D3.5.2 –Compiler support for exascale

WP3: Development Environment

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exascale computing, software and simulation

Due date:	M24
Submission date:	30/09/2013
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organisation	UEDIN
Version:	1.0
Status	Final version
Author(s):	David Henty, Luis Cebamanos (UEDIN) Jing Gong, Stefano Markidis (KTH) Alistair Hart (CRAY)
Reviewer(s)	Alan Gray (UEDIN), Christoph Niethammer (USTUTT)

Dissemination level	
<PU/PP/RE/CO>	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	31/08/2013	Partial draft of the deliverable containing CPU results	David Henty, Luis Cebamanos (UEDIN); Jing Gong, Stefano Markidis (KTH); Alistair Hart (CRAY)
0.2	02/09/2013	Now contains all GPU results as well.	David Henty, Luis Cebamanos (UEDIN); Jing Gong, Stefano Markidis (KTH); Alistair Hart (CRAY)
0.3	03/09/2013	First complete draft for internal review	David Henty, Luis Cebamanos (UEDIN); Jing Gong, Stefano Markidis (KTH); Alistair Hart (CRAY)
1.0	24/09/2013	Final version after internal review; includes updated GPU results.	David Henty, Luis Cebamanos (UEDIN); Jing Gong, Stefano Markidis (KTH); Alistair Hart (CRAY)

Table of Contents

Index of Figures

1 Executive summary

A study of the performance of the computational kernels relevant to the Nek5000 CRESTA co-design application was completed last year [1]. This included a brief study of CPU performance and a more in-depth study of performance on a single GPU. The GPU study used the PGI compiler suite and OpenACC accelerator directives, coupled with auto-tuning compiler technology from the University of Edinburgh School of Informatics. A standalone benchmark version of the full Nek5000 application, called Nekbone, was subsequently ported to large-scale Cray GPU parallel systems using the Cray OpenACC compiler and then optimised by hand [2]. The design of a CRESTA auto-tuning framework was also developed, and a prototype implementation produced [3]. In this study we draw these three strands together and use the CRESTA auto-tuner on the Nekbone kernels to attempt to produce an optimised accelerated version for Cray hardware whose performance can be compared with the hand-optimised accelerator code. We also perform an in-depth investigation of a similar approach applied to the CPU version of Nekbone to enable comparisons of the auto-tuning procedures and performances achieved between CPU and GPU. Although auto-tuning is found to be extremely beneficial for all the kernels and for the full Nekbone code on the CPU, there is still substantial room for improvement in the performance of the auto-tuned accelerator code. It appears that the kernel routines behave very differently when run within the Nekbone application compared to being run in isolation. Initial results indicate that good performance could be obtained if the kernels were auto-tuned from within Nekbone, but this work is still in its early stages. Although hand-tuned versions of Nekbone currently perform much better than using default OpenACC settings , their performance still does not substantially exceed that of the best CPU code. Our future auto-tuning work will be aimed at further increasing the GPU performance.

2 Introduction

This report is an update to *D3.5.1 Compiler Support for Exascale*, extending the work done there and also widening its scope to include results from other CRESTA workpackages. It should be read in conjunction with the previous work in D3.5.1; we do not reproduce any background material here.

2.1 Structure of the report

In section 3 we present results for the CPU performance of the Nek5000 [4] DGEMM kernels on Cray CPU hardware, and discuss the issue of what precise versions of the kernels are most representative of the Nek5000 application. In section 4 we incorporate these optimised kernels into the CPU version of Nekbone (a stripped-down version of Nek5000 designed specifically for benchmarking studies) and present performance results. We then repeat the same procedure for the OpenACC accelerated version of the code: we investigate optimisation of the kernels for the GPU using the CRESTA auto-tuning framework in Section 5; we incorporate these kernels into an OpenACC version of Nekbone in Section 6. We then compare these CPU and GPU results with those obtained from a hand-optimised OpenACC version of Nekbone in Section 6.2, and provide conclusions and suggestions for further work in Section 7.

2.2 Hardware and software

As the ultimate aim of this workpackage is to investigate performance on very large scale systems, we exclusively target Cray hardware and associated compilers in this study. All CPU benchmarking was performed on the UK's national supercomputer HECToR, a Cray XE6 system operated by EPCC. All GPU benchmarking was performed on Raven, a Cray-internal XK6 development system. A single node of HECToR has two sockets, each containing a 16-core AMD Interlagos CPU. Raven has almost identical hardware, except that one of the AMD processors on each node is replaced by an Nvidia Kepler GPU. This enables very clean performance comparisons between CPU and GPU: the correct comparison is one node of HECToR (2x16-core CPUs) against one node of Raven (1x16-core CPU + 1xGPU).

2.3 Purpose

The purposes of this deliverable are:

- to investigate the auto-tuning of a GPU-accelerated version of the Nekbone benchmark code using the CRESTA auto-tuning framework;
- to compare these results with the best performance obtained from hand-optimised GPU, and auto-tuned CPU, versions of the same code.

Nekbone contains all the core functionality of the CRESTA co-design application Nek5000, so this deliverable is relevant across the whole CRESTA project.

2.4 Glossary of Acronyms

BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
D	Deliverable
EC	European Commission
FLOP	Floating-point Operations per Second
GPU	Graphics Processing Unit
HECToR	High End Computing Terascale Resource
HPC	High Performance Computing
MFLOPs	Million Floating-point Operations per Second
OpenACC	Open Accelerator Directives
PGI	The Portland Group, Inc.
RMS	Root Mean Square
WP	Work Package

3 CPU performance

When performing any performance investigation of new hardware such as GPUs, it is essential to have reliable baseline results from existing hardware (here the CPU). For a fair comparison, this should be the best possible performance that could be obtained.

In [1] we developed a small stand-alone kernel benchmark program by extracting existing benchmark subroutines from Nek5000. We observed that in some cases one or more of the 8 different hand-coded kernels out-performed the optimised DGEMM routine from the BLAS library (e.g. for small matrices, since BLAS is usually targeted at large matrices). In order to obtain reliable baseline performance for Nekbone, it was therefore necessary to run a version which used the optimal version of these kernel routines in all cases. Since we have already seen that no single kernel is optimal for all parameter values, this required additional work to be done on the CPU code.

3.1 Existing benchmark

The existing DGEMM benchmark times the following operation:

- Declare three arrays *A*, *B* and *C*, each containing *M* matrices
- Repeat many times for timing purposes
    - Loop over $i = 1, 2, \ldots, M$
        - Compute matrix multiplication $C(i) = A(i) * B(i)$
    - End loop
- End repetition

The sizes of the individual matrices are determined by the value of *N*, which corresponds to the order of the spectral elements in Nek5000. The benchmark considers three cases for a given value of *N* each of which uses matrices of different dimension.

The three test cases of $C = A * B$ are:

1. $(N^2 \times N)$ matrix = $(N^2 \times N)$ matrix * $(N \times N)$ matrix (“tall times square”)
2. $(N \times N)$ matrix = $(N \times N)$ matrix * $(N \times N)$ matrix (“square times square”)
3. $(N \times N^2)$ matrix = $(N \times N)$ matrix * $(N \times N^2)$ matrix (“square times long”)

The value of M is related to the size of the problem that is being solved in Nek5000. In the kernel benchmark, a fixed amount of memory comprising *NFLOAT* floating-point numbers is declared (as distributed, *NFLOAT* = 4*8*8*8*130 = 266240). The value of *M* is then set to fill this array with as many matrices as possible, i.e. for each case:

1. $M = NFLOAT / (N^3)$
2. $M = NFLOAT / (N^2)$
3. $M = NFLOAT / (N^3)$

This means that the inner loop always has roughly the same number of floating-point operations of order $M \times N^3$. Although there is a factor of *N* more matrices for case 2, the matrices themselves are smaller so each matrix multiplication has corresponding a factor of *N* fewer floating-point operations.

As originally written, the Nek5000 benchmark measures the performance of each of 9 different kernels (8 hand-coded + DGEMM library) and for every value of *N* reports which is the fastest routine separately for each of the three cases. It also reports the routine which performs best on average by looking for the maximum harmonic mean performance (i.e. minimum runtime) over the three cases.

3.2 Relationship with Nekbone kernels

Because we expected to have to write new code in order to investigate the performance of Nekbone on GPUs, we wanted to fully understand the Nekbone computational kernels and how they related to the kernel benchmarks.

As mentioned above, the value of *N* equates to the order of the spectral elements in Nekbone which would be fixed by the user. The number of elements, *nel*, corresponds to setting a particular value of *NFLOAT* in the kernel benchmarks. As each element requires N^3 storage, *nel* = *NLFOAT* / N^3. In Nekbone, updating a single element requires six kernel calls: two separate calls for each of the three cases. The cases actually correspond to operations across different spatial dimensions of the 3D elements. For cases 1 and 3, the update can be cast a single matrix operation; for case 2 it requires *N* separate calls operating on smaller matrices. This is why the benchmark increases the value of *M* by a factor of *N* for case 2.

To summarise, the computational load of the Nekbone kernel is distributed equally between the three kernel cases. In terms of the number of elements *nel*, the value of *M* in the benchmark is set as *M* = *nel* for cases 1 and 3, and *M* = *nel* * *N* for case 2.

The core operation in Nekbone is implemented by a routine called **`ax_e`** which has the following form:

- Loop over elements *e* = 1, 2, …, *nel*
    - Loop over repeat = 1, 2
        - *Update e using kernel case 3*
        - *Loop over i = 1, 2, …, N*
            - *Update e using kernel case 2*
        - *End loop*
        - *Update e using kernel case 1*
    - End loop over repeat
- End loop over elements

The fact that the inner operations are repeated twice is not important in terms of the performance, and neither is the order of the calls to the cases, so we will ignore these details from now on and consider the following as the canonical form for **`ax_e`**

- Loop over elements *e* = 1, 2, …, *nel*
    - Update *e* using kernel case 1
    - Update *e* using kernel case 2 (*N* times)
    - Update *e* using kernel case 3
- End loop over elements

We will refer to this as a ***scalar*** version because the kernels operate on a single element at a time (although note that these are actually matrix operations).

Written in this way it is clear how it relates to the kernel benchmark as we are calling the kernels repeatedly across whole arrays of *nel* matrices, with a factor of *N* more calls for case 2.

Given that *nel* and *N* are likely to be fixed for many runs of Nek5000 (as they correspond to the basic discretisation parameters of the simulation) the way for a user to optimise Nekbone performance using the benchmark is as follows:

1. Set *NFLOAT* based on *nel*; the precise choice is not important with respect to performance, e.g. assuming that *nel* is large then it only has to be large enough to ensure that data is read from memory and is not cache resident.
2. Run the benchmark and find the routine with the best harmonic mean performance across all three cases.
3. Compile that single version and use in all calls.

This is a perfectly sensible procedure for the CPU version.

