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 D2.15

Report: Simulation report 3

ExNET compute performance analysis

Due date of deliverable: 28.02.2018
Actual submission date: 20.08.2018

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):	Stanislav Böhm (IT4I), Jan Martinovič (IT4I), Tom Ashby (IMEC)
Revision:	V4

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	19.03.2018	First draft	CO
1	25.04.2018	Weak scaling experiment results	CO
2	23.07.2018	Added all graphs and rest of writeup, close to text complete	CO
3	25.07.2018	Text complete, reviewer copy	CO
4	20.08.2018	Reviewers inputs integrated	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
2.4.2	The subject of the performance analysis in this report is the implementation of ExNET that was direct result of *Task 2.4.2 Provide a highly scalable, reference implementation of new machine learning algorithms*	None
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

Executive Summary

This document reports on the compute performance of the *ExNET-v4* machine learning implementation that was described in deliverable D2.11 [2]. The focus here is on single node performance of representative ExNET workload for modern CPU and GPU compute cluster nodes. We observed that GPU node performance was better (4× faster) in a single model training scenario. However, the CPU node outperformed (2× faster) the GPU when training multiple models simultaneously.

This deliverable was delayed in order to report on a more representative version of ExNET: version 4. This was ExNET implementation version used by WP3 for their full-scale validation and benchmarking of the Neural Network learner.

Dissemination and Exploitation

The results and conclusions in this report were presented only internally to the consortium and within Intel. ExNET was bundled into an internal benchmark for use by Intel engineers working on linear algebra and machine learning performance.

1 ExNET benchmark

1.1 Introduction

The purpose of this report is to characterize the execution performance of ExNET in a typical model training scenario, where a large number of models are trained on the ExCAPEdb dataset folds. Due to hyperparameter search (model tuning), many different models get trained with the same training data. It is thus significant to look at the execution time of a single fold, but consider that many training instances will get run in large parallel batches. As multiple training instances are embarrassingly parallel, these scale well across a cluster. This scaling across a cluster is achieved using HyperLoom [1][4], for details on its scaling characteristics we refer to Scalability Report 1 [3].

Distributing a single ExNET training instance over multiple compute nodes is not considered here, as it is not a realistic scenario. The additional communication cost of networked weight updates needs to be amortized by the additional available compute, however as we show below the compute utilization for a single node is already low due to local memory bottlenecks. Indeed, multiple simultaneous training runs are already necessary to get the most out of a single compute node.

As with the previous Simulation Report [5], the node performance of ExNET and TensorFlow has been studied in greater detail using the Intel-internal SNIPER performance simulator. The performance analysis reported here are not the result of simulation, but are the results of direct execution on existing compute hardware. The additional details provided by the performance simulation are of a low-level hardware nature, sensitive information and are chiefly of interest to hardware architecture designs to guide design decisions of future compute architectures. The conclusions of these results do reflect the same findings of the proprietary performance analysis tools.

1.2 Setup

Hardware We characterize the performance of both CPU and GPU compute nodes of a type commonly found in HPC or cloud infrastructure. The CPU node is a dual-processor Skylake Xeon 8170, which hosts a total of 104 threads and 36MB caches. The GPU node in our setup uses a NVIDIA Pascal-architecture 1080Ti with 11GB GDDR5 memory, in tandem with a Skylake Xeon processor. Note that TensorFlow executes its training on CPU and GPU simultaneously.

Workload The execution times reported below are the total process execution time of a single-epoch training job, the training data is taken from ExCAPEdb and consists of 5,200 64-sample minibatch iterations. This execution time includes data input and preparation, as well as model initialization. Data preparation time is minimized by using a pre-processed binary input format. Training for only a single epoch is a a compromise to benchmarking practicality. Realistically, models are trained over 100≈500 epochs, which diminishes the contribution of data preparation and model initialization. A possible concern is that this sequential data preparation part is proportionally larger in our single-epoch benchmark than in a real training scenario. However, the preparation time in a single epoch scenario is already small relative to the total execution time.

Benchmark experiments: weak and strong scaling We investigate two types of parallel scaling on the CPU nodes. Strong scaling investigates the execution speedup gained when more parallel processing units are made available for the same problem. In other words, how much faster is the job finished with each extra worker. In our benchmark, we increase the number of CPU cores for the same training job.

