Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Performance Portability of OpenCL with Application to Neural Networks

Jan Christian Meyer[a]* and Benjamin Adric Dunn[b]

[a] *High Performance Computing Section, IT Dept., NTNU*
[b] *Faculty of Medicine, Kavli Institute for Systems Neuroscience / Centre for Neural Computation, NTNU*

Abstract

This whitepaper investigates the parallel performance of a sample application that implements an approximate expectation-maximization method for inferring the network structure and time varying states of a hidden population within the framework of the kinetic Ising model. The size of networks that can yield informative results can be made arbitrarily large, and the long-running computational demand is highly localized, making the application a strong candidate for future exascale platforms.
Previous investigations using OpenMP on the Intel Xeon Phi architecture have suggested that the class of accelerator unit may play a significant part in attainable application performance. An OpenCL parallelization enables experiments with a variety of accelerator units. We examine how this programming model affects the performance of a portable implementation, and use it to compare accelerator technologies in terms of their suitability for future extreme-scale computations.

1. Introduction

Selecting an appropriate node architecture is essential to enable applications for extreme scale computing. This whitepaper studies the performance portability of the OpenCL programming model at the heterogeneous node level, with focus on evaluating performance using a model program which emulates the workload of a neural network application. Specifically, we implement an approximate expectation-maximization method for inferring the network structure and time varying states of a hidden population within the framework of the kinetic Ising model.

In neural computational research, the problem of estimating structures of hidden neural networks based on limited experimental data admits a wide range of computational techniques. From statistical physics, the kinetic Ising model presents an efficient, highly parallel method to analyse non-equilibrium systems. The utility of this model is related both to the size of network that can be represented, and the time required to infer its structure. Thus, the highly parallel nature of the problem makes it an attractive candidate for exascale computations, both in terms of access to greater memory resources, and obtaining results within a reasonable time frame. In a previous study, we investigated a proof-of-concept adaptation of this method to Intel Xeon Phi accelerators using OpenMP[1], and while we found that the computation scaled well with increasing thread counts, absolute performance was not competitive with that of conventional multi-core processors.

On the path toward applying the method to exascale problems, a key component of exploiting the inherent parallelism in the computation is to develop methods to evaluate candidate node architectures in terms of their

[a]* Corresponding author. *E-mail address*: Jan.Christian.Meyer@ntnu.no

application-specific performance. The OpenCL programming model [2] is suitable for this purpose, as it offers portability across a range of accelerator architectures, thereby reducing the need to produce highly customized tests for each candidate system.

We present experiments from three different heterogeneous node architectures with variable fitness for the task, and evaluate their applicability. Our method and findings are relevant to communities involved in systems benchmarking and dimensioning, such as the Performance optimization and Productivity (PoP) Centre of Excellence.

2. Model Problem

To investigate how the computational requirements of the problem interact with target architectures without imposing the constraints of particular problem data, we construct a model program with deterministic behaviour, and parameterize it with respect to the dimensions of the input data.

2.1. Program State Representation

Program state is encoded in a set of matrices representing a set of matrices dimensioned according to three problem specific parameters, K, N, and M, as shown in Figure 1.

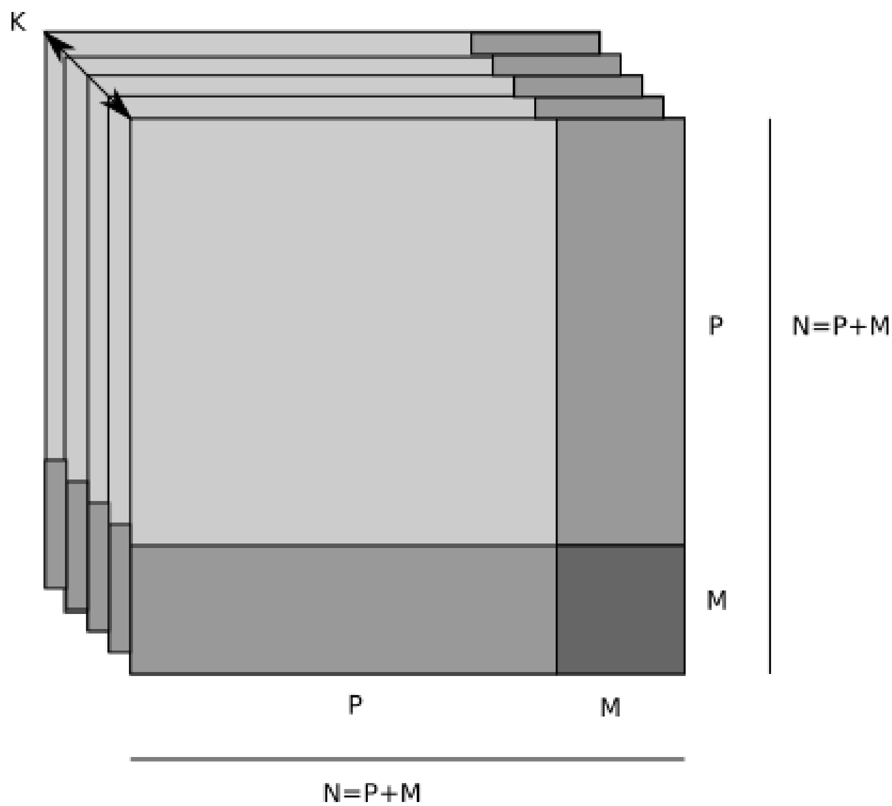

Figure 1. Data structure in dimensions K, N and M.