3.3 Implications for OpenACC kernels

To exploit the massive parallelism of GPUs requires many independent floating-point operations, and it was clear in advance that there would not be sufficient parallelism within a single kernel. We therefore decided that the accelerated kernels would have to operate on a whole array of elements at once. We will refer to this as a ***vector*** version because it operates on all the elements at once.

We therefore expected the accelerated Nekbone kernel **vax_e** to have the following form:

- Update all elements *e* using accelerated kernel case 1 (vector length = *nel*)
- Update all elements *e* using accelerated kernel case 2 (vector length = *nel* * *N*)
- Update all elements *e* using accelerated kernel case 3 (vector length = *nel*)

Writing the kernels in this form revealed that the actual operations of Nekbone were not exactly as measured in the existing kernel benchmarks. Rather than being of the form *C*(*i*) = *A*(*i*) * *B*(*i*), one of the matrices (either *A* or *B)* was actually fixed throughout the loop. Although this would initially seem to increase the complexity of benchmarking (two options, fixed *A* or fixed *B*, for each case) this was not true as the actual form was:

- Call accelerated kernel case 1 for *C*(*i*) = *A*(*i*) * *B* (vector length = *nel*)
- Call accelerated kernel case 2 for *C*(*i*) = *A*(*i*) * *B* (vector length = *nel* * *N*)
- Call accelerated kernel case 3 for *C*(*i*) = *A* * *B(i)* (vector length = *nel*)

We call these versions of the kernel *vxm* (cases 1 and 2) and *mxv* (case 3) to indicate that one argument (*A* first or *B* second*)* is a fixed **m**atrix, multiplied by the other which is a **v**ector of matrices. In this terminology, the original benchmark is *vxv* and the core matrix-matrix routine (e.g. DGEMM) is *mxm*.

Based on this information and experiences from the previous deliverable, we therefore anticipated three differences between the benchmarking required for auto-tuning the GPU version compared to the CPU version:

1. the value of *nel* is much more critical as it is an integral part of the vector kernel, so we might want to tune separately for each value of *nel*;
2. the GPU performance varies much more significantly with the case so we might want to call different kernels for each case and not just pick the best average;
3. we might want to optimise OpenACC settings separately for *C*(*i*) = *A*(*i*) * *B* and *C*(*i*) = *A* * *B(i)* and not just take the values from benchmarking *C*(*i*) = *A*(*i*) * B(*i*).

Finally, Paul Fischer (the author of Nek5000 and Nekbone) had also anticipated the need for parallelism over elements to be explicit in the OpenACC version but had taken a different approach. In the final version of Nekbone he supplied us, he included a new subroutine that was called **ax3d** and was of the form:

- Loop over elements *e* = 1, 2, …, *nel*
    - Update *e* completely using explicit code
- End loop over elements

i.e. it does not call any kernel routines at all. In this version, all the computations associated with the six kernel calls are written out in full with explicit loops over all indices, e.g. one section reads:

```
      do e=1,nel
         ...
         do k=1,n
         do j=1,n
         do i=1,n
            w(i,j,k,e) = 0
            do l=1,n
               w(i,j,k,e) = w(i,j,k,e) + D(l,i)*ur(l,j,k,e)
     $                                 + D(l,j)*us(i,l,k,e)
     $                                 + D(l,k)*ut(i,j,l,e)
            enddo
         enddo
         enddo
         enddo
      enddo
```

Note that, as expected, the computational complexity of each elemental calculation is $O(N^4)$; although matrix multiplication is $O(N^3)$ remember that for cases 1 and 3 one of the matrix dimensions is of size N^2, and for case 2 there is an additional loop over N.

This is an extreme form of a performance optimisation called *loop jamming*, so we refer to this as the ***jammed*** version of the kernel.

3.4 CPU kernel auto-tuning

As well as being an interesting exercise in itself, the CPU auto-tuning is a useful testing ground for optimising Nekbone before introducing the additional complexity of GPU acceleration. Note that we did not use any auto-tuning frameworks for this; we simply ran the existing benchmark many times as previously done for the CPU benchmarking in [1]. For simplicity we still call this “auto-tuning” because it is done in a mechanical way that could easily be automated in principle. Auto-tuning is relatively simple here as there are no compiler options to vary; we simply set the highest optimisation flags on the Cray compiler or link to the Cray-optimised DGEMM routine. All we are doing is selecting which kernel version to call rather than how best to compile it. This is not the case for the subsequent accelerator auto-tuning where there are many more options to explore because of the additional complexity of OpenACC and the relative immaturity of the GPU compilers compared to the long-established CPU versions.

The differences between this work and that done in [1] are:

- We set the number of elements by hand (i.e. we adjust *NFLOAT* based on the value of *N*) and benchmark separately for each value of *nel* as we expect that this will be important in the OpenACC versions.
- We benchmark the *vxv* and *mxv*/*vxm* versions of the kernels separately.
- As well as running on a single core, we also run in a mode where all 32 cores on a node are running simultaneously. This better represents the real situation in a parallel run of Nekbone and has a substantial effect on performance due to sharing of cache and memory bandwidth between cores.

We record the best performing kernel for every different combination of *nel*, *N* and case, and store in a file for later use by Nekbone. For the ranges of parameters used here this amounted to over 250 separate entries from over 2000 individual measurements just record the *vxv* data for a single core. The individual performance of these kernels is not particularly interesting as we are mainly interested in the ultimate performance of the Nekbone code, but we present some selected results in Figure 1 and Figure 2 (note the differences in scale). The single-process data in Figure 1 has been multiplied by 32 so it can be compared directly to the 32-core results.

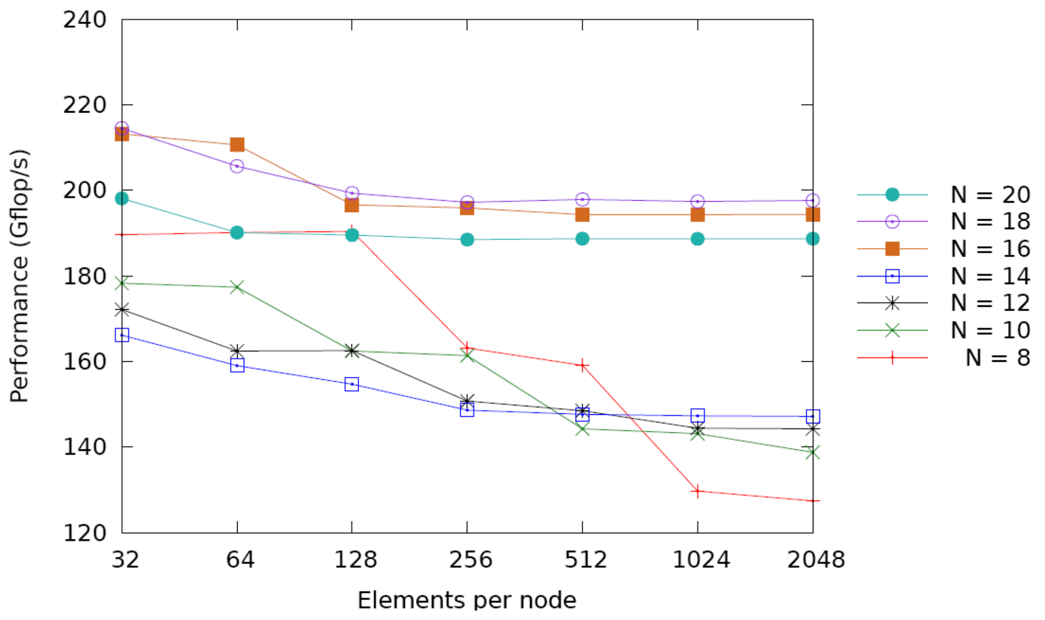

Figure 1: Auto-tuned mxv/vxm kernel performance (case 1) from single core execution

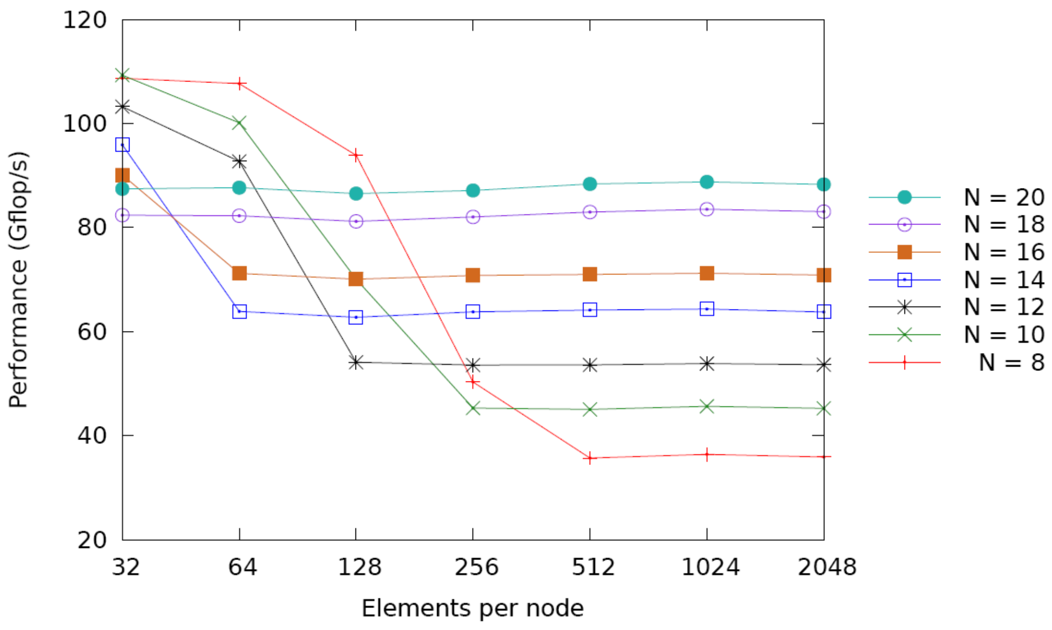

Figure 2: Auto-tuned mxv/vxm kernel performance (case 1) from 32-core execution

This shows the effect of stressing the resources on a shared-memory node by fully populating all the cores: we see a factor of two drop between the single core and 32-core results.

4 CPU performance of Nekbone

In this section we report on the performance of Nekbone measured using the optimal CPU kernels as identified in Section .3.4. In all cases, Nekbone is run on 32 processes completely filling a node of HECToR to enable direct comparison to subsequent GPU results. In the previous CPU tuning section, *nel* referred to the number of elements on each process. In this section, however, *nel* in the graphs refers to the number of elements in the entire parallel calculation (i.e. the total number per node: 32 times larger), again to help with direct comparison to the GPU results.

4.1 Extensions to Nekbone code

In order to enable Nekbone to use the best kernel in all routines, it was extended in two ways:

1. At startup, a new routine **`readtune`** is called which reads in the tuning data from Section 3.4 and stores the results in an array. Note that this data could come from benchmarking the *vxv* kernels, or the *mxv*/*vxm* ones, or from a single or multi-processor run of the benchmark.
2. At runtime, the best kernel is called based on the current values of *nel* and *N*, and the particular case (1, 2 or 3).

