Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Computational Throughput of Accelerator Units with Application to Neural Networks

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

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

Abstract

The size of data that can be fitted with a statistical model becomes restrictive when accounting for hidden dynamical effects, but approximations can be computed using loosely coupled computations mainly limited by computational throughput. This whitepaper describes scalability results attained by implementing one approximate approach using accelerator technology identified in the PRACE deliverable D7.2.1 [1], with the aim of adapting the technique to future exascale platforms.

Introduction

A challenging problem in neural computational research is that of estimating structures of neural networks which are hidden from the limited experimental data that can be acquired from, for example, neural recordings. This whitepaper investigates the potential for improving the parallelisation of a sample application that implements an approximate expectation-maximization method [2] to inferring the network structure and time varying states of a hidden population within the framework of the kinetic Ising model. The kinetic Ising model is a tool developed in statistical physics which has been recently brought forward as an efficient method for the analysis of non-equilibrium systems [3]. The size of networks that can yield informative results can be made arbitrarily large, and the long-running computational demand is highly localised, making the application a strong candidate for exascale computations. We describe a proof-of-concept implementation that adapts a sample application code to use OpenMP on the Intel Xeon Phi accelerator architecture. Accelerator-enabled platforms are currently at the forefront of massively parallel computations at the petascale, and can be expected to remain key technologies as supercomputers address exascale challenges: both were identified as candidate exascale technologies in the PRACE deliverable D7.2.1[1].

Sample Case Characteristics

This section describes a case study of an example application written in Python, and the adaptations made to enable it to use accelerator units. It begins with a description of the test platform, before it describes results from a preliminary profiling of its behaviour, and finally describes how it was translated to an implementation that shows favourable strong scaling characteristics both on a Xeon Phi accelerator unit and a general processor.

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

Test Platform

Development and experiments were conducted on a 6-core i7 workstation featuring a Xeon Phi card, with the technical specifications shown in Table 1.

Table 1: Test system configuration

Host CPU	6-core i7-4930K, 3.4 GHz clock speed
RAM	2x8GB, 1.6GHz bus speed
Motherboard	ASUS P9X79 WS, Socket-2011, PCIexpress 3.0 x16
Accelerator unit	57-core Intel Xeon Phi 3120A, 6GB onboard memory, 28.5MB cache

A Python 2.7.3 interpreter was configured with NumPy 1.7.0 and SciPy 0.12.0c1, compiled using the Intel compiler toolchain 14.0.1 using its MKL libraries[5] to satisfy BLAS and LAPACK dependencies. The same compiler version was used to compile all C sources, targeting both the host CPU and Xeon Phi architectures.

Structure of the Example Application

As a starting point for this study, a small, representative application program was written in Python, and a sample data set was generated for validation purposes. This implementation utilises the *NumPy* and *SciPy* packages for data format handling, programmability, and low-level linear algebra routines.

The high-level structure of the computation is a sequence of iterations which produces a series of single scalars reduced from a large working matrix, which was sized to a PxT matrix, with P=22 and T=199986 derived from a 33MB data set in the sample case. It is summarized in Figure 1.

For each iteration:

1. Initialize a matrix μ from a window of the input data
2. Iteratively refine μ matrix to convergence
3. Reduce single scalar from converged μ matrix

Figure 1: High-level structure of the computation

The reduction step contains negligible data movement and cost compared to the convergence of the inner iteration to convergence, which can be performed for several iterations after a single transfer of the input data, to place it in an accelerator unit's memory. This suggested the inner loop as a viable target for optimization.

The iterative refinement process consists of a sequence of element-wise operations and dot products applying a small set of PxP and PxM masks also derived from the input data, with M=3 in the sample data, as well as some intermediate results derived from μ.

Preliminary Profiling

The initial Python version suggested that integrating parallelized BLAS and LAPACK routines may be a viable method of parallelization, and the customized Python interpreter was used to investigate this. An initial profile of the sample application was obtained using the *callgrind* module of the *valgrind* run-time profiling framework [6], running on the host CPU, and a visualization of the distribution of run time for 10 converged iterations is show in Figure 2, which was produced using the tool *kcachegrind* [7].