The four KxPxP, KxPxM, KxMxP and KxMxM submatrices are stored separately, and extruded along a fourth axis T, which can be expected to far exceed the other dimensions, and is modelled to have an exponential magnitude relative to the other dimensions, as shown in Figure 2.

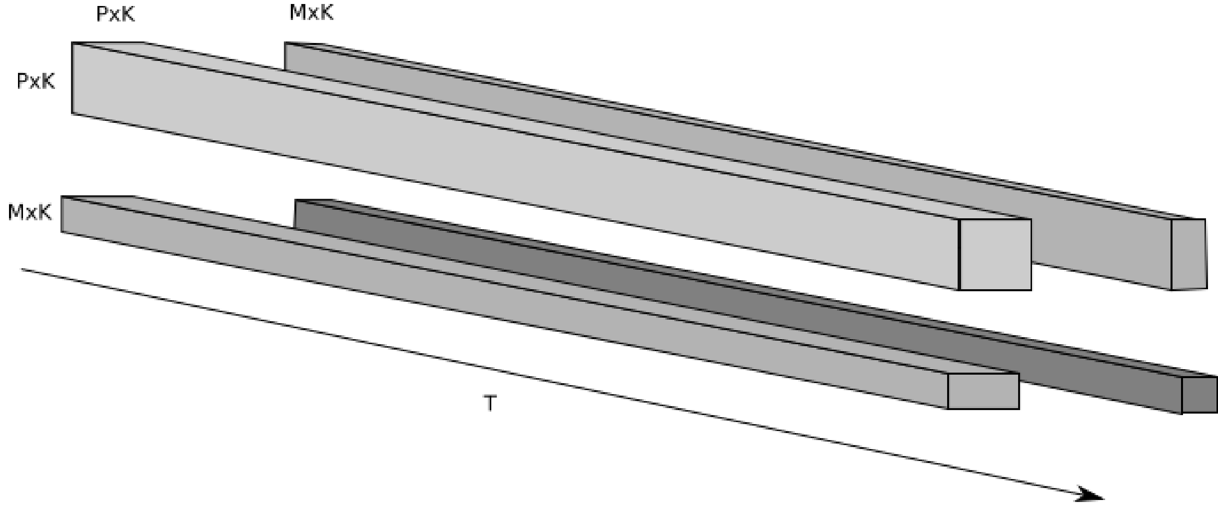

Figure 2. Regions extruded along T dimension. Note that the K dimension is collapsed for clarity.

The program predominantly operates by combining variables which represent various projections of this space, 12 of them proportional to KN^2, 3 proportional to KTN^2, and 13 proportional to NT. Sequential dependencies occur between element-wise combinations of these, with the exception of dependencies along the T axis that require a neighbourhood of K+1 points in both positive and negative directions. An example of one of the KTN^2 operations is presented below, displaying how the core computation consists of independent arithmetic combinations. The sole exception to this pattern is NT hyperbolic tangent values.

```
for ( int64_t k=0; k<K; k++ )
    for ( int64_t t=k; t<T; t++ )
        for ( int64_t y=0; y<(N-M); y++ )
            for ( int64_t x=0; x<M; x++ )
                Ho(y,t) += JouI(k,y,x) * (real_t)Mu(x,t-k);
```

This suggests that interference and synchronization requirements are minimal when parallelizing the computation, and the T axis is targeted for parallel execution as it admits the greatest task granularity. Such a partitioning over distributed memory would require 1-dimensional periodic border exchanges of 20 constant areas at most proportional to KN^2.

Apart from these frequent element-wise operations, two less frequent reductions in the T direction are required. These can be predicted to introduce an overhead which grows logarithmically with the number of communicating nodes, and thus, represent the asymptotic limit to scalability. To maximize potential problem scale, we will focus on identifying partition sizes to optimize computational throughput per node in a strong scaling scenario. Therefore, we treat the problem dimensions as independent variables, and investigate performance characteristics as functions of the parameter space, providing approximate ranges for how specific problem data maps onto differing hardware architectures.

2.2. *Initial Data Distribution*

Arbitrary size input data is emulated using pseudo-random numbers with a Gaussian distribution. This ensures that the numerical results will not destabilize the computation and affect performance by floating point exceptions or related artefacts. The random number generator was seeded using a constant for correctness testing purposes, as this produces a deterministic, repeatable state at any point throughout the computation.

Because we require consistent testing across different programming paradigms and platforms, random number generation is manually implemented, in the form of an Xorshift pseudo-random generator 10, coupled with the Box-Muller transform to produce normal distribution 10. This technique requires sequential execution from the same seed value in order to produce identical sequences of values, but as this is an artificial constraint for testing purposes, initialization cost is omitted from run time measurements.