The second extension was relatively easy to implement as all calls to the kernels (including the new *mxv*/*vxm* versions) already funneled through a single **`mxm`** stub. Rather than pointing this stub to a fixed matrix-matrix function, a simple switch statement (**`SELECT CASE`** in Fortran) allowed the best kernel to be selected at runtime based on the data from **`readtune`**. The performance overheads of this switching were found to be minimal.

Nekbone already had the functionality to switch between different versions of the element update routines based on a value **`i_ax`** read from file. This could already be used to choose between the scalar (**`ax_e`**) and jammed (**`ax3d`**) versions. Introducing the vector kernel (**`vax_e`**) was relatively straightforward based on the existing scalar kernel and contains code such as:

```
call mxv(nmat,    D,m1,u, m1,ur,m2)
call vxm(nmat*m1, u,m1,Dt,m1,us,m1)
call vxm(nmat,    u,m2,Dt,m1,ut,m1)
```

Here *nmat* = *nel*, *m1* = *N* and *m2* = N^2, so this corresponds to cases 3, 2 and 1 respectively (see Subsection 3.3) with the *u* variables being arrays of matrices (i.e. arrays of elements) and the *D* variables fixed matrices.

In subsequent calls for cases 1 and 2 the result needs to be accumulated into *C*, i.e. the operation is actually of the form $C(i) = C(i) + A(i) * B(i)$. For efficiency in the later OpenACC version we introduced a separate vector function **`addvxm`** to achieve this. However, it differs so trivially from **`vxm`** that there was no need to benchmark it separately so it simply uses the best approach as already determined for **`vxm`** (and in the CPU version it ultimately calls the identical **`mxm`** routine in any case).

It is important to note that although the performance of the scalar and vector versions of Nekbone will be affected by the choices of kernel, that of the jammed version will be unaffected as it comprises completely explicit code with no function calls.

4.2 Performance studies

For subsequent comparison, we used the same settings for *nel* and *N* as employed in the previous study of hand optimisation of Nekbone for accelerators [1]. All the different combinations of options lead to many hundreds of individual runs of Nekbone and generate an enormous amount of data. Here we present the data in a way that aims to answer the following questions:

a. what is the default performance of the three versions (scalar, vector and jammed) of Nekbone?
b. can the scalar and vector versions be improved by tuning, and if so how much?
c. how sensitive are these results to the details of how the kernel benchmarking as performed?
d. which of the three versions (tuned scalar, tuned vector or jammed) is the fastest overall?

It is important for question (a) to define what "default" means. Although we could ascribe enormous benefits to auto-tuning be selecting a very slow "default" versions, this would not be a fair comparison. Given that the BLAS library is widely accepted as being the best implementation of linear algebra available, we define "default" as performing all matrix-matrix operations with DGEMM. This is simply achieved by supplying a dummy file to **`readtune`** that lists DGEMM as the optimal routine for every parameter choice.

4.3 Default performance

The default performance of Nekbone for different choices of the spectral order *N* (8, 10, , …, 18, 20) and the number of elements per node *nel* (32, 64, …, 1024, 2048) are shown in Figure 3 (scalar), Figure 4 (jammed) and Figure 5 (vector). To enable these to be more easily compared we plot the percentage difference (either faster or slower), compared to the jammed version, of the scalar version in Figure 6 and the vector in Figure 7.

It is clear that that the performance characteristics depend very strongly on *N*; for *N* = 8, 10 or 12 the jammed version performs best for all values of *nel*. For larger values of *N*, the scalar version always outperforms jammed, presumably because the DGEMM routine is highly optimised for this case. For these large values of *N* vector also outperforms jammed for smaller values of *nel*, but performs worse for large values (the crossover point being around 256 or 512). This is presumably a memory or cache effect: the vector algorithm involves reloading all the element data each time a kernel is called, so will suffer from limited memory bandwidth when there is a lot of data to be processed. The scalar and jammed versions do not have this problem as they complete all the operations required on a single element before moving on to the next.

It is quite interesting at this stage that no single algorithm is the best across all the values of *N* and *nel*, although the vector version is almost never the fastest.