Weak scaling investigates the execution speed when the problem size is increased. Concretely for this benchmark, we increase the number of model training jobs that are computed simultaneously. Note that we do not perform a textbook weak scaling experiment by increasing the available parallel resources proportionally to the problem size. Instead, this benchmark strikes a balance between weak and strong scaling by distributing all available compute resources evenly and fairly over the simultaneous training jobs. This way, this benchmark better represents the practical trade-offs of increasing the number of simultaneous training jobs, and help answer the question: "How many models should I train simultaneously per compute node?".

For the weak scaling ExNET for the CPU, we scale the number of cores that the process can use. This leads to better performance than by fixing the number of worker threads in TensorFlow directly. TensorFlow maintains two different worker thread pools with different scaling behaviors. These worker threads also are not pinned to compute cores, they can and do migrate over to different cores and even different processor. Indeed, running TensorFlow without restricting the compute cores it can use leads to sub-optimal performance. As such, a training job using the entire two-processor CPU benchmarking platform takes an average of 241 seconds, while the best average single-node performance of 189 seconds is reached when restricting TensorFlow to use the first 18 (out of 26×2) cores.

The strong scaling benchmark for CPU is scaled by launching multiple ExNET training processes. A Non-Uniform Memory Architecture or NUMA-aware round-robin allocation strategy is used to divide the available computation cores fairly across these processes. In other words, the more training processes are run simultaneously, the less hardware threads each individual process can use.

For the GPU, we omit the weak- and strong-scaling (thread-scaling) experiments. Strong-scaling experiments in context of the GPU programming model are both hard to implement and can produce inappropriate results. A GPU programmer has, by design, little to no control over thread scheduling.

The weak scaling experiments on GPU are omitted due to memory limitations; only a single ExNET full scale train job could be run within the 11GB GPU memory. ExNET is a relatively large model and thus has a relatively large working memory footprint, circa. 2.6GB under the ExCAPEdb experiments. While this fits multiple times in the 11GB budget, the working memory used while training is many times higher on GPU than CPU. This is due to the TensorFlow 'SparseTensor' datastructure and operations using a disproportionate memory footprint. The TensorFLow GPU implementation of sparse-matrix dense-vector multiplication operation converts parts of the sparse matrix to a dense representation, which quickly balloons the memory use when training a large set of features and targets, as is the case in ExNET with ExCAPEdb data. In other words, the current best throughput for training ExNET models on GPUs is achieved by training them one after the other.

In contrast, many CPU training jobs can run in parallel, both because they have a smaller working memory footprint and because CPU nodes typically carry more memory. On the flip side, as we will see, a single CPU train job does not sufficiently use the CPU node compute resources, such that multiple jobs are necessary to achieve optimal hardware utilization and throughput.

1.3 Results

Our benchmark experiments report on execution speed and parallel efficiency. The former tells us how fast the training job took from start to end, the latter how well additional parallel resources are utilized. In the strong scaling benchmark, the execution time for n worker threads (T_n) ideally should decrease proportionally with the number of workers. For the weak scaling experiments, we report *amortized* execution time, which reflects the increased throughput gained when executing a parallel batch of n training jobs. The ideal amortized execution time decreases proportionally with the number of threads.

CPU strong scaling Figure 1 shows results of scaling the number of hardware threads for a single-epoch training job. It is evident from this experiment that the parallel efficiency of the TensorFlow ExNET implementation is low, it does not efficiently make use of the extra compute threads made available by the hardware. At two threads, the efficiency already drops below 60%, a commonly used cutoff point. However, making additional threads available quickly limits the additional speedup gained, up to the point where additional threads slows down the time to solution.

In the training scenario where only a single model needs to be trained as quickly as possible on a CPU compute node, it is important to limit the number of threads to roughly half the available hardware cores on a single hardware core. For the more common training scenario where multiple models are trained, it can be beneficial to limit the number of cores per training job even further.

Vertical help lines on Figure 1 show where TensorFlow is being allocated threads from a single processor (1P), across two processors (2P) and when it starts relying on HyperThreading hardware threads (HT). This shows that additional threads slow down execution even before NUMA or HyperThreading effects come into play.

CPU weak scaling The strong scaling experiment above essentially demonstrates the time to solution or latency when training a TensorFlow ExNET model. The weak scaling experiment shown in Figure 2 in contrast is concerned with throughput. In practice it is throughput that typically matters most. Establishing a model involves doing an extensive hyper-parameter search with cross-validation: thousands of models are trained in parallel. This experiment helps to inform the decision of how many such training tasks should be scheduled simultaneously on compute nodes.