The intention of admitting arbitrarily dimensioned input data is to carry out experiments which can treat the problem parameters as independent variables, and thus focus on the performance characteristics of the computing platform without regard to how well the computed results model any specific neural network of

interest. Our results show that the choice of programming tools that achieves highest performance is sensitive to variations in these parameters, but in a specific, applied scenario, the data partitioning may be subject to constraints determined by empirical data collection procedures, rather than by available computer architectures.

For the purpose of evaluating candidate node architectures for extreme scale computations, it is important to note that our benchmarking approach must be used in conjunction with information about the specific structure of interesting problem instances, as it indicates that neither of our tested programming models unilaterally obtains superior performance for every problem configuration on newer architectures.

3. Methodology

3.1. General Performance Considerations

The structure of the computation can be divided in two major phases: an inner loop and an outer loop.

The inner loop consists of element-wise combinations of local matrices, and is an iteration to convergence. As convergence criteria are ultimately data-dependent, we examine this in terms of the computation rate per iteration rather than the absolute time to solution, and note that configurations with a better rate will accumulate a performance advantage proportional to the convergence properties of the input data set.

The outer loop contains an update of the overall system according to the state after the inner loop, which requires neighbourhoods along the t axis to be taken into consideration. It also measures progress by collecting statistics on global state, requiring two global reductions. These steps are what introduce the application's communication requirements, and their frequency ultimately form the bottleneck to scalability.

It follows that the optimal configuration for a given data set is determined by the optimal rate of inner loop iterations weighted by the convergence rate, and the performance of global reductions. In order to assess node architectures with respect to scalability, we will compare their characteristics in these terms, with the expectation that the optimal choice of node architecture will depend on the dimensions and nature of particular experiments.

3.2. OpenCL Parallelization

The fitness of a given accelerator technology is predominantly tied to the efficiency with which it can execute the inner loop, as its computational load can be entirely hosted on the accelerator unit. The outer loop is less frequently executed, but it was found during implementation that its two required reductions easily become a serialization bottleneck that dominates the cost of transferring the required matrices to the host unit, using OpenMP reduce directives, and transferring the results back to the accelerator. This hybrid approach was used in our measurements, and is reported as a separate cost.

3.3. OpenMP Parallelization

A complete version of the program using OpenMP exclusively serves as a basis for comparison with the OpenCL version, in order to provide a realistic picture of the trade-off between using homogeneous compute nodes. Its data structure layout is identical to that of the OpenCL version, and parallelization is performed along the same dimensions. It was found during development that performance is sensitive to synchronization requirements between sockets in multi-socket systems, so each instance of the OpenMP program has been tested with the thread count of a single chip, and pinned to its cores using process affinity masks.

3.4. Target Platforms

We investigate the effectiveness of our candidate implementations on three candidate CPU/accelerator combinations, shown in Table 1.

Type	CPU	CPU cores / socket	Accelerator	Accelerator cores
A	Intel E5-4627	8	Nvidia K6000	2880
B	Intel i7-3770	8	AMD Radeon HD 7970	2048
C	Intel i7-4930K	6	AMD Radeon HD 4650	320

Table 1. Evaluated node architectures.

It should be noted that these systems span a range of technology choices for the purpose of testing our method of evaluation. Types A and B are clearly the more suitable candidates for an exascale system if only because of their more recent architectures, but we expect further architectural developments before a large-scale installation becomes feasible, and aim primarily to support rapid assessment of future architectures.

4. Results and Discussion

The test data sets in this section have been collected from runs of N=25 through N=200, in steps of 25, with every combination of M=3 through M=12 in steps of 3, as this was sufficient to obtain clear tendencies in the great majority of cases. Results in the figures are interleaved along the horizontal axis, with N as the major tendency, and M the minor. Each result is a cost per iteration, averaged from a collection of 10.

4.1. Inner Loop

The parallelization of the inner loop is critical to the scalability of the application, as it has a significantly higher trip count than the outer. For the OpenCL solution, the entire inner loop can utilize the data parallelism of the accelerator unit, as all operations are independent combinations of individual matrix elements. Comparisons for all node types are shown in Figure 3, Figure 4, and Figure 5.