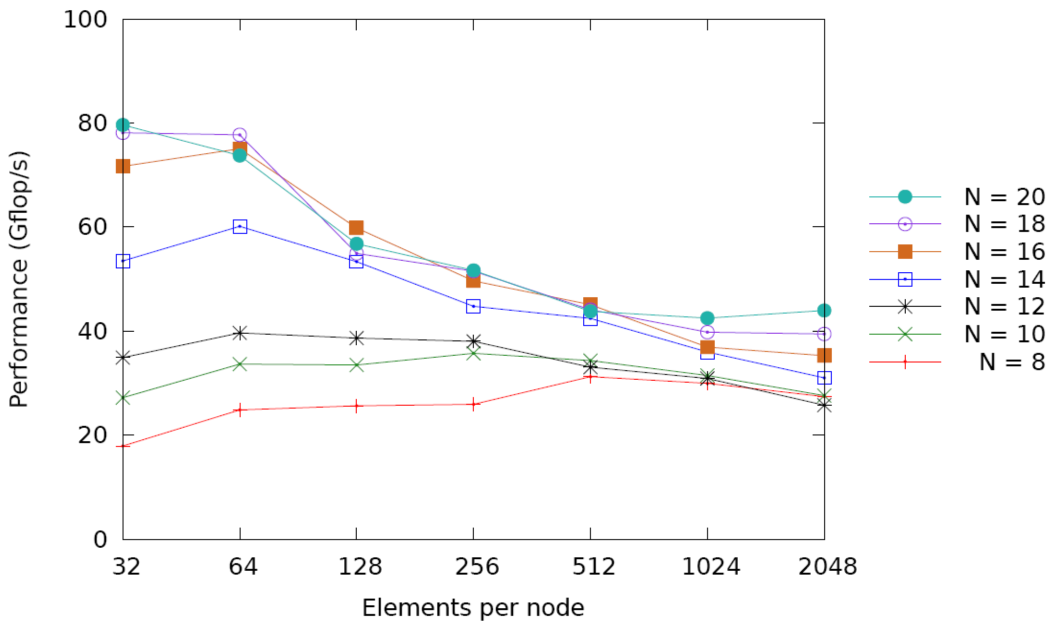

Figure 3: Default performance of scalar version of Nekbone

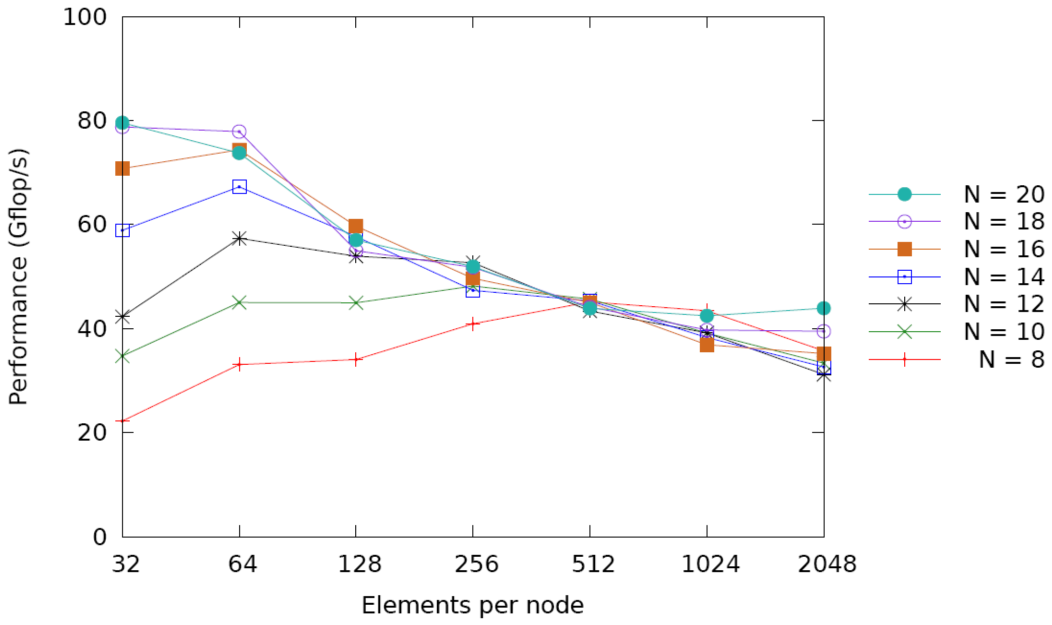

Figure 4: Default performance of jammed version of Nekbone

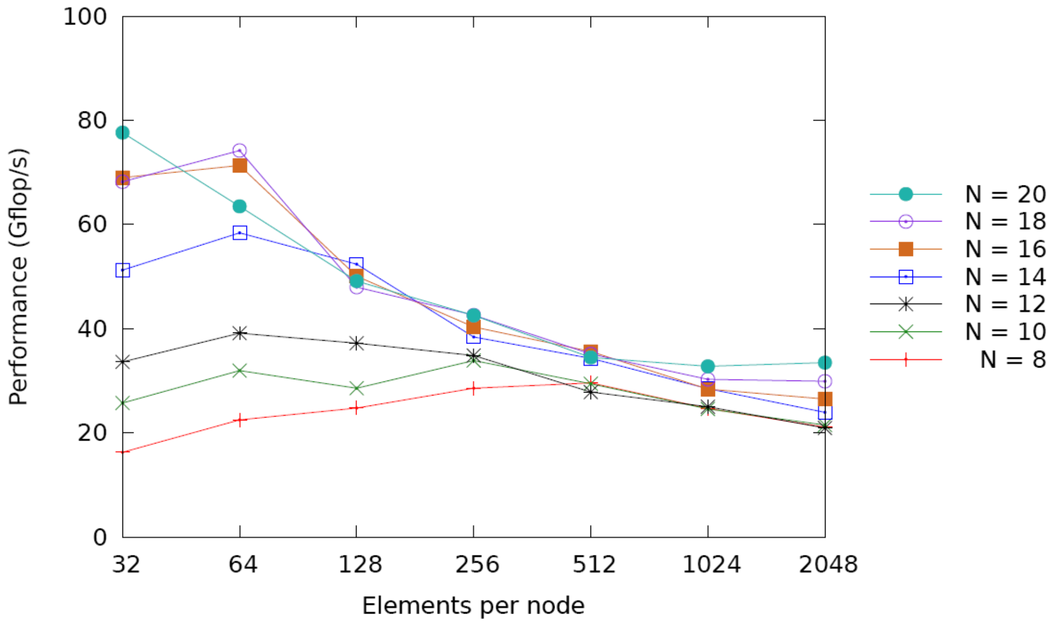

Figure 5: Default performance of vector version of Nekbone

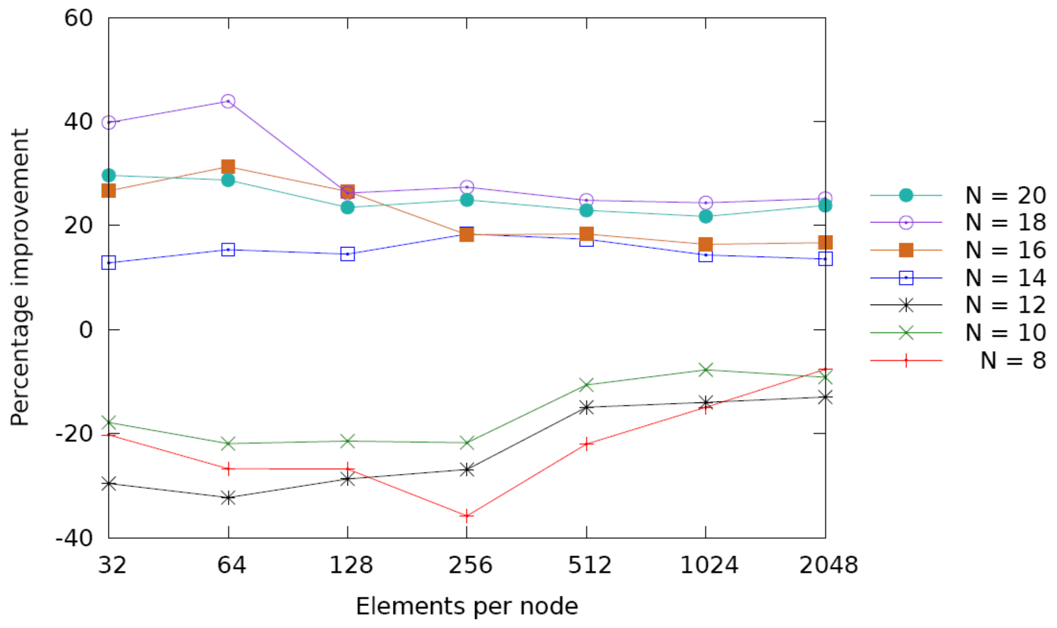

Figure 6: Percentage improvement of default scalar version vs jammed

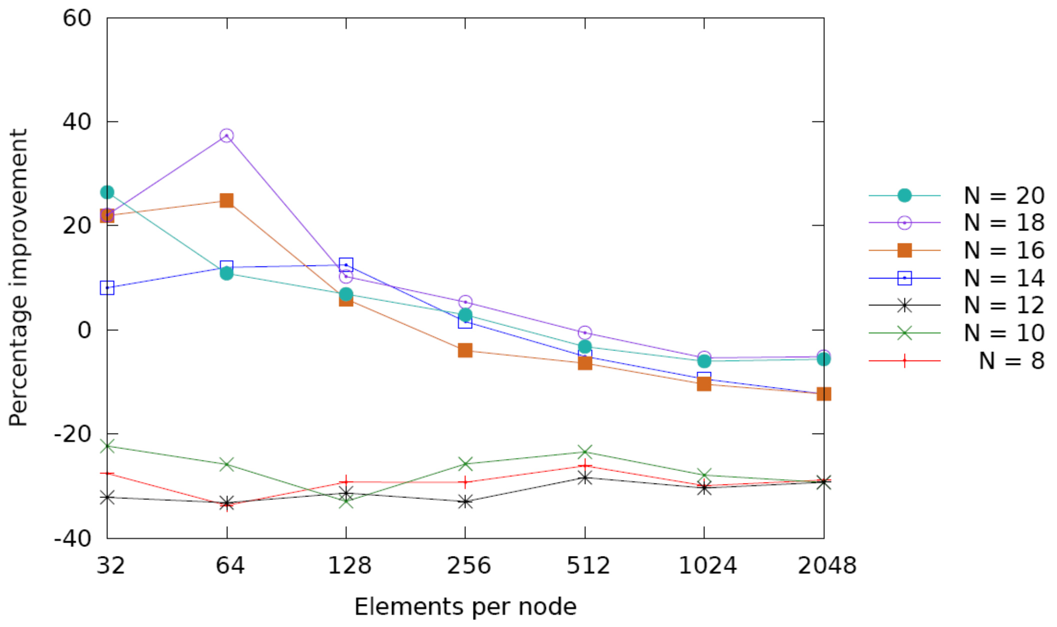

Figure 7: Percentage improvement of default vector version vs jammed

4.4 Auto-tuned results

To try and obtain the best possible scalar and vector performance, we re-ran Nekbone with tuning results obtained by running the *mxv*/*vxm* kernels on 32 cores of HECToR, which is the closest match to the real operation of Nekbone (indeed, for the scalar version these are the actual routines called). The performance is shown in Figure 8 (scalar) and Figure 9 (vector). It appears that the performance is improved significantly, especially for the smaller values of *N* where the previous results from the jammed version have already indicated that the default (i.e. DGEMM) routine may not be optimal.

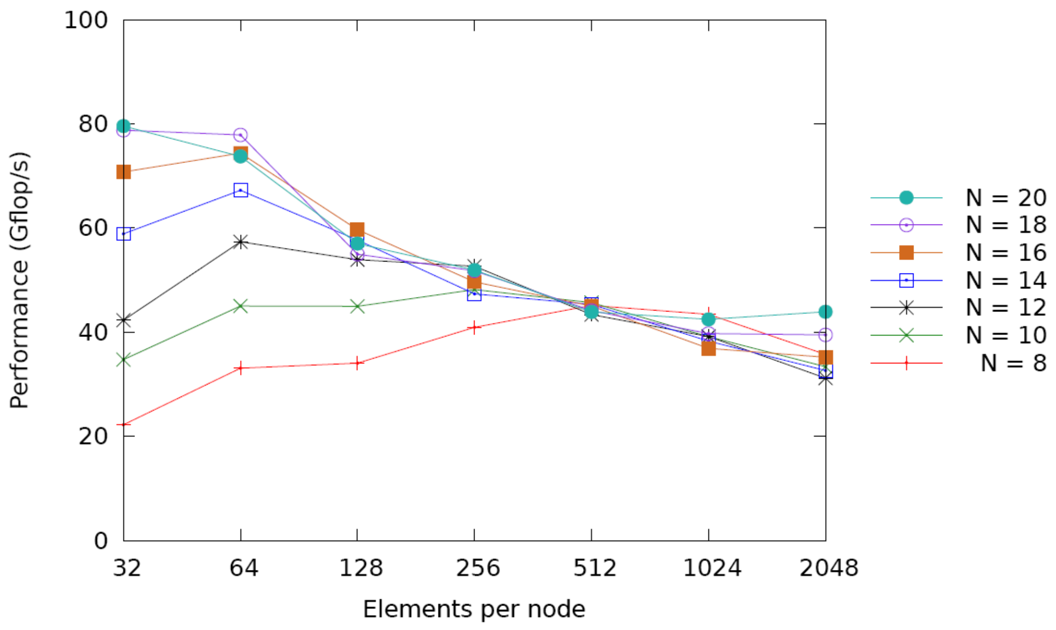

Figure 8: Auto-tuned performance of scalar version of Nekbone

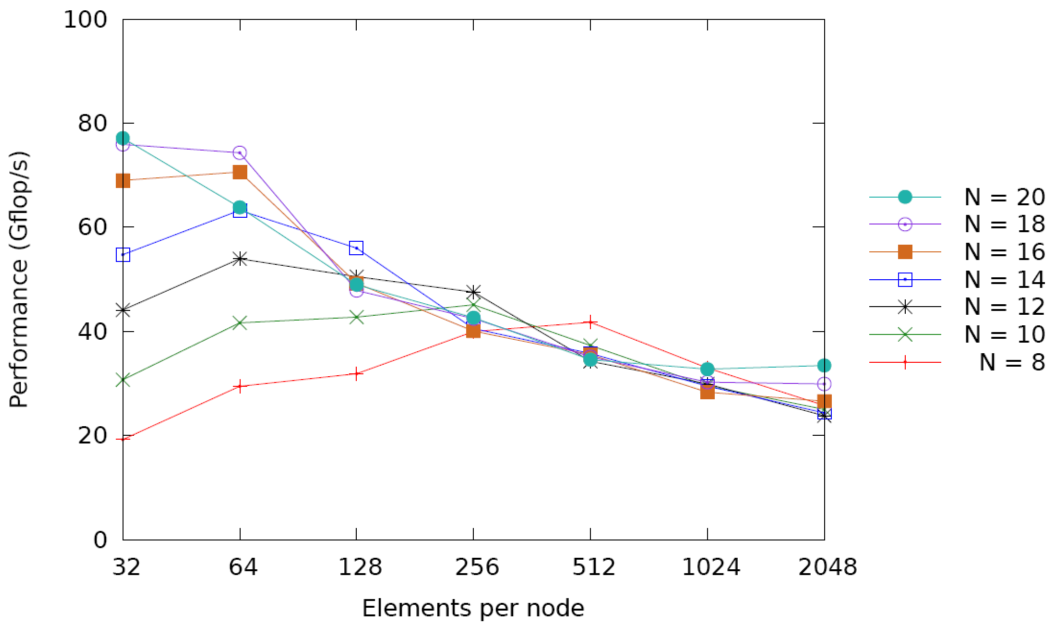

Figure 9: Auto-tuned performance of vector version of Nekbone

To better understand what auto-tuning is doing see, for example, Figure 10 which shows the improvement coming from auto-tuning for the vector version of Nekbone. For all values of *N* up to and including 14 we see improvements across all values of *nel*. There is essentially no improvement at any value of *nel* for all higher values of *N*, i.e. DGEMM is always optimal in this range. There is one apparently anomalous improvement for *N*=18 at the smallest value of *nel*, but this does not change what is overall a very clear picture. The equivalent results for the scalar version (not plotted) are qualitatively the same, showing exactly the same trends but with slightly greater performance improvements across the board.