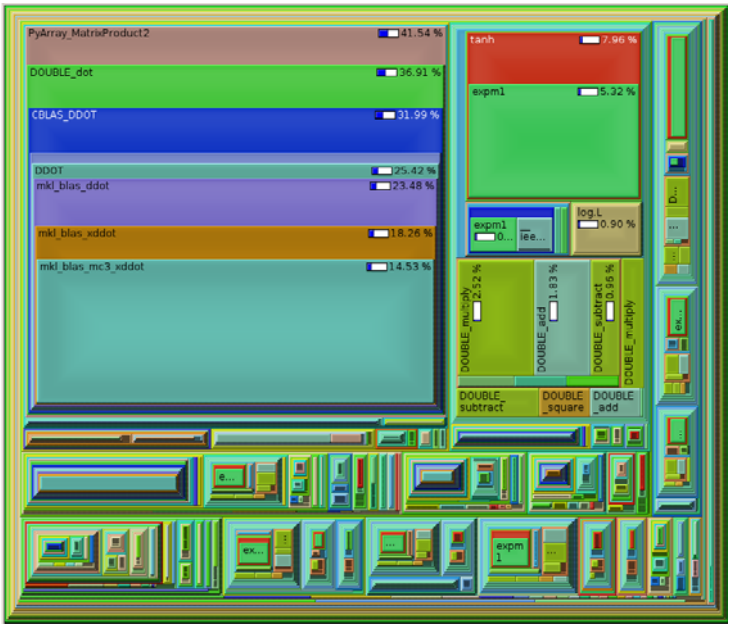

Figure 2: Visualization of run time for 10 iterations

As can be seen in Figure 2, this initial profile indicates that the fraction of time spent in MKL subroutines amounts to some 36.91%, all due to use of the *ddot* function. With this as the only source of parallelism in the program, adjusting the execution environment to admit higher thread counts gave no measurable benefit, and only a single core was utilised. This can partly be explained by the limited size of the sample data set, but regardless of this consideration, Amdahl's Law suggests that unless larger problem sizes result in dot products constituting a significantly larger fraction of the total work, parallel speedup will be bounded by a number of cores far smaller than the available number on future exascale architectures. As production use with larger data sizes has been reported to achieve peak speedup between 2 and 4 cores, an alternative to the library-based parallelisation was deemed necessary.

Adaptation to OpenMP

Examination of the source program's inner iteration to convergence showed that it consists entirely of operations which can be written as doubly or triply nested loops that are amenable to parallelization. Figures 3 and 4 show an example of an element-wise operation in Python, and a corresponding implementation in C.

```
DTB = 1.0 – tanhB ** 2
```

Figure 3: Combination of element-wise operations in a Python expression

```
#pragma omp parallel for
for ( size_t t=0; t<TS; t++ )
    for ( int8_t y=0; y<M; y++ )
        DTB(y,t) = 1.0 – SQUARE(TANHB(y,t));
```

Figure 4: C implementation corresponding to the expression in Figure 3

The indexing of matrices DTB, tanhB and the SQUARE operation were all encapsulated in parametric macros, in order to admit experiments with different memory layouts and implementations of exponentiation. Rewriting each operation in the convergence iteration of the original program produced a sequence of which could be subjected to manual implementation of loop fusion[8] where matrix sizes were identical. Manual strength reduction [8] was also found to be effective, in the form of implementing the SQUARE operation as a multiplication operator, rather than a call to the standard library exponentiation routine. After fusion and optimizations, the implementation consisted of four triply nested loops, five doubly nested loops, and two linear array assignments, all of which were subject to parallelization with OpenMP[9]. A row-major layout of the matrices combined with parallelization of the outer loop was found to give the greatest benefit in terms of execution time, and the compiler identified vectorizations of the inner loops. Apart from straightforward

floating-point arithmetic, two of the inner loops involve hyperbolic tangents, which was implemented as a standard library call because it is non-trivial to implement explicitly.