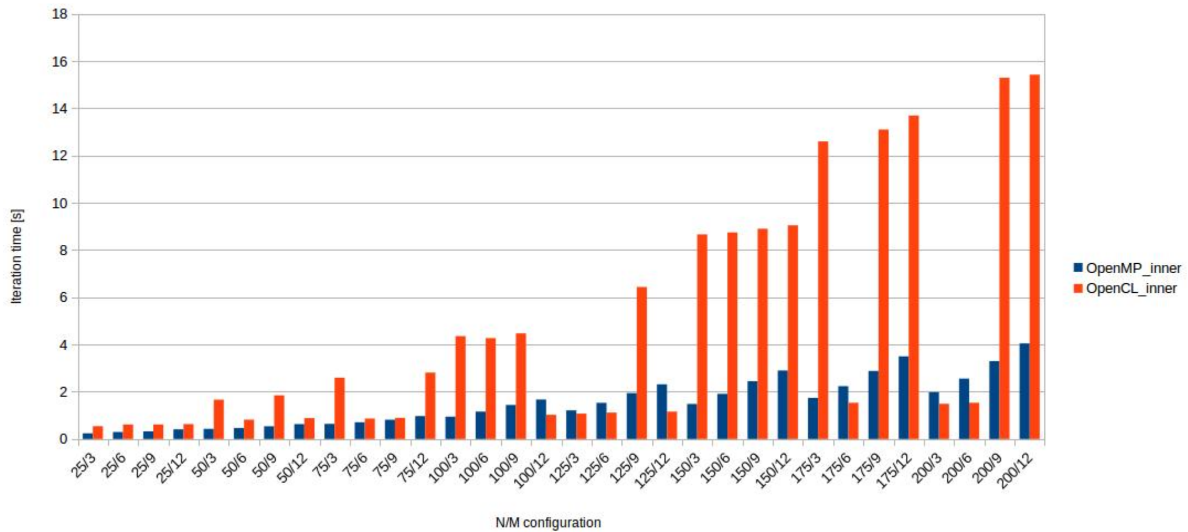

Figure 3. Inner loop iteration times for node type A.

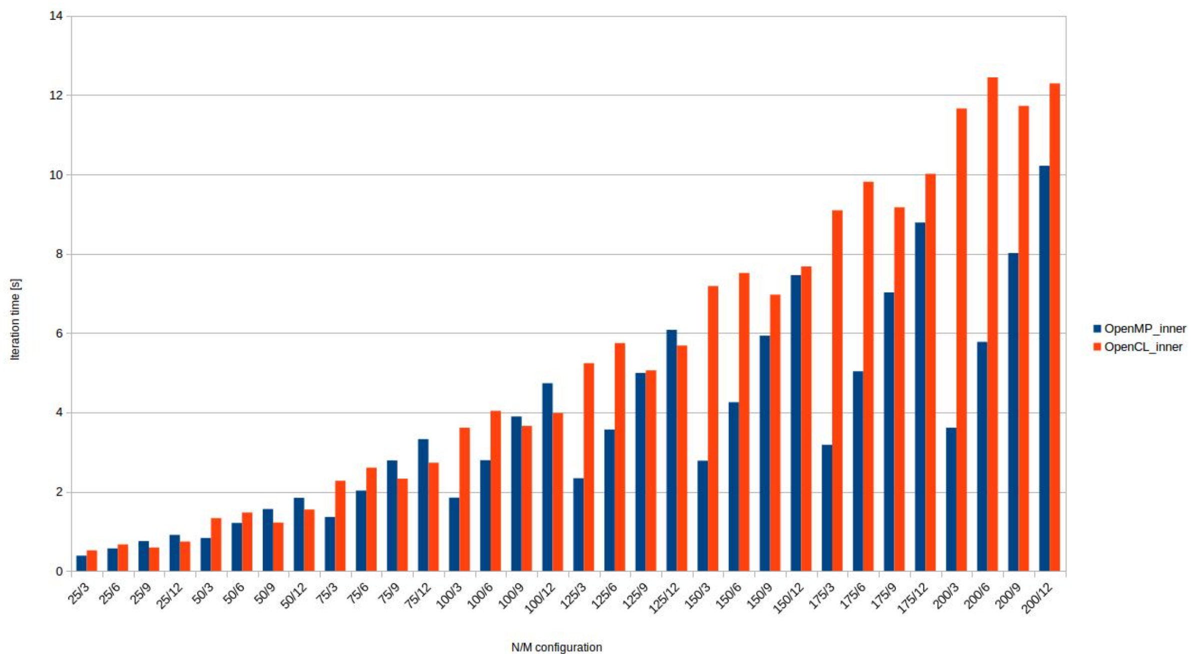

Figure 4. Inner loop iteration times for node type B.