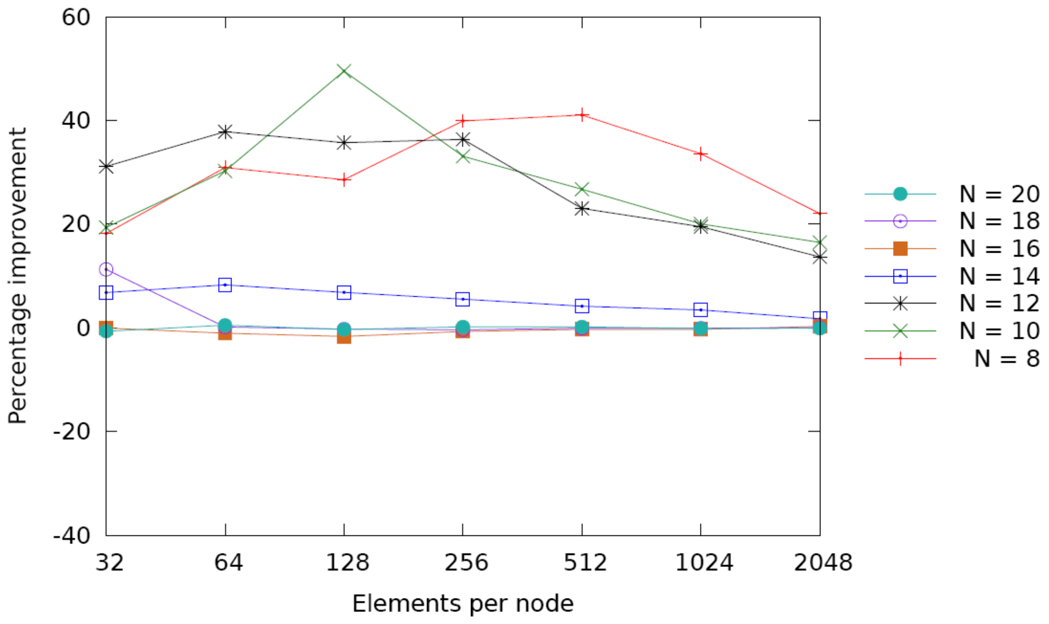

Figure 10: Percentage improvement of auto-tuned vector version vs default

We also repeated these runs using tuning data collected in the same way as in [1], i.e. a single processor run of the original *vxv* kernels (as opposed to the multi-processor *mxv*/*vxm* data above). The two results are virtually indistinguishable apart from a very few isolated cases where we see some small advantage from using the *mxv*/*vxm* data.

4.5 Fastest version overall

To find out what the fastest version is across the scalar, vector and jammed versions we must compare Figure 8 and Figure 9 with Figure 10. We again compute the percentage improvement (or reduction) in performance between the best auto-tuned versions of the scalar and vector algorithms and the jammed version (which is unaffected by tuning); the results are shown in Figure 11 and Figure 12.

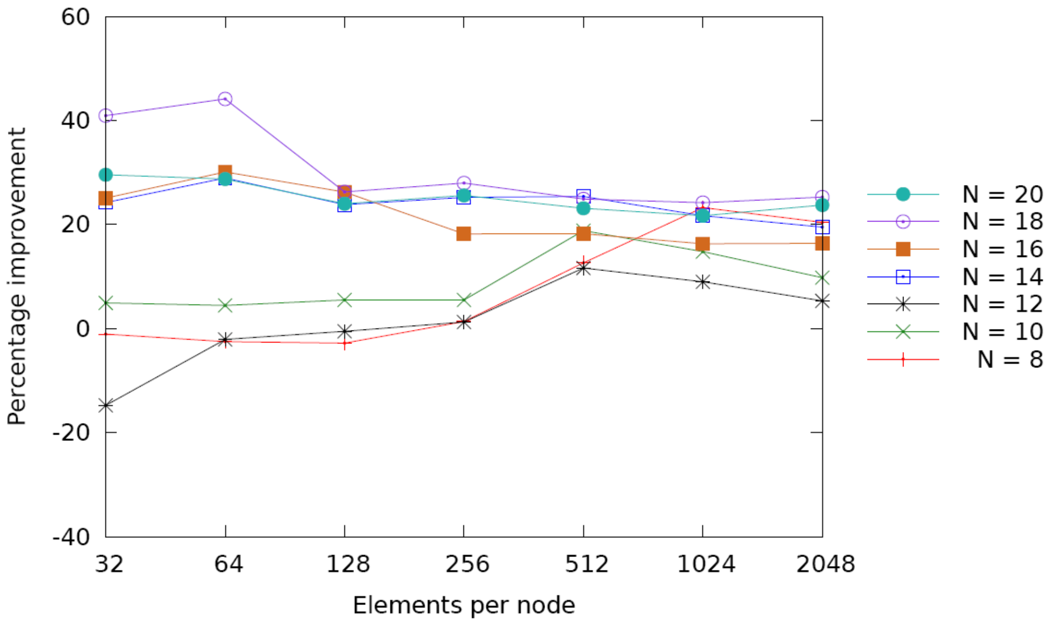

Figure 11: Percentage improvement of auto-tuned scalar version vs jammed

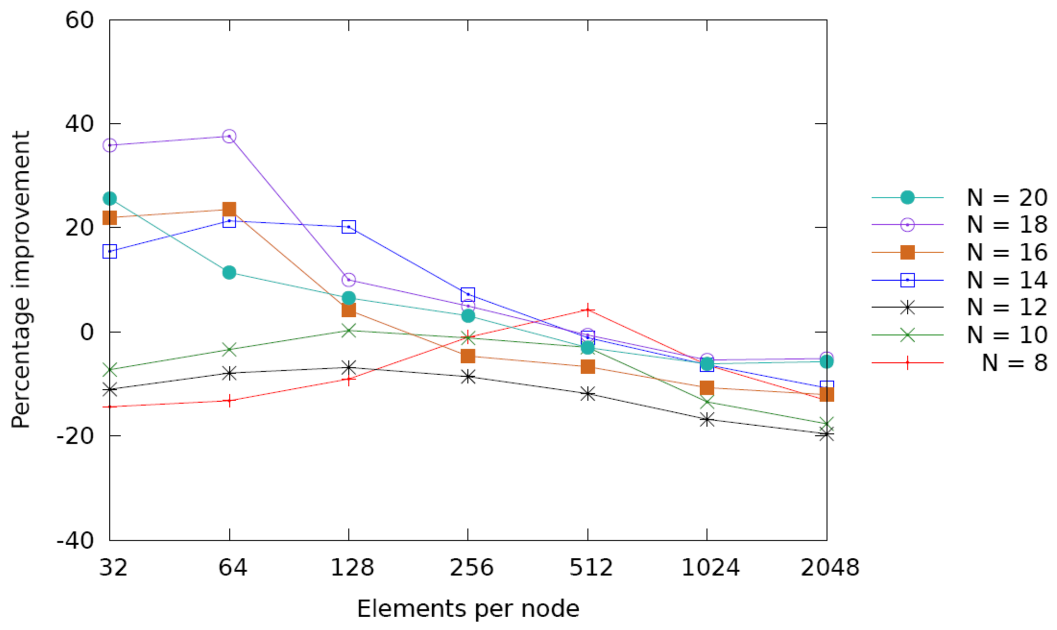

Figure 12: Percentage improvement of auto-tuned vector version vs jammed

Except for a few points at small *N* and *nel*, the scalar version matches or outperforms the jammed version in all cases; only a single point is significantly slower.

For the vector version, we only see improvements over the jammed version at small *nel* (e.g. 256 or less) and large *N* (e.g, 16 or more). Despite this, however, the performance is never worse than 20% below the the jammed version which indicates that the effects of increased memory traffic are not too severe.

4.6 Conclusions

The results of this section are relatively encouraging, showing that auto-tuning can have significant benefits in certain regions of the parameter space. Here, in cases of small *N* (8, 10, 12 and 14) we were able to improve performance above the default by using some alternative to the DGEMM library function. However, for larger values of N (16, 18 and 20) no improvement was possible.

Given that only the vector and jammed versions are viable candidates for GPU acceleration, Figure 12 gives some hope that the approach of separating out small kernels and auto-tuning them may be a useful approach in some parameter regions (here, large *N* and small *nel*) despite the fact that this vector version makes higher demands on the memory system. However, these improvements were possible because existing library routines (here, DGEMM) were able to significantly outperform the handwritten jammed code. Unless similarly optimised libraries can be found for the vector versions on the GPU it seems unlikely that we will be able to reproduce the modest success of these CPU results.

5 OpenACC kernels

We now aim to auto-tune GPU kernels for Nekbone. All performance data is from a single GPU and therefore directly comparable to the previous 32-core CPU results.

5.1 Reproducing previous study

In [1], results for auto-tuned OpenACC versions of the *vxv* kernels were presented. However, those results are not directly applicable to this study as they were obtained using the PGI compiler and we now want to investigate the Cray compiler. In addition, they employed an auto-tuning framework developed by the University of Edinburgh School of Informatics (see [5] which includes our results as part a wider study of a range of kernels) and we now want to evaluate the CRESTA auto-tuning framework [3].

The previous study used kernels written in C to enable easy integration with the existing Informatics framework. As Nekbone is written in Fortran, and the CRESTA framework is language-neutral, we rewrote the NekGemm benchmark in Fortran. Previously all array dimensions (i.e. the values of *N* and *nel*) were compile-time constants, a decision partly based on limitations of C. However, modern Fortran deals equally well with static and dynamic arrays so we wrote two versions of the kernels: one (called *DIMARGS*) where array dimensions are passed as subroutine arguments at runtime, the other (*NODIMARGS*) where dimensions are compile-time constants Note that we had to disable subroutine inlining: by default the kernels were inlined and the Cray compiler was then clever enough to recognise that subroutines were being called with constant parameter values. Although this is a useful feature in a real code, it led to both versions being effectively identical which is not what we want in this study.

5.2 Auto-tuning code and script

For completeness, the script used for the auto-tuning is shown in Section 8.1; see [3] for the full syntax of the tuning file. Although we can use the script to tune how the code for the kernels is generated, it is the user's responsibility to write as many different versions as possible, i.e. to supply a tunable code. We did this via a combination of multiple subroutines as was done previously for the CPU version (they appear here as different choices for **algorithm)** and multiple possible options for the OpenACC directives passed as **#defines** via **-D** flags to the Makefile (in the script these are values such as **NUM_GANGS** and **VECTOR_LENGTH**). The particular test case, e.g the value of *N* (**NPSEC** in the script), is selected in the same way. To illustrate how the tuning script and the application interact a representative piece of tunable code, e.g. for algorithm 106, is shown in Section 8.2.

As well as writing many different versions of the kernels, we did initial experiments with calling the DGEMM routine from the cuBLAS library. The performance was always very poor as any parallelisation has to be performed within a single matrix and *N* is simply not big enough for this to be efficient; *N* has to have values around 1000 for this to be competitive. Launching many separate DGEMM calls simultaneously using the OpenACC **async** clause was also not successful, perhaps because the number of concurrent calls is currently limited to 16.

Using the script, the auto-tuning framework runs the code many times and reports the combination of parameters that produces the best results (here the maximum performance). The data for all the individual runs is also logged for later reference.

5.3 Performance results

The results are show in Figure 13 (dimensions as arguments) and Figure 14 (constant dimensions); note the difference in scales between the two. The features to note are:

- Default performance is substantially better in all cases when the compiler knows the array dimensions at compile time.
- Auto-tuning dramatically increases performance in all cases, but there is always a substantial performance benefit from knowing dimensions at compile time.

- The default performance is less than previously achieved with the PGI compiler, quite substantially so for the larger values of *N*.
- The auto-tuned performance is always better than previously obtained.