In a weak scaling experiment, the amount of work is scaled proportionally with the amount of workers. The execution time T_n is thus ideally a constant. For ease of interpretation, Figure 2 reports on the *amortized* execution time $\frac{T_n}{n}$ rather than just execution time T_n. That is, we average the execution time by the number of training tasks in the batch. This yields a per-task virtual parallel execution time: although an individual training task does not take less time to complete, we gain a parallel speedup by running multiple jobs side by side. An amortized execution time of e.g. $30s$ means that we achieve a throughput of 120 training tasks per hour.

Concretely, the graph shows that launching multiple independent TensorFlow execution tasks can make better use of the available CPU compute power. This contrasts the strong scaling experiment above, which shows that TensorFlow struggles to make use of additional CPU threads, which is crucial for modern processors to reach their compute potential.

Figure 2 also contrasts a single- and dual-processor system. Additional processors

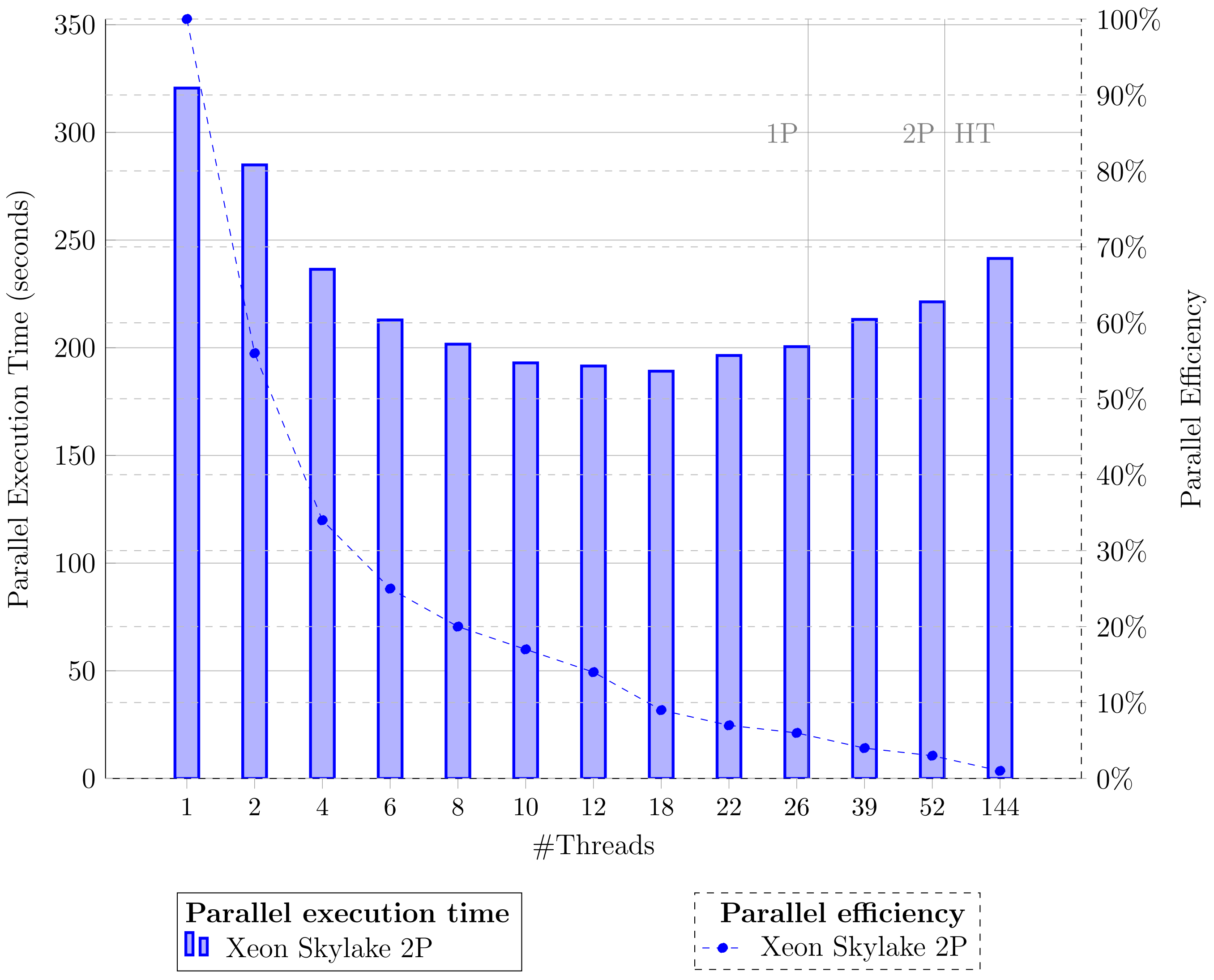

Figure 1: Strong scaling graph showing the execution time T_n of a single training task using n hardware threads. It also displays parallel efficiency $\frac{S}{n}$ with $S = \frac{T_{\text{seq}}}{T_n}$ the parallel speedup. Threads are allocated from a single processor first, before using threads across two processors and finally starting to use HyperThreading. These boundaries are denoted on the graph respectively with the 1P, 2P and HT boundary lines.