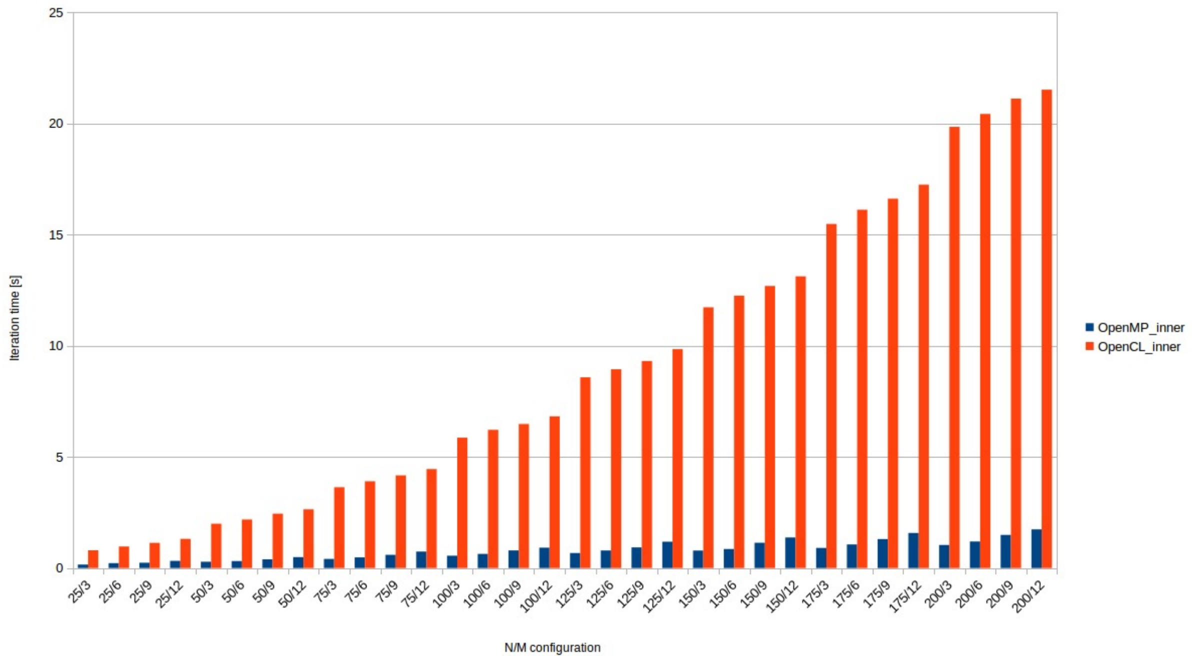

Figure 5. Inner loop iteration times for node type C.

The most striking feature of Figs. 3, 4, and 5, is that inner loop performance of the OpenCL and OpenMP versions is similar to within an order of magnitude, in spite of the fact that the computational work contains a far greater number of independent operations than the available processing core counts in either case, and the OpenCL implementation accesses between 53 and 360 times as many cores. We attribute this to the relatively low numerical intensity of the computational kernels: apart from the hyperbolic tangent operations, the majority amount to expressions of up to 7 unique values, with each appearing at most twice in an expression, for a ratio only slightly more than 1 floating point operation per memory fetch operation.
It is still interesting to note that as these are costs per iteration of a frequently executed loop, small advantages multiply with the trip count, so the fact that there are points where the OpenCL rate is slightly better than the OpenMP one indicates that there are problem configurations where it can result in substantially shorter time to solution. In order to verify that these slight advantages were not due to inaccurate measurements, test cases of the 200/3 and 100/12 configurations were re-run with inner loop trip counts of 300 per outer loop iteration, which resulted in a wall clock performance gain over the corresponding OpenMP configurations.
Finally, we can note that the type C node shows no case where OpenCL is the favorable version, and that its accelerator unit is of an older design. Noting that memory latency sensitivity has been a concern for general-purpose graphics processor use for a number of years, it can be expected that coming hardware generations may shift the balance demonstrated here, which favours OpenMP in the great majority of cases.

4.2. Outer Loop

The parallelization of the outer loop encompasses reduction operations, and is thus not trivially parallel in the same manner as the inner. The majority of these still prove to be efficiently executable on accelerator hardware; as they aggregate sums over 2-dimensional matrices, they can be structured in two phases, with the first aggregating sums in one direction into a temporary, linear array, and the second obtaining the overall sum. This proved inefficient in two cases, which are discussed in the next subsection. The per-iteration costs of the outer loop sections which were amenable to OpenCL parallelism are shown in Figure 6, Figure 7, and Figure 8.

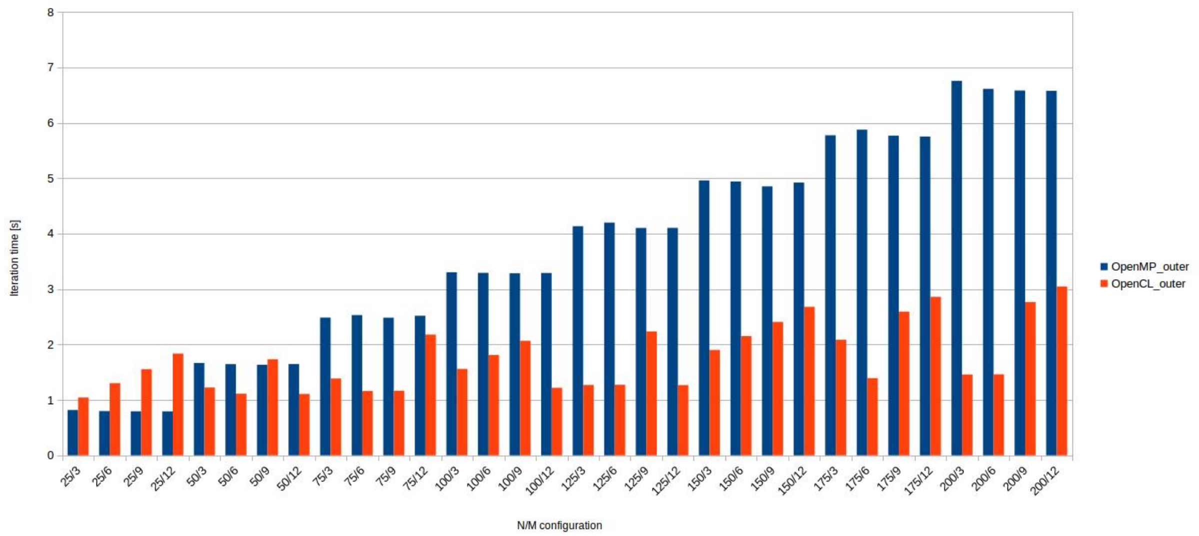

Figure 6. Outer loop iteration times for node type A.