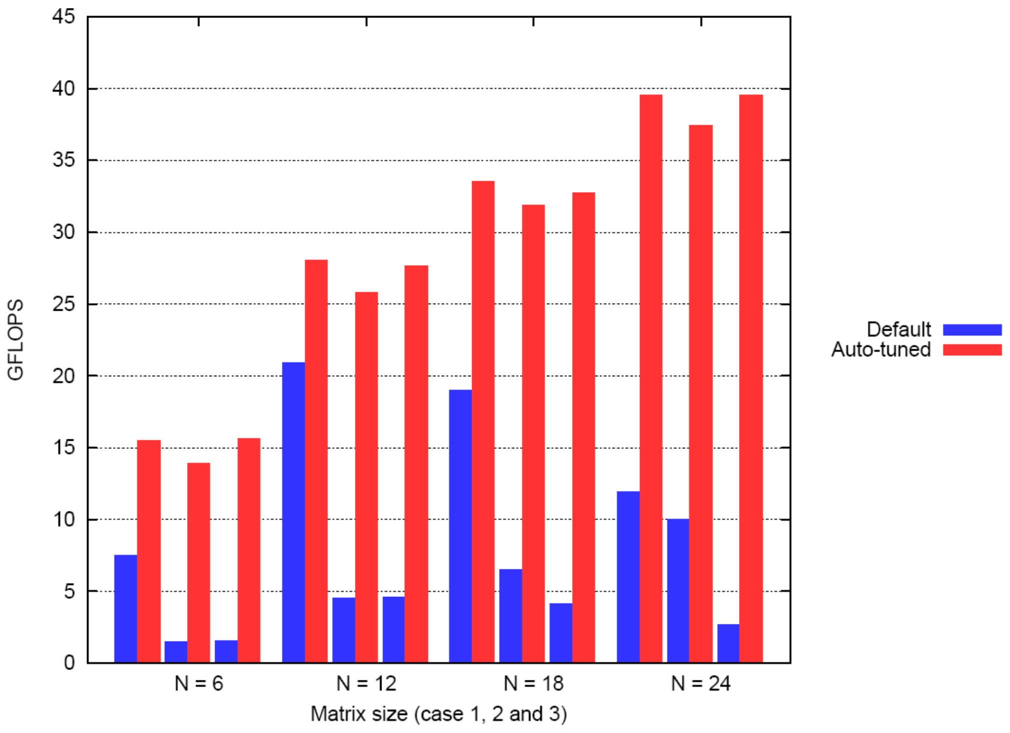

Figure 13: Performance of kernels with array dimensions as subroutine arguments

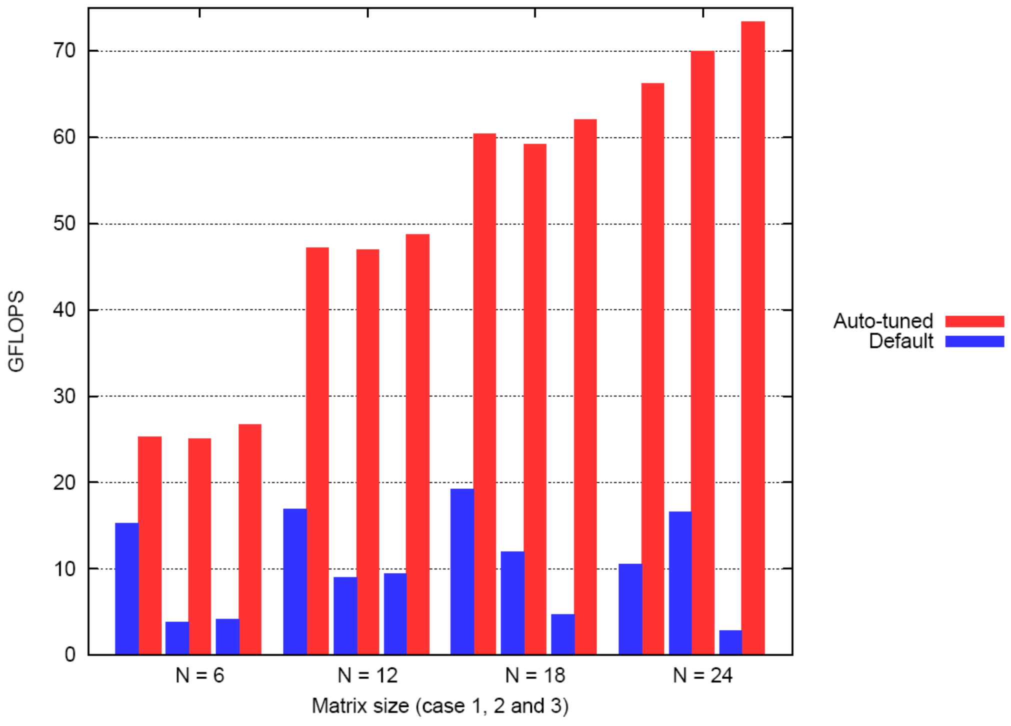

Figure 14: Performance of kernels with array dimensions as compile-time constants

The fact that the default performances are so different between Cray and PGI is perhaps not that surprising as the two compilers take totally different approaches to OpenACC. The PGI compiler focuses on the `kernels` directive, whereas Cray focuses on the `parallel` directive. Based on these results, for all future studies we decided to use versions of the subroutines where array dimensions are fixed at compile time.

5.4 Tuning based on the number of elements

The approach taken above was to set the total memory consumption and pick the number of elements based on this value. This means that the number of elements drops significantly with increasing *N*; for example, for the settings used here (*NFLOAT*=266240) the number of elements is as small as 19 for *N* = 24. As mentioned before, this is not an issue for the CPU code where performance tuning is relatively insensitive to *nel*. However, for the GPU we expect performance characteristics to vary significantly with *nel* and therefore want to both set it explicitly and tune individually for each value. Note that we are still working with the original *vxv* kernel benchmarks.

To allow for direct comparison with [2], we now use the same values of *N* and *nel* as used there. The results for cases 1 and 2 are presented in Figure 15 and Figure 16 (default), and Figure 17 and Figure 18 (auto-tuned); case 3 is very similar to case 2 and is not shown.

