# D3.5.1 –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:** | M10 |
| **Submission date:** | 31/07/2012 |
| **Project start date:** | 01/10/2011 |
| **Project duration:** | 36 months |
| **Deliverable lead organisation** | UEDIN |
| **Version:** | 1.0 |
| **Status** | Final |
| **Author(s):** | David Henty (UEDIN) |
| **Reviewer(s)** | Michael Schliephake (KTH), David Lecomber (ASL) |

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

# Version History

| Version | Date | Comments, Changes, Status | Authors, contributors, reviewers |
|---|---|---|---|
| 0.1 | 26/06/12 | First version of the deliverable | David Henty (UEDIN) |
| 0.2 | 27/06/12 | OpenACC added | David Henty (UEDIN) |
| 0.3 | 28/06/12 | Auto-tuning added | David Henty (UEDIN) |
| 0.4 | 01/07/12 | First draft for review | David Henty (UEDIN) |
| 0.9 | 24/07/12 | Updated re reviewers' comments | David Henty (UEDIN) |
| 1.0 | | Final version after proof reading | |

# Table of Contents



# Index of Figures

# 1 Executive Summary

A survey of compiler technologies relevant to exascale was performed in a previous CRESTA deliverable [1]. Two particular issues that were identified were the growing requirement for compilers to support CPU accelerators, and the possible advantages of auto-tuning to produce better performing code on today's increasingly complex and heterogeneous processors. Motivated by this, we study key kernels from one of the CRESTA co-design applications, Nek5000. We investigate to what extent user-level source modifications affect performance at different levels of optimisation across a range of compilers. We then study new GPU-enabled version of the kernels, written for this study using the new OpenACC standard for accelerator directives, to explore current compiler capabilities for heterogeneous architectures. Finally, we attempt to optimise the performance of OpenACC using auto-tuning technology developed at the University of Edinburgh.

# 2 Introduction

Rather than performing a review of existing literature, we decided to undertake a practical investigation of compiler performance. The most important decision in any study of this type is what particular source code(s) should be used in any experiments. There is an extremely wide range of standard benchmark codes available, ranging from small serial kernels through to large parallel applications, which offers a somewhat bewildering amount of choice. Fortunately, however, one of the cornerstones of CRESTA is that six applications have been identified as the chosen co-design vehicles so this gives us a much more constrained set of codes to choose from.

After a brief survey of the contents of the CRESTA benchmark suite [2], Nek5000 [3] was chosen because it already includes a self-contained kernel benchmark that measures the performance of a variety of matrix-matrix operations that are a core part of the algorithm. These are already implemented via multiple subroutines that perform the same calculation using different pieces of code. The kernels have also already been used in an extensive auto-tuning study [4] (using the CHILL framework [5] previously described in [1]) that leaves us free to concentrate these auto-tuning studies on our new GPU-accelerated kernel.

## 2.1 Nek5000

Nek5000 is a Fortran code that performs engineering simulations using spectral elements. The core computational kernel involves a large number of matrix-matrix operations that could be implemented by a call to the standard DGEMM function from level 3 of the BLAS library. DGEMM is a standard HPC test case and it might therefore appear that there is little new to be learned here. However, the particular calculation required in Nek5000 has a number of distinguishing features:

1. It requires multiplications between a large number of independent small matrices as opposed to a small number of multiplications of large matrices. This means that memory bandwidth is stressed as well as floating-point performance, which is not true of a typical large DGEMM test case.
2. The benchmark investigates a wide range of matrix sizes (though all are small).
3. The matrices are not always square: for a given value of $N$ the benchmark has three cases with matrices of size $N$ x $N$, $N^2$ x $N$ and $N$ x $N^2$. The values of $N$ range between 1 and 24.

## 2.2 Structure of the report

In section 3 we describe the construction and verification of a small stand-alone benchmark as extracted from the main Nek5000 code, and describe the various matrix-matrix implementations already present in the code. In section 4 we study how the performance of the kernels varies across a range of compilers and optimisation levels on HECToR, the UK national supercomputer. We then port the benchmark to a platform which contains NVIDIA GPU accelerators, and develop a simple OpenACC version of the kernel which is investigated in section 5. In section 6 we describe how this was ported to C to enable it to fit into the chosen auto-tuning framework, and present results from that investigation. Section 7 presents some conclusions and describes possible further work.

In many cases we are able to compare the performance achieved from the user code to that obtained from the highly-optimised DGEMM library. This library typically provides an upper bound on the achievable performance and is therefore a useful reference value. The reason we are looking at the performance of the Nek5000 matrix-matrix kernels is not to try and out-perform DGEMM: rather, we are using them as examples of the types of kernels that scientists might want to use in a real code. Of course, for production runs of Nek5000 itself then using an optimised library would probably be the preferred option.

