execution of that program is handled entirely in TensorFlow's `C++` execution engine. TensorFlow's CPU back-end relies on the Eigen library for its linear algebra operations. Specifically, TF uses a tensor operations extension to Eigen, which is a contribution to Eigen by the Google TensorFlow team.

The TensorFlow ExCAPE pipeline implementation, dubbed *ExNet*, is a Multi Layered Perceptron with three hidden layers, using RELU and dropout fully-connected layers with a sigmoid cross-entropy loss layer. This network trains a two-class multi-target classification. An important characteristic is that the network deals with sparse input and sparse (scarce) ground truth. To deal with the scarcity of compound-activity training data, the last layer collects relevant output weights using a gather operation, before calculating the sigmoid cross-entropy loss. During back-propagation, only the gathered weights have their gradients calculated. ExNet, to keep parity with Binet, uses the Momentum SGD optimizer.

A single epoch workload measures a wallclock time of 289 *seconds*, a 4.7× *improvement* over Binet. Most of these gains can be attributed to the multithreaded execution support of the TensorFlow execution engine. TensorFlow parallelizes not only inside operations as in Binet, but also across operations. We measure a CPU efficiency of 73%, compared to Binet's 6%. The majority of the low-level Eigen kernels operate at a favorable CPI (Cycles Per Instruction). Indeed the majority of the CPU cycles are spent on efficient dense linear algebra kernels. There are several operations that cause the CPU to run at 1/5-th of its maximum FLOPS potential. However, a deeper analysis of Sniper[4] simulation data of the workload reveals another source of inefficiency for TensorFlow.[5]

We break down the workload performance of a single mini-batch iteration by visualizing the CPU cycle and FLOP instruction utilization over time in Figure 6. These stacks show that the forward pass accounts for only about a quarter of the execution time, with the backward pass taking the rest. The last layer, which fully connects the last hidden layer to the output layer, takes the majority of the forward- and backward-pass execution

[4]Sniper is an Intel-proprietary architecture-level processor performance simulator. Its accuracy is within 10% of cycle-accurate simulators, but Sniper's simulation speed allows the simulation in practice of entire applications running on entire multi-core chips.

[5]Note that the following analysis omits a large amount of raw performance statistics, which are available on request. Eigen uses an expression-template programming model, which unlike BLAS-routines requires significant interpretation and source code analysis.