on a compute node not only add additional hardware threads to the system, they also multiply the available memory bandwidth. Evidence of this can be read from the parallel efficiency curve, which shows the Skylake Xeon 2P node reaching roughly double the efficiency for the same number of parallel jobs. It takes sixteen parallel training jobs to dip the parallel efficiency under 50% on the dual-socket system, where the single socket reaches this point at eight jobs. This indicates that scheduling independent training jobs in parallel successfully exploits this increase in available compute resources. In other words, giving a single TensorFlow execution job more compute resources is significantly less effective than running many independent TensorFlow jobs in parallel.

Comparing GPU and CPU results on the weak scaling experiment indicates that the sparse nature of the ExNET neural network architecture is less amenable to GPU acceleration than traditional deep convolutional neural networks. On paper, the NVIDIA 1080Ti GPU has a order of magnitude better theoretical compute power at 11.3TFLOPs, compared to 1.5TFLOPS of our testbed 1P Skylake Xeon. In terms of memory bandwidth, the 484GB/sec offered by the 1080Ti's GDDR5 memory is also greater than the theoretical 178GB bandwidth for the same CPU's six DDR4-2666 channels. The GPU result for a single train job indeed offer a 3× speedup over the CPU. However Figure 2 show that when the CPU is better utilized through parallel jobs, overcoming TensorFlow's thread-scaling limitations, the CPU platforms outperform the GPU on throughput. The previous simulation report [5] established that roughly half of the execution time is spent on the sparse layers in ExNET. Such sparse tensor operations are less suited to the GPU architecture, due to their unpredictable indirect memory access patterns and low arithmetic intensity (FLOPS/Byte).