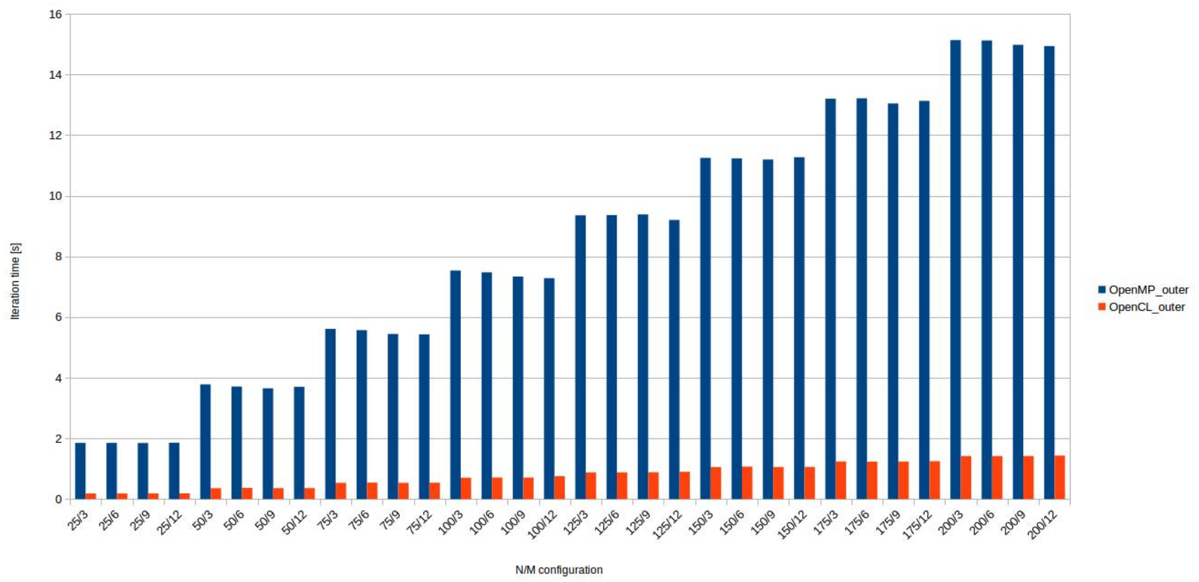

Figure 7. Outer loop iteration times for node type B.

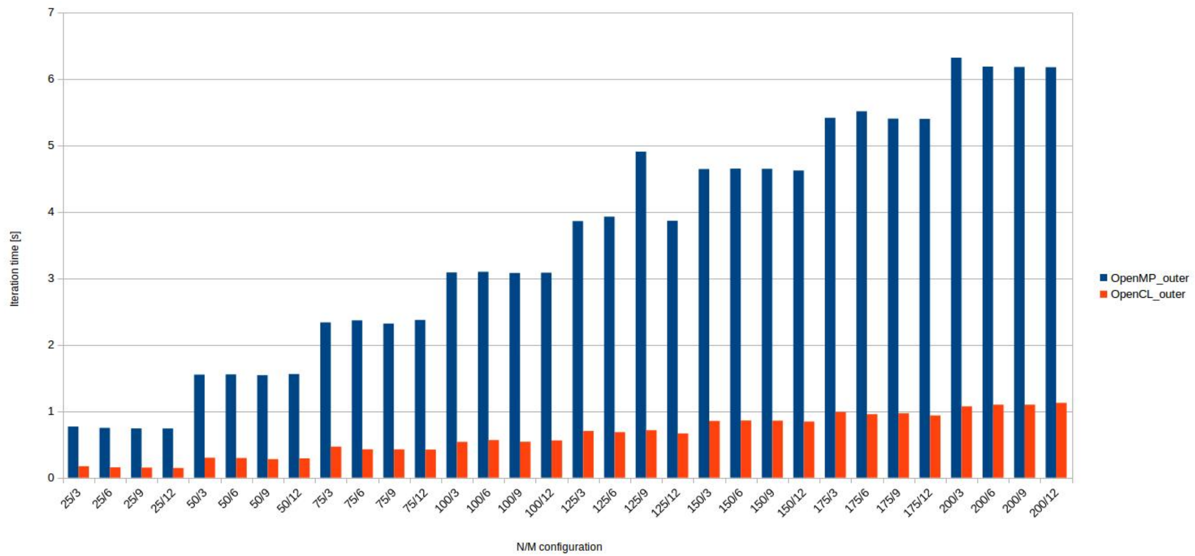

Figure 8. Outer loop iteration times for node type C.

4.3. Vector Reductions

Two reductions that are carried out in the outer loop were badly suited to the two-phase scheme, as they operate on vectors, causing a significant sequential bottleneck when run on accelerator units. The OpenCL program handles these by copying the data to the CPU side, and using the reduction routines of the OpenMP implementation. Figure 9, Figure 10, and Figure 11 report the costs of data transfer and reduction, in comparison with the OpenMP version.