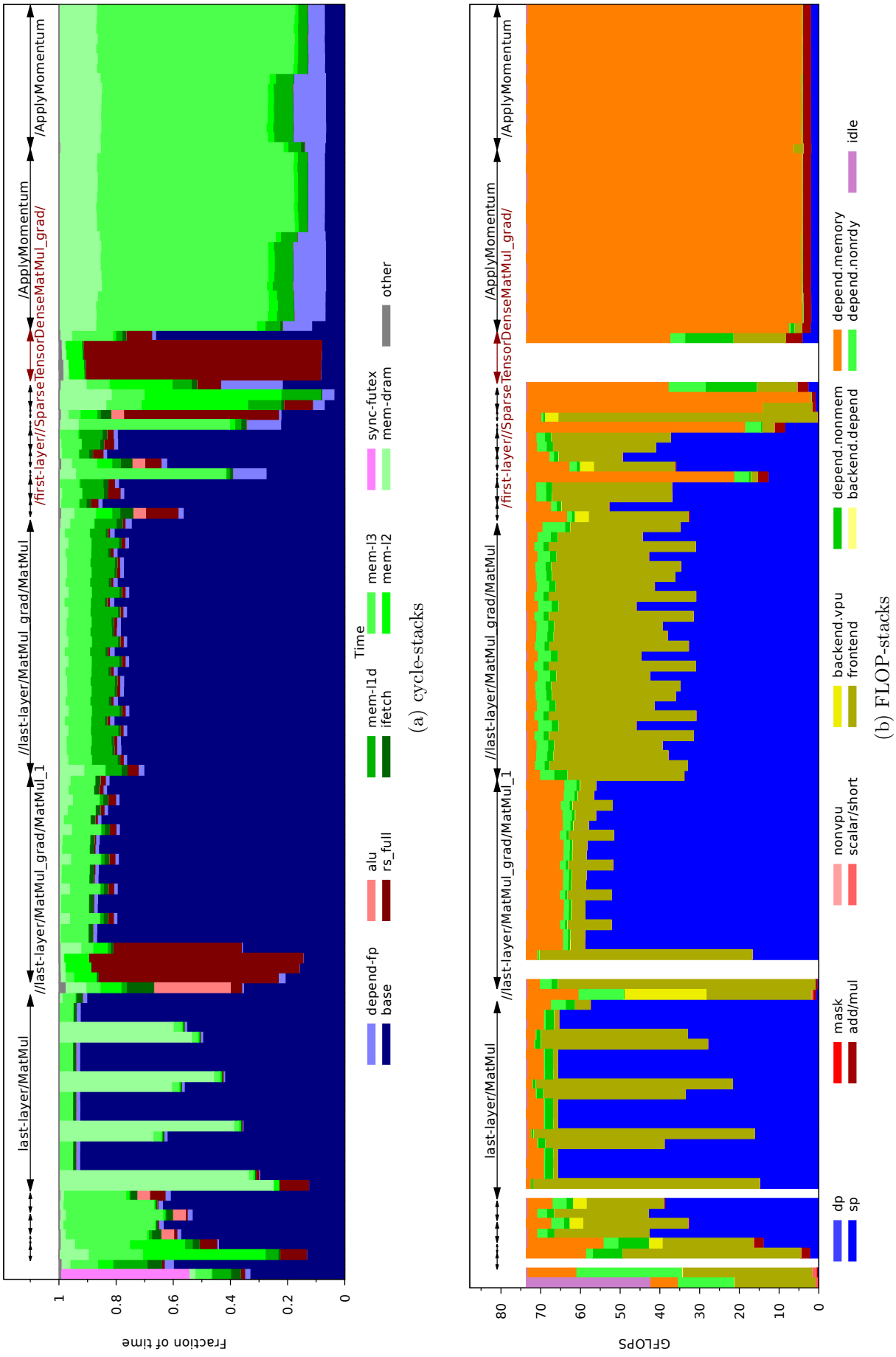

Figure 6: A visualization of the CPU utilization through time for the single-threaded execution of the TensorFlow operations in the ExNet network. TF operations are plotted as arrows at the top of the graph. The cycle-stack [12] in 6a visualizes whether CPU cycles were either useful (blue) or wasted (other colors). The FLOP-stack in 6b visualizes the achieved FLOPS rate (blue) and the reasons why the maximum-FLOP rate was not achieved (other colors).

time. This is expected, as the last layer contains roughly ten times more weights than a hidden layer. Even in the backward pass, the gradient calculation of the last layer takes a disproportionate amount of execution time. We also observe two phases dominated by structural hazards (`rs_full`, colored red), which by closer examination are caused by a high amount of load/store operations. The first such phase happens at the start of the last layer gradient calculation, the second at the end of the first layer gradient calculation; both are attributed to the gradient calculation of the fully connected sparse input and output layers (`SparseTensorDenseMatMul()`).

The majority of the execution time (62.31%) is taken by TensorFlow's matrix multiplication operations (`tensorflow::MatMulOp()`). These run efficiently at an average of two and a half instructions per CPU cycle (0.39CPI). Although the matrix multiplication routines are relatively efficient, we can observe a large amount of data movements. We expect these data movements to be generated by the n-dimensional block-panel tensor multiplication algorithm from `Eigen::unsupported::Tensor`. Substituting these with (size-)specialized matrix multiplication can be expected to further improve CPU efficacy. Even a small efficiency improvement can have an impact on wallclock time, due to the disproportionate amount of matrix multiplication operations involved.