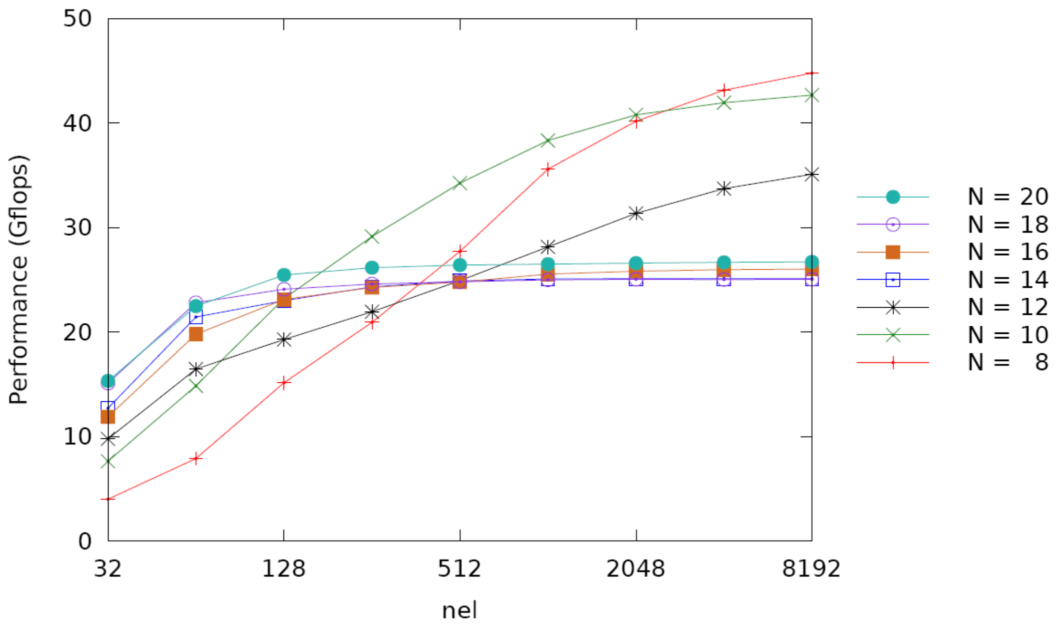

Figure 15: Default performance of case 1 vs number of elements

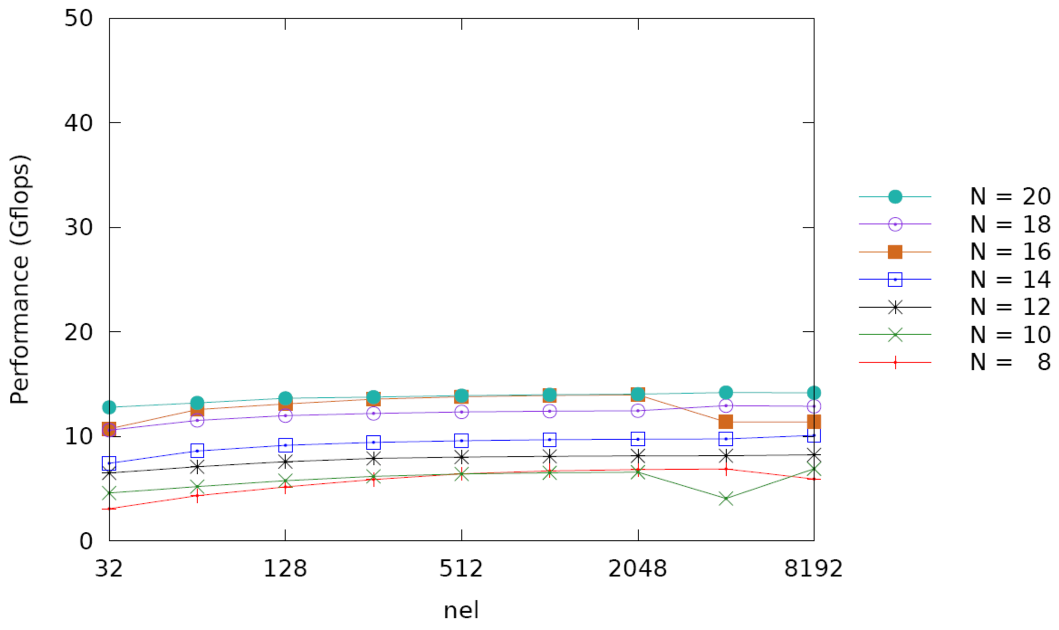

Figure 16: Default performance of case 2 vs number of elements

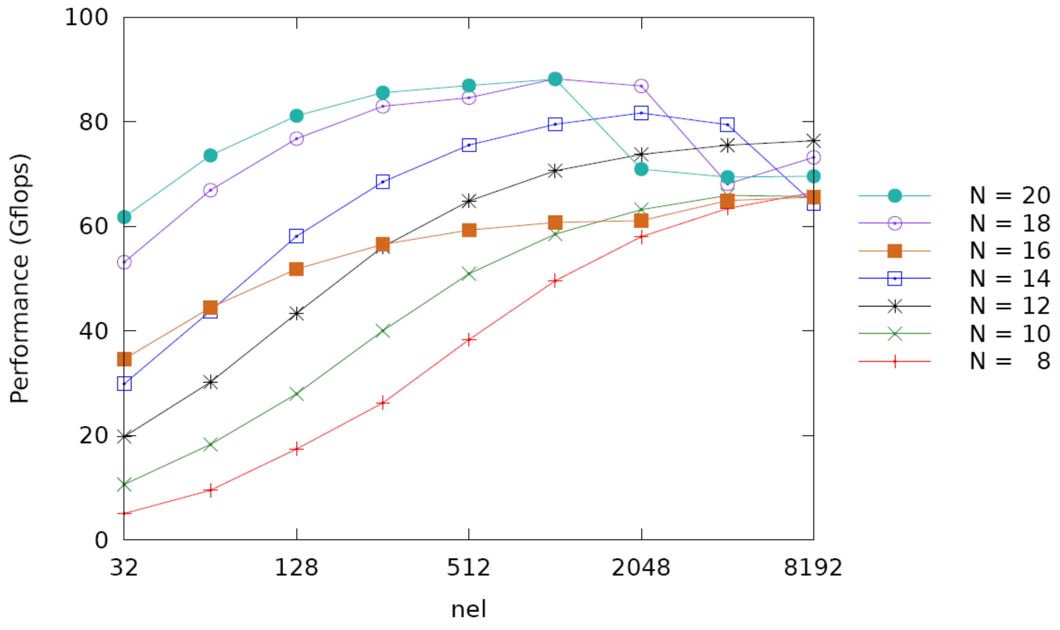

Figure 17: Auto-tuned performance of case 1 vs number of elements

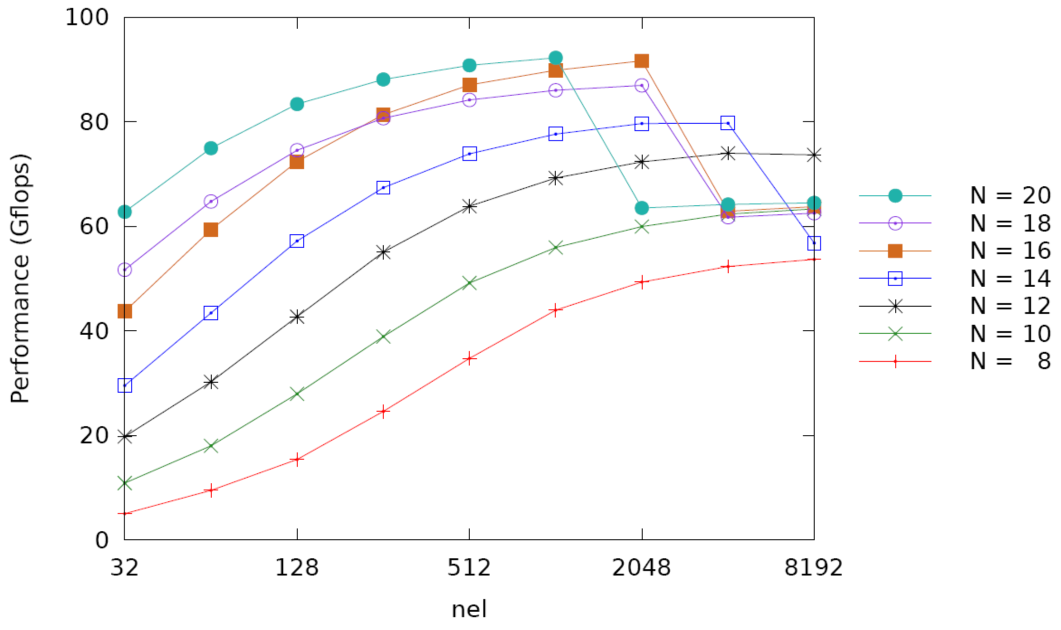

Figure 18: Auto-tuned performance of case 2 vs number of elements

They show a very consistent picture:

- significant benefits from auto-tuning in all situations;
- auto-tuned performance is similar for all cases;
- any differences in default performance (e.g. cases 1 and 2 differ significantly) is removed by auto-tuning.

Overall, these results for auto-tuning of the kernel benchmarks are very encouraging.

6 Performance of Nekbone

We are now in a position to take the optimal parameter settings from Section 5.4 and introduce them into an accelerated version of Nekbone to see what effect the kernel auto-tuning has on overall application performance.

6.1 Developing a tunable OpenACC version of Nekbone

As part of the work done in [2], a fully functioning OpenACC version of NekBone was developed based on the jammed kernel **`ax3d`**, so this was the obvious starting point. Only two issues major were encountered:

1. Some of the optimisations introduced for the OpenACC version of **`ax3d`** require different lookup tables; these are initialised instead of the default tables use by **`vax`** causing the code to fail. The solution was simply to ensure that the correct setup routines are called based on the value of **`i_ax`** (see Section 4.1).
2. The **`ax3d`** version does not call any subroutines, whereas **`vax`** calls subroutines such as **`vlocal_grad3`** which subsequently call the *mxv*/*vxm* vector kernels. This meant that data scoping was more complicated, especially as the original code worked with different lookup tables. As had already been done carefully elsewhere in Nekbone, it was important to ensure that extended data regions were used to minimise data copying at entry and exit from accelerated regions.

Unlike the CPU version, switching between different optimised routines at runtime is not simply a matter of calling the correct precompiled version. A single function can be compiled in many ways depending on the OpenACC settings, so some coding work would be needed to enable multiple versions of the same subroutine to exist in the same program, and the best version called at runtime as appropriate. There has not yet been sufficient time to do this, so at present all OpenACC settings are the same regardless of the values of *N* and *nel*. Given the structure of the code, however, we are able to use different settings for each of the three cases.

6.2 Initial performance results

This work is in its early stages, so as described above we do not yet change the OpenACC settings dynamically within the application. In addition, we are currently obtaining tuning parameters from the *vxv* kernels (as presented in Section 5.4) and not the *mxv*/*vxm* kernels that are actually called in Nekbone. Although we saw in Section 4.4 that this had no significant impact for the CPU version, there is no reason to assume that this will also be true on the GPU.

The performance results obtained in [2] serve as useful reference values, and are shown in Figure 19; remember that these come from a hand-tuned version of the jammed code (i.e. using **`ax3d`**). The maximum value of *nel* is often smaller than the previous value of 8192 used in the kernel benchmarks because Nekbone uses more memory, and we run into memory limits on the GPU (especially at large *N*). The default performance we obtain for the vector version (i.e. calling **`vax`**) is shown in Figure 20 (note the difference in scale).

To illustrate the effects of parameter tuning, we picked a set of OpenACC parameters that gave good kernel performance for all values of *N* and *nel* for each case. Surprisingly, this resulted in extremely poor performance for Nekbone which was worse than the default settings in all cases; the reasons for this are not yet clear. It could be because the parameters came from the *vxv* kernels, or because the compiler is treating the kernels differently when placed within a larger application.

However, by selecting non-default OpenACC parameters by hand (effectively using trial and error) we were able to obtain good performance for the vector version – see Figure 21. The performance here is close to that of the hand-tuned jammed version and suggests that auto-tuning the entire vector version of Nekbone (rather than the vector kernels in isolation) could result in even better performance.

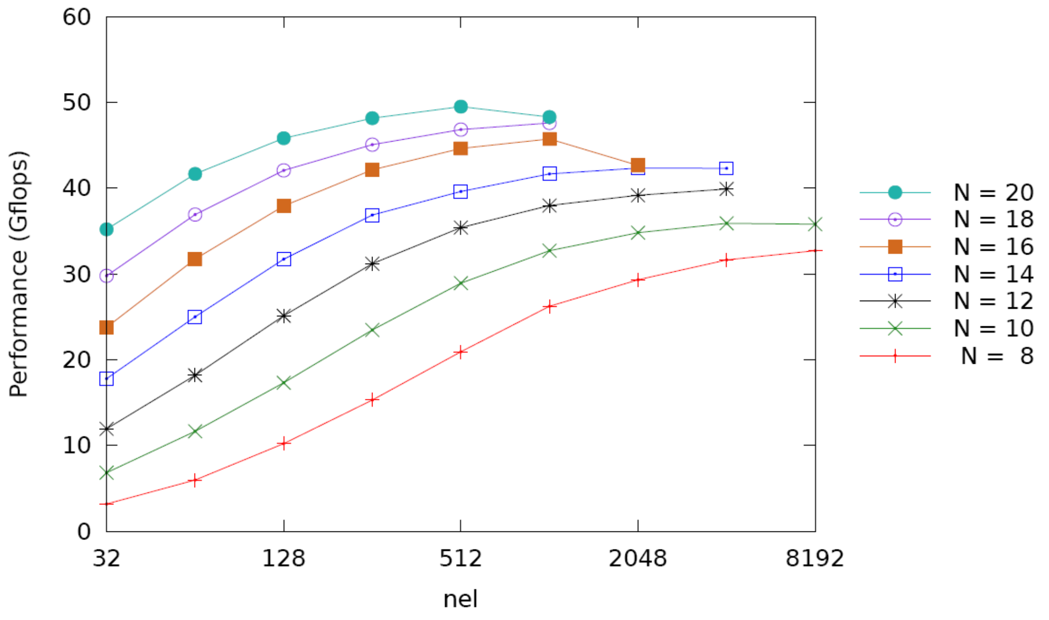

Figure 19: Performance of hand-tuned OpenACC jammed Nekbone (data from [2])