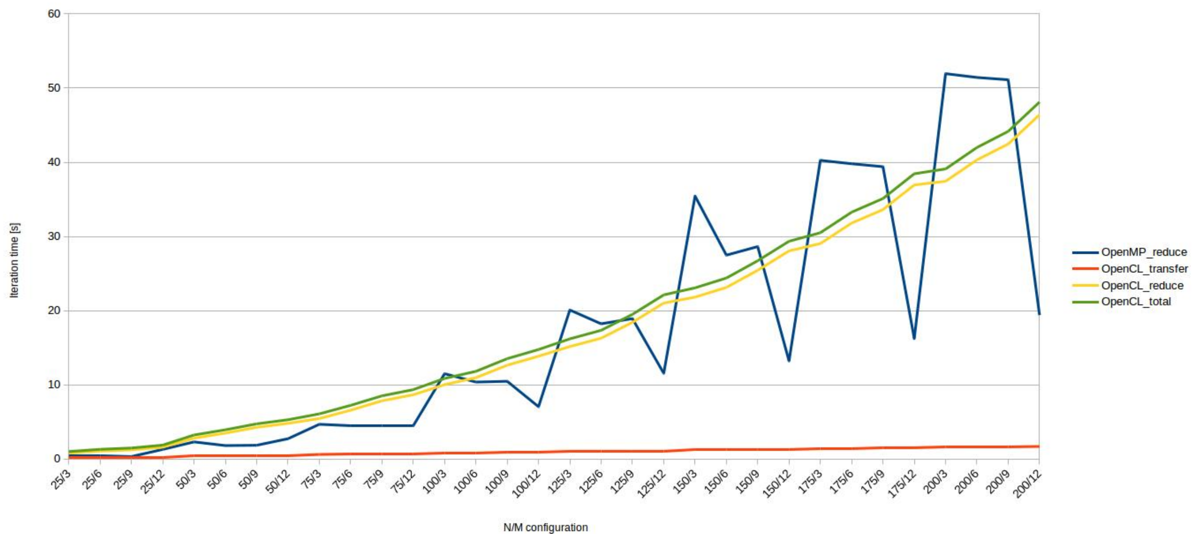

Figure 9. Vector reduction times for node type A.

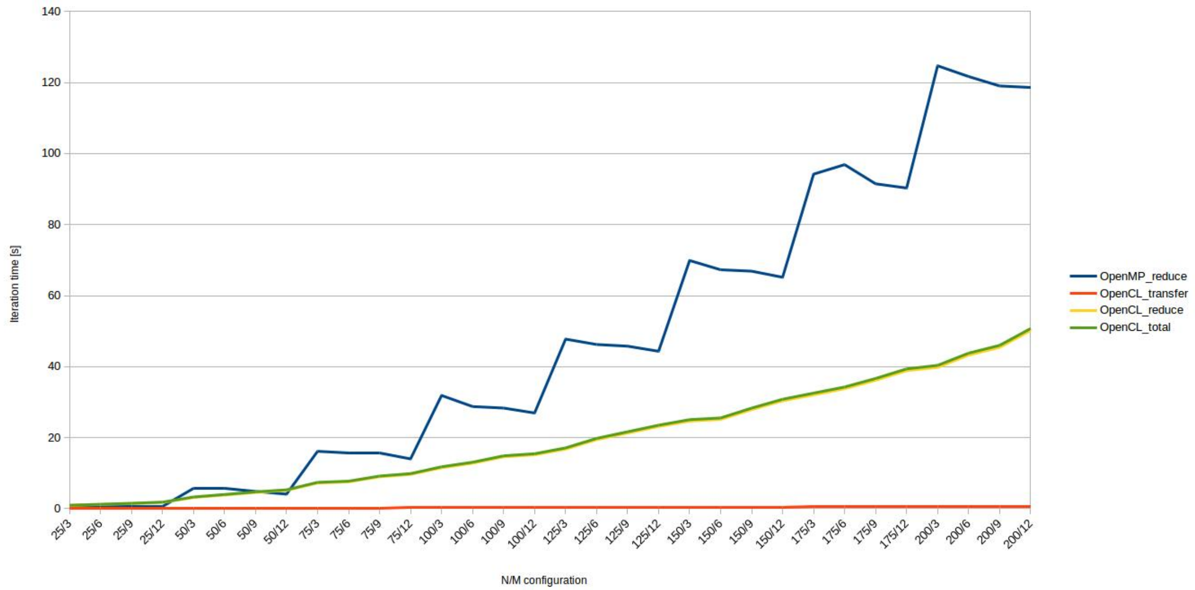

Figure 10. Vector reduction times for node type B.