In contrast, the `ApplyMomentumOp` executes only a small fraction (6%) of total instructions at a rate of nearly three cycles per instruction (2.67CPI), but takes nearly a third of the execution time (28.33%). The cause of this inefficiency is obvious from Figure 6: memory operations. The `ApplyMomentumOp` is in essence a simple component-wise kernel, calculating

```
accum = accum * momentum + gradient
  var = var - accum * learning_rate
```

where `momentum` and `learning_rate` are scalar constants. This kernel has low arithmetic intensity, performing four arithmetic over six memory operations. Such kernels are expected to be memory-bound. However, detailed statistics on the involved load and store instructions reveals that these are not delayed due to queuing, as one would expect when memory bandwidth is saturated. Instead, the `vmov` instructions are waiting for the data to arive. In other words, `ApplyMomentumOp` is memory *latency* bound. If the kernel were to be applied on a dense tensor, the linear memory access would make effective use of the caches through hardware prefetching. Instead, we observe nearly no L1 or L2 cache hits, which points to an unpredictable memory access pattern with large strides. We can conclude that `ApplyMomentumOp` is accessing the gradients and weights of the neural net model in a sparse way. Our recommendation is to introducing software prefetching to vastly improve the kernel's execution time.

5 Conclusions and Future Work

This document provided insight in the performance of several important ExCAPE computational kernels and applications via detailed performance analysis. This insight helped to optimize the matrix factorization and deep learning implementations in ExCAPE.

In the future we will look on how we can combine optimizations on the implementation with algorithmic changes to achieve even better results. For example, we will look at feature selection to reduce the memory footprint. A reduced memory footprint will increase the cache efficiency and hence the performance of a.o. the SpMV kernel in the different ExCAPE machine learning toolkits.

References

[1] EXascale Algorithms and Advanced Computational Techniques. http://www.exa2ct.eu/.

[2] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Reston, VA, 1967. AFIPS Press.

[3] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[4] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, New York, NY, USA, 2011. ACM.

[5] Imen Chakroun, Tom Vander Aa, Thomas J. Ashby (Imec), Jan Martinovič (IT4I), and Yves Vandriessche (Intel). ExCAPE deliverable D2.22: Analysis report 1. Technical report, 2016.

[6] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. High-performance conjugate-gradient benchmark. *Int. J. High Perform. Comput. Appl.*, 30(1):3–10, February 2016.

[7] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[8] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[9] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[10] Joseph L. Greathouse and Mayank Daga. Efficient sparse matrix-vector multiplication on gpus using the csr storage format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 769–780, Piscataway, NJ, USA, 2014. IEEE Press.

[11] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[12] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 38–49, Nov 2011.

[13] Tom Vander Aa (IMEC) and Yves Vandriessche (Intel). ExCAPE deliverable D2.4: Programming model 2. Technical report, 2016.

[14] Intel Corporation. *Intel math kernel library*, 2017. See `http://software.intel.com/en-us/articles/intel-mkl/`.

[15] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[16] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, Aug 2009.

[17] Jaak Simm, Adam Arany, Pooya Zakeri, Tom Haber, Jörg K. Wegner, Vladimir Chupakhin, Hugo Ceulemans, and Yves Moreau. Macau: Scalable bayesian multi-relational factorization with side information using MCMC, 2015.

[18] Paolo Toccaceli, Ilia Nouretdinov, Zhiyuan Luo, Vladimir Vovk, Alex Gammerman; Lars Carlsson; X. Qin, P. Blomstedt, E. Lepp aaho, P. Parviainen, and S.Kaski. ExCAPE deliverable D1.4: Other: Report + code 2. Technical report, 2016.

[19] Thomas Unterthiner, Andreas Mayr, Marvin Steijaert, Jörg K Wegner, Hugo Ceulemans, and Sepp Hochreiter. Deep learning as an opportunity in virtual screening. In *Deep Learning and Representation Learning Workshop (NIPS 2014)*, 2014.

[20] Tom Vander Aa, Imen Chakroun, and Tom Haber. Distributed bayesian probabilistic matrix factorization. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 346–349. IEEE, 2016.

[21] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[22] A. N. Yzelman, D. Roose, and K. Meerbergen. Sparse matrix-vector multiplication: parallelization and vectorization. In J. Reinders and J. Jeffers, editors, *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, chapter 27, page 20. Elsevier, 2014.