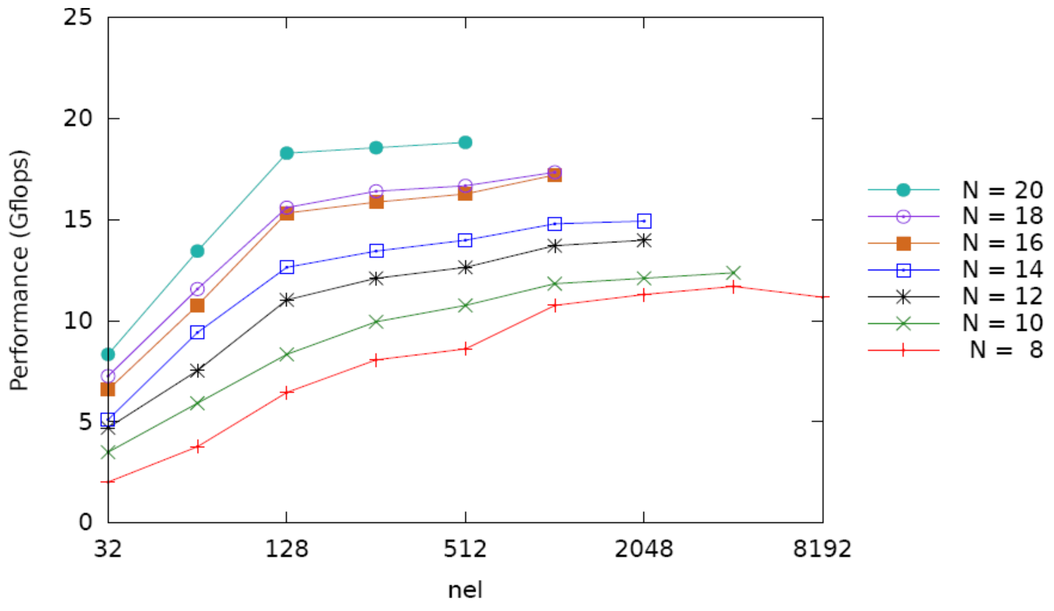

Figure 20: Performance of OpenACC vector Nekbone with default settings

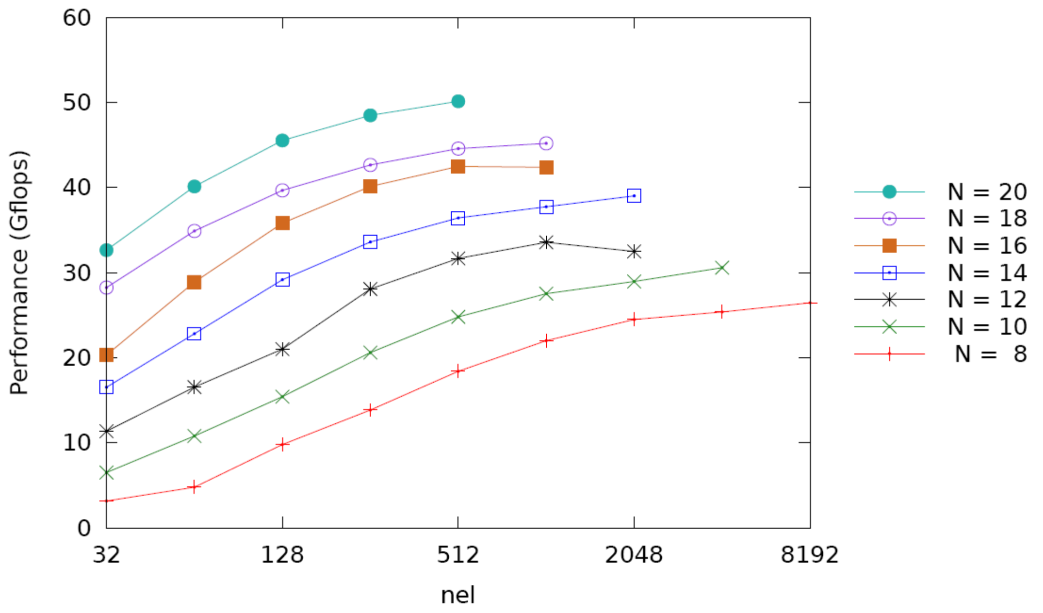

Figure 21: Performance of OpenACC vector Nekbone with hand-tuned settings

6.3 Summary

In terms of application performance, the results in Figure 21 are encouraging when compared to the hand-optimised results in Figure 19. It is surprising that, unlike for the CPU version, the auto-tuned OpenACC kernel parameters resulted in very poor performance for the Nekbone application and we had to resort to hand-tuning. There was not sufficient time available to create a fully tunable version of Nekbone, although this should not be difficult to develop in the future.

There are several possible explanations for this behaviour:

1. The OpenACC settings from Section 5.4 are currently determined from the original *vxv* kernels and, despite the results from the CPU studies, might not be appropriate on the GPU for Nekbone's *mxv*/*vxm* kernels.
2. Perhaps the kernel auto-tuning search space was too small; given the number of combinations of *N*, *nel* and case, we had to restrict the ranges of some of the parameters to ensure that the time spent on auto-tuning was not too large.
3. The application performance may be strongly affected by factors not present in the kernel benchmarks, e.g. the presence of many other large arrays and extra operations such as matrix addition.
4. The compiler may be making different decisions when compiling OpenACC code within Nekbone compared to the kernel benchmark.

Despite the current limitations of our approach, however, we do see many potential benefits from auto-tuning compared to selecting default values. Unlike on the CPU, for the GPU it appears that these parameters need to be determined from running the full application rather than from isolated kernels. Although possible in principle, this will be a lot more difficult in practice as the application runtime will always be much larger than the kernels so we will not be able to explore such a large region of parameter space.

7 Conclusions and further work

This study deliberately took a simple but methodical approach to auto-tuning the Nekbone application:

- gain experience of the application and its auto-tuning on the familiar CPU;
- develop a version calling vector kernels suitable for auto-tuning on the GPU;
- determine OpenACC parameters by auto-tuning the existing kernel benchmark;
- use these parameters in the vector version of Nekbone;
- compare performance to the existing hand-tuned version of Nekbone.

The prototype CRESTA auto-tuning framework was used for the OpenACC work and shown to be very effective and easy to use; a number of suggestions for enhancements have been passed on to the authors. Auto-tuning worked very well on the GPU kernel benchmark, increasing the performance significantly above that obtained with default settings. Suprisingly, no benefits were observed when these settings were applied to the full Nekbone application, with performance less than the default settings. However, good performance was achieved by hand-tuning these parameters.

The GPU auto-tuning work has not yet been completed and there are a number of areas of further work (in approximate order of importance):

- repeat studies in Section 5.4 using the more representative *mxv*/*vxm* kernels;
- auto-tune the *mxv*/*vxm* kernels within Nekbone application and not in isolation;
- extend Nekbone so it automatically calls the fastest vector kernel in all cases;
- extend the auto-tuning parameter space for the *mxv*/*vxm* kernels;
- investigate auto-tuning the existing hand-tuned jammed version of Nekbone.

Despite the somewhat limited scope of the current study, it has clearly demonstrated that auto-tuning can be beneficial for OpenACC-accelerated applications running on GPUs. The default settings chosen by current compilers can be far from optimal, even if they know all the application parameter settings (e.g. array dimensions) at compilation time.

Once the auto-tuning work has been completed for Nekbone, it would be interesting to extend it to the full Nek5000 application. The same approach could also be used for other CRESTA applications, e.g. GROMACS. A hand-tuned GPU version of GROMACS already exists, with core kernels implemented in CUDA. It would be interesting to see if the CUDA code's performance can be matched using OpenACC, and also how much improvement is gained from the CRESTA auto-tuning framework.

8 Appendix

8.1 Auto-tuning script

```
! File is called: nekgemm.tune

begin configuration
 begin tune
  mode: tune
! do not tune on NEKCASE, NMAT !
  scope: VECTOR_LENGTH algorithm NSPEC
  target: max
  metric-source: file
  postrun-metric-file: output.$run_id
  metric-placement: lastregexp
  metric-regexp: tune run metric +(\S+)
 end tune
end configuration

begin parameters
 begin typing
  label NUM_GANGS
  label NUM_WORKERS
  int VECTOR_LENGTH
  label SCALAR_REDUCTION
  label DIMARGS
  int NEKCASE
  int NSPEC
  int algorithm
 end typing
 begin constraints
  range NUM_GANGS none default none
  range NUM_WORKERS none 1 2 4 8 16 32 default none
  range VECTOR_LENGTH 64 128 256 default 128
  range SCALAR_REDUCTION no yes default yes
  range DIMARGS no yes default no
  range NEKCASE 1 2 3 default 1
  range NSPEC 6 12 18 24 default 6
  range algorithm 101 104 105 106 107 108 121 122 131 132  default 101
! Put runtime parameters first in this list
  depends algorithm,VECTOR_LENGTH,NSPEC,NEKCASE
 end constraints
!if you change something here, you need to re-compile
 begin collections
  BUILD: NUM_GANGS NUM_WORKERS VECTOR_LENGTH SCALAR_REDUCTION DIMARGS
NEKCASE NSPEC
 end collections
end parameters

begin build
 command: make clean; make NUM_GANGS=$NUM_GANGS
          NUM_WORKERS=$NUM_WORKERS VECTOR_LENGTH=$VECTOR_LENGTH
          NEKCASE=$NEKCASE NSPEC=$NSPEC
end build

begin run
 command: aprun -n1 -N1 ./nekgemm $algorithm > output.$run_id
 validation-source: command
 validation-command: cat output.$run_id
 validation-failure-mode: warning
end run
```

8.2 Representative auto-tuning kernel

```
SUBROUTINE accgemm106(a, b, c, n1arg, n2arg, n3arg, nmatarg)

  INTEGER :: n1arg, n2arg, n3arg, nmatarg
  INTEGER :: n1loop, n2loop, n3loop, nmatloop
  INTEGER :: i, j, k, imat

  DOUBLE PRECISION, DIMENSION(n1arg, n2arg, nmatarg), INTENT(IN)  :: a
  DOUBLE PRECISION, DIMENSION(n2arg, n3arg, nmatarg), INTENT(IN)  :: b
  DOUBLE PRECISION, DIMENSION(n1arg, n3arg, nmatarg), INTENT(OUT) :: c
  n1loop=n1arg
  n2loop=n2arg
  n3loop=n3arg
  nmatloop=nmatarg

#ifdef SCALAR_REDUCTION
  DOUBLE PRECISION :: tmp
#endif

!!$ I might choose to specify num_gangs and/or num_workers,
!!$ I will always specify vector_length.

  !$acc parallel present(a,b,c) private(i,j,k) &
#ifdef SPECIFY_NUM_GANGS
  !$acc    num_gangs(NUM_GANGS) &
#endif
#ifdef SPECIFY_NUM_WORKERS
  !$acc    num_workers(NUM_WORKERS) &
#endif
  !$acc    vector_length(VECTOR_LENGTH)
  !$acc loop gang collapse(2)
  DO imat = 1, nmatloop
     DO j = 1, n3loop
        !$acc loop vector
        DO i = 1, n1loop
#ifdef SCALAR_REDUCTION
!!$ Do the reduction into a loop-private scalar
           tmp = 0
#else
!!$ Do the reduction directly into the array
           c(i,j,imat) = 0
#endif
           DO k = 1, n2loop
#ifdef SCALAR_REDUCTION
              tmp         = tmp         + a(i,k,imat) * b(k,j,imat)
#else
              c(i,j,imat) = c(i,j,imat) + a(i,k,imat) * b(k,j,imat)
#endif
           END DO
#ifdef SCALAR_REDUCTION
           c(i,j,imat) = tmp
#endif
        END DO
     END DO
  END DO
  !$acc end parallel

END SUBROUTINE accgemm106
```

9 References

[1] Compiler Support for Exascale, CRESTA Project Deliverable D3.5.1.

[2] OpenACC Acceleration of Nek5000, a Spectral Element Code, presented at "Exascale Applications and Software Conference" (2013).

[3] Domain Specific Language (DSL) for expressing parallel auto-tuning, CRESTA Project Deliverable D3.6.2.

[4] Nek5000 project web page http://nek5000.mcs.anl.gov/.

[5] Alberto Magni, Dominik Grewe and Nick Johnson, "Input-Aware Auto-Tuning for Directive-based GPU Programming", proceedings of GPGPU6 (2013).