All intermediate numerical results of this inner loop were validated to be identical with the values produced by the Python version, and convergence was established with the same values, in an identical number of iterations. To account for the altered overheads of switching implementation languages, execution was timed to convergence of the inner loop for the first iteration of the sample problem, which gave a factor 10.6 speedup without parallel execution. The performance of this C translation will henceforth be used as a baseline for further comparisons.

Performance Results

Tests were conducted on both the 6-core CPU of the host system, and native mode execution on its 57core Xeon Phi coprocessor. Figure 5 shows parallel speedups in strong scaling mode, using the sample problem.

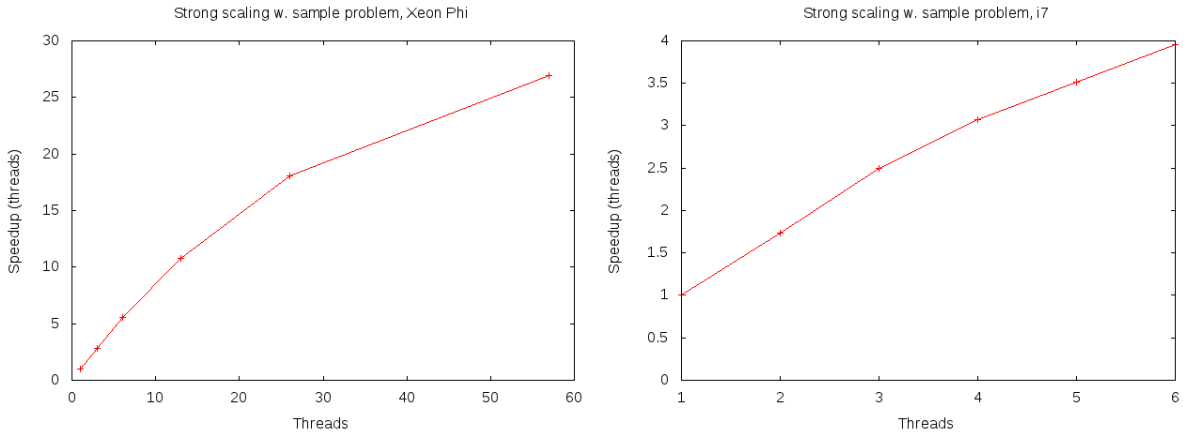

Figure 5: Parallel speedup of sample problem on Xeon Phi and i7

As per the PRACE Xeon Phi best practice guide [4], Xeon Phi cores support execution of multiple threads, and can be expected to yield advantages by exploiting this capability. Figure 6 shows the absolute timings of sample runs utilising the full number of cores, and overcommitting threads by factors 2, 3 and 4.

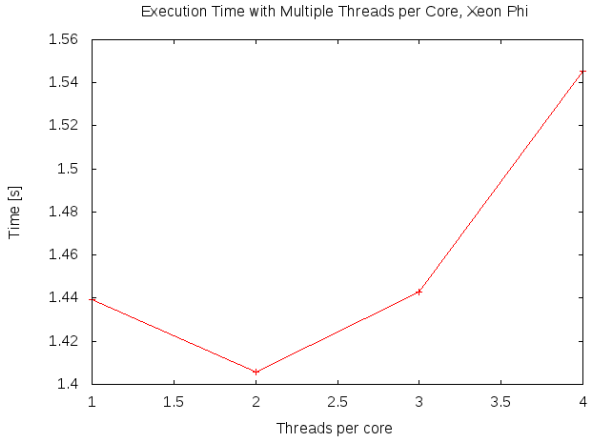

Figure 6: Absolute times to convergence of first iteration, multiple threads per core

Finally, as the speedup plots give no indication of absolute execution time between the two architectures, Figure 7 shows them in comparison.