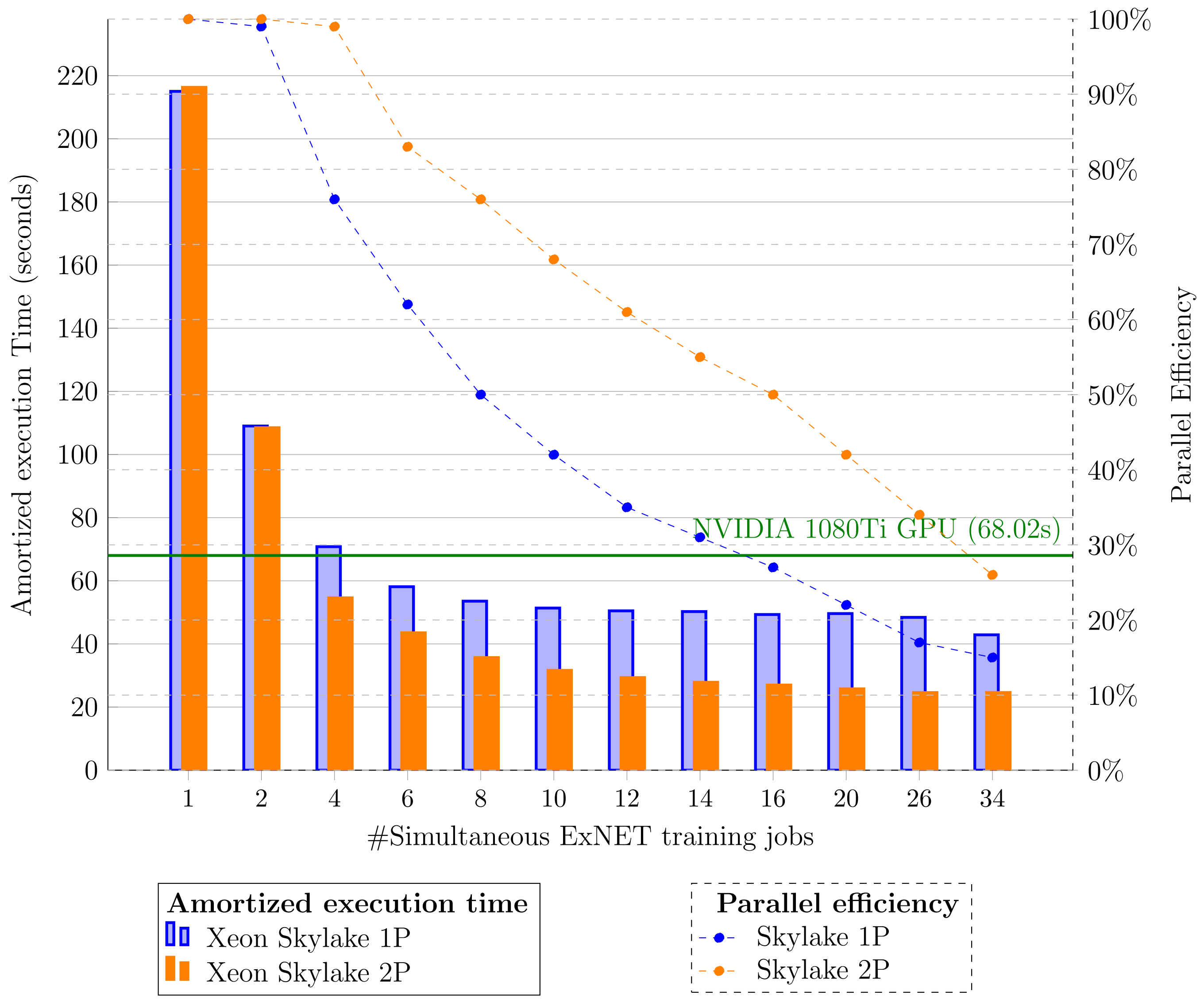

Figure 2: Weak scaling graph showing the performance behavior when scaling the problem size. The parallel execution time T_n is equivalent to the execution time of the slowest training job in the batch, the vertical axis reports on the amortized execution $\frac{T_n}{n}$ and parallel efficiency $\frac{S}{n}$ with $S = \frac{T_{\text{seq}}}{T_n}$. The Skylake 1P and 2P are the results for single- and dual-socket Intel Xeon server compute nodes. This graph shows that while the benchmarked GPU solves a single job faster, the CPU nodes provide a better throughput when training at least six models simultaneously.

2 Conclusion

To conclude, ExNET's TensorFlow implementation has certain properties inherent to its shallow-but-wide neural network architecture and sparse data handling that make CPUs an attractive compute platform. CPUs are at the time of this writing typically only considered competitive for neural network inference tasks, with GPUs being the accelerator of choice for model training. However, we show here that TensorFlow can make efficient use of the available CPU compute resources when multiple model training jobs are executed simultaneously.

Looking ahead, several neural network training accelerators are being developed and brought to market. As with GPUs, these accelerators focus on the arithmetically-intense deep learning networks, with special-purpose convolution accelerators and dense tensor memory stores. A network such as ExNET has different computational demands, leading us to believe that the CPU/GPU results here will still hold for the near future.

References

[1] Vojtech Cima et al. HyperLoom possibilities for executing scientific workflows on the cloud. In *Proceedings of the CISIS 2017 : The 11th International Conference on Complex, Intelligent, and Software Intensive Systems*, 2017.

[2] Yves Vandriessche (Intel). ExCAPE deliverable D2.11: Software: Implementation 3. Technical report, 2018.

[3] Vojtěch Cima (IT4I). ExCAPE deliverable D2.13: Scalability report 1. Technical report, 2018.

[4] Tom Vander Aa (Imec) and Yves Vandriessche (INTEL). ExCAPE deliverable D2.4: Programming model 2. Technical report, 2017.

[5] Yves Vandriessche and Tom Vander Aa. ExCAPE deliverable D2.6: Simulation report 1. Technical report, 2017.