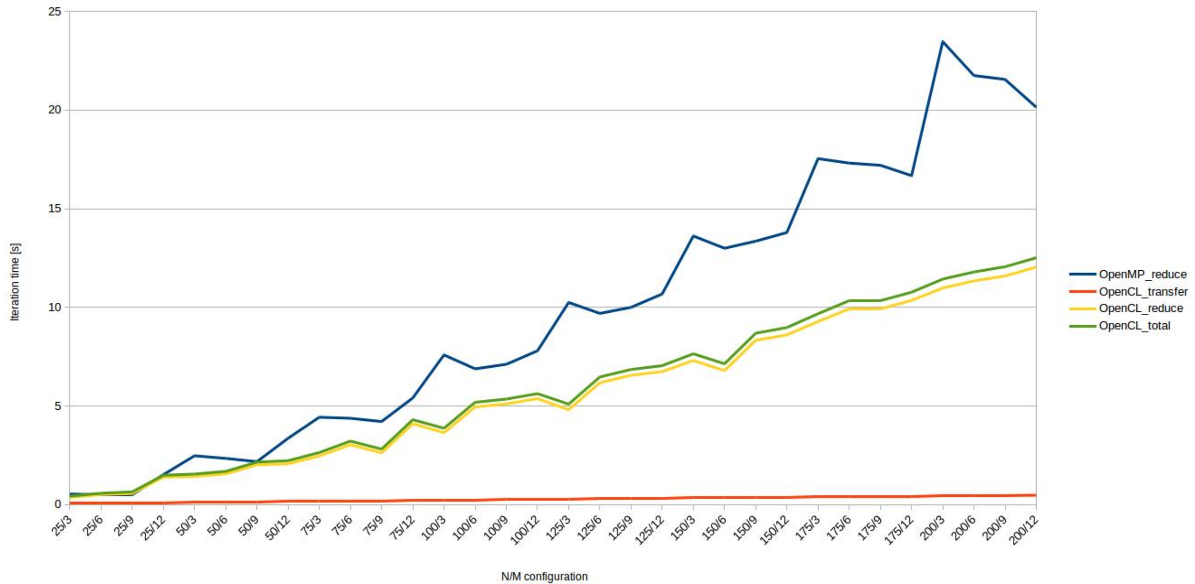

Figure 11. Vector reduction times for node type C.

The main point to note in Figures 9, 10, and 11 is that the transfer cost is insignificant compared to the cost of the reduction operations, and thus, that this method has little impact on the choice of fastest implementation.
It is interesting to observe that on both node types B and C, the total time to carry out the reductions in question is shorter than that of the full OpenMP implementation, despite the fact that the computational work is precisely identical. This can be attributed to a locality effect: the memory space required for the vectors is a fraction of the total memory footprint of the entire problem representation. When this is primarily hosted by the accelerator, the working set for the reduction operations becomes smaller and denser in the CPU address space.

The memory requirements of the application are dominated by 17 matrices sized proportionally to NT, representing the total network size and time variations, respectively. This creates a memory footprint on the order of gigabytes already at our most modest problem configuration of 25 nodes and 10^6 observations. Parallel speedup at the node level is sub-linear beyond the point where the application begins to exhibit memory bound behaviour, but using the OpenMP implementation, we observe a consistent speedup factor 5 for 8 threads on node type A, for a parallel efficiency of 62.5%, regardless of problem size. Distributed memory parallelization would circumvent the memory bottleneck by increasing the aggregate memory bandwidth in proportion to the

number of participating nodes, and as increased network sizes can be compensated by reducing the size of each node's section of the T domain, we expect the limiting factor on the scale of admissible problems to be the cost and frequency of global reductions, as dictated by the input data.

5. Conclusions

We have studied performance portability at the heterogeneous node level, by examining a range of heterogeneous node architectures, and evaluating their suitability as components in large-scale systems for inferring hidden network structures from time-varying samples. The portability of OpenCL programs allows us to examine performance across a range of accelerator hardware without requiring porting work to vendor-specific programming models. The resulting code is also performance-portable, in the sense that it exposes the same architectural limitations on each platform. Specifically, the moderate numerical intensity of the performance-critical inner loop of the application stresses memory latency masking, indicating that architectural developments to address this will produce more suitable accelerator units.

Regrettably, the test platforms respond non-uniformly to changes in the dimensions of the input, making the ideal choice of architecture a function of the specific problem instance. This implies that in order to identify the most suitable architecture for large scale problems, it is recommended to carry out a preliminary exploration of the parameter space similar to the one presented in this study, determine the most favorable sub-problem size per node, and select architectures accordingly.

An interesting direction for future work will be to investigate the performance of the OpenMP port used in this study on the Intel Knight's Landing architecture, as its design specifically addresses the combined requirements of a large number of cores and significant amounts of high-speed memory.

References

[1] Computational Throughput of Accelerator Units with Application to Neural Networks, http://www.prace-ri.eu/IMG/pdf/ wp164.pdf, retrieved 12.12.2016.

[2] OpenCL – The open standard for parallel programming of heterogeneous systems, https://www.khronos.org/opencl/, retrieved 12.12.2016.

[3] Xorshift RNGs, G.Marsaglia, Journal of Statistical Software, Vol. 8, No. 14.

[4] A Note on the Generation of Random Normal Deviates, G.E.P. Box, M.E. Muller, Annals of Mathematical Statistics, Vol. 29, No. 2.

Acknowledgements

The authors would like to thank prof. Anne C. Elster and HPC-lab at the Dept. of Computer Science, NTNU, for access to their AMD GPU resources.

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838.