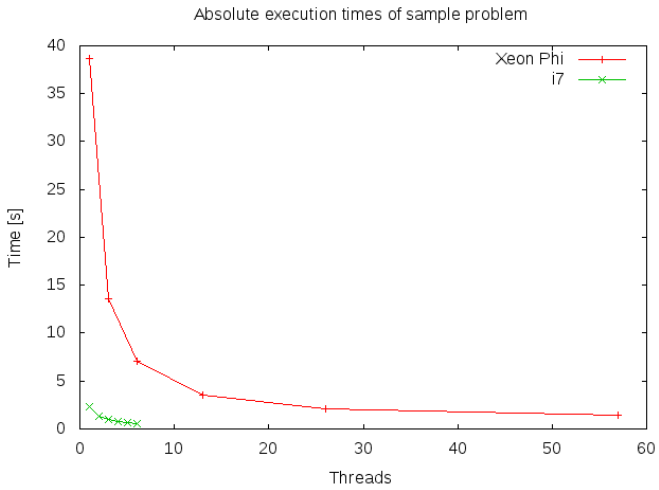

Figure 7: Absolute times to convergence of first iteration

Discussion

As the figures show, the performance bottleneck of the computation is parallelizable in its entirety, and in spite of reaching diminishing returns with a comparatively small sample problem, still obtains speedups through employing the full hardware resources of both a conventional multi-core processor, and an Intel Xeon Phi accelerator board. The memory available to both architectures suggests that problem sizes can easily be scaled to many times that of the sample, and utilise all the available hardware, making it a strong candidate for future exascale platforms.

While the proof-of-concept implementation studied in this whitepaper falls short of a full implementation, the loosely coupled nature of the computation suggests that the small communication requirements between iterations to convergence can admit a throughput-oriented implementation with a coordinating process managing working processes on both of the tested architectures. Although a small amount of coordinating code will be needed for convergence testing in the bottleneck iteration when using multiple accelerators, such testing is not necessary on each iteration, and can be tuned to communication latencies. The remaining operations are all element-wise or associative, suggesting that the input data can be split across multiple, independent units quite flexibly. The advantage of superior absolute execution time on the multi-core processor suggests that utilising a combination of units may cause a load balancing issue that can be ameliorated with appropriate partitioning of a larger problem, but quantifying the balance will depend strongly on the details of the target machine, placing it beyond the scope of this whitepaper.

It can be remarked that a similar run-time profile of the implementation as that shown in the preliminary profiling indicate that a 23% portion of the C version's run time is spent on computing hyperbolic tangent values. While the Intel Xeon Phi implements this as a function call, this is an indication that further speed improvements may be obtainable by porting the application to graphics accelerators, where support for trigonometric functions may be implemented in hardware on appropriate models.

Conclusions

We have described a proof-of-concept parallelization of the computationally intensive step in a neural network inference application, and demonstrated that it shows favourable scalability up to full utilisation of host and accelerator processors in a system resembling a node from current petascale architectures. The limited communication requirements of the application suggest that its performance is primarily bounded by computational throughput, which makes it a feasible candidate to utilise exascale resources.

References

[1] M. Lysaght, B. Lindi, V. Vondrak, J. Donners, M. Tajchman, PRACE-3IP D7.2.1 *A Report on the Survey of HPC Tools and Techniques*, http://www.prace-project.eu/IMG/pdf/d7.2.1.pdf

[2] Dunn, Benjamin and Roudi, Yasser, Learning and inference in a nonequilibrium Ising model with hidden nodes, Physical Review E, 2013

[3] Roudi, Yasser and Hertz, John, Mean field theory for nonequilibrium network reconstruction, Physical review letters, 2011
[4] M. Barth, M. Byckling, N. Ilieva, S. Saarinen, M. Schliephake, V. Weinberg, *Best Practice Guide: Intel Xeon Phi,* http://www.prace-ri.eu/Best-Practice-Guide-Intel-Xeon-Phi-HTML
[5] Intel Math Kernel Library, http://software.intel.com/en-us/intel-mkl
[6] Valgrind, http://valgrind.org
[7] Kcachegrind, http://kcachegrind.sourceforge.net/
[8] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman: *Compilers: Principles, Techniques and Tools*, 2nd edition, Addison Wesley, 2007.
[9] OpenMP, http://www.openmp.org

Acknowledgements

This work was financially supported by the PRACE-3IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-312763.