## 2.3 Purpose

The purposes of this deliverable are:

- To investigate the performance variation of the Nek5000 kernels occurring from differences in source code, compiler and optimisation level;
- To implement an accelerated OpenACC kernel and investigate its performance on an NVIDIA GPU;
- To attempt to optimise the accelerated kernel using an auto-tuning framework.

## 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 |
| **NAIS** | Numerical Analysis and Intelligent Software |
| **OpenACC** | Open Accelerator Directives |
| **OpenMP** | Open Multiprocessing Directives |
| **PGI** | The Portland Group, Inc. |
| **PM** | Project Manager |
| **RMS** | Root Mean Square |
| **WP** | Work Package |

# 3 Stand-alone kernel benchmark

With the Nek5000 distribution, a standard program is provided that executes some low-level benchmarks without requiring any input files. As supplied it runs both computation (matrix-matrix) and communication (ping-pong, reduction etc.) kernels. Although it would have been possible simply to comment out the communication benchmarks, this has two major disadvantages:

1. The benchmark would still have contained thousands of lines of unused code;
2. It would not have been easy to port because, as supplied, the unused code does not compile on some compilers (e.g. the Cray compiler).

The minimal amount of code was therefore extracted to enable the calculation benchmarks to be run. This only required us to retain two source files (mxm_wrapper.f and mxm_std.f) and three include files (OCPTR, SIZE and TOTAL). The Fortran source files were left untouched, and all new code (e.g. the driver program) was contained in a single file nekbench.f. Of the include files, only TOTAL had to be altered. In our benchmark it is simply a dummy file containing only comments. It is provided so that we do not have to alter any other files that explicitly include it.

## 3.1 Verification

A new version of the existing driver routine 'mxmtest' in mxm_std.f was written to provide more control. In addition, since new kernels were to be added as part of this study, a verification routine was introduced to ensure correctness. A reference solution is computed using the simplest version of the kernels (this is called 'std' and is completely straightforward naïve Fortran). All subsequent results are compared to this reference by computing the RMS difference, and if this exceeds a certain tolerance (set to 1.0e-12), then an error is reported. All calculations are done in double precision, although this is actually achieved by promoting reals to doubles using compiler flags.

This verification actually discovered an error in one of the routines. This had already been identified when the source code was read by the author while trying to understand how each kernel routine operated (there is a simple typo in the 'mxm44' implementation where the wrong specialised subroutine is called for $N$=1). It was encouraging that the verification test reported this as an error at runtime, and also that no errors were reported in any of the other routines.

## 3.2 Mode of operation

The driver routine originally had the following structure:

- Initialise data to random values
- Loop over the three test cases
    - Loop over values of $N$ from 1 to 24
        - Loop over the ten different matrx-matrix kernel implementations
            - Start the timer
            - Perform many repetitions of the kernel, attempting constant runtime by choosing the repetition count based on the value of $N$ and the number of matrices $M$ being considered (as distributed $M$ = 4*8*8*8*130 = 266240).
            - Stop the timer and report a MFLOP value

For a given value of $N$, the three test cases of $A$ x $B$ = $C$ are:

1. $N^2$ x $N$ matrix times $N$ x $N$ matrix equals $N^2$ x $N$ matrix ("tall times square")
2. $N$ x $N$ matrix times $N$ x $N$ matrix equals $N$ x $N$ matrix ("square times square")
3. $N$ x $N$ matrix times $N$ x $N^2$ matrix equals $N$ x $N^2$ matrix ("square times long")

It is important to remember that each test is actually repeated over many different matrices, i.e. the actual calculation is $A_i$ x $B_i$ = $C_i$ for $i$ = 1, 2, 3, .... If the number of repetitions exceeds $M$ then $i$ is reset to 1 and the process continues from the start.

In the original code the memory management was quite complicated with the arrays of matrices being declared as one-dimensional arrays in the main code, and the loop over *i* done manually by indexing into the appropriate section of these linear arrays when calling the kernel subroutine. This means that the loop over matrices is performed in the driver code and cannot easily be performed in the kernel routine. This would be a major problem for any accelerated kernel where this loop is an obvious candidate for parallelisation (especially for small values of *N*). This scheme was simplified, and although it had an effect on the memory access pattern no significant performance difference was observed. Indeed, the original memory access pattern did not appear to be very representative as all matrix arrays were accessed in strides of $N^3$ even when they were actually of size $N^2$. In the new code, all arrays are accessed contiguously.

The original version had one exceptional case where a matrix addition was performed as opposed to a multiplication. This was removed as it substantially complicated the logic of the code and did not provide any useful additional information for this particular study. To reduce the total amount of output, the exhaustive loop over *N* was replaced by only four values: 6, 12, 18 and 24.

Finally, the original driver contained both “fast” and “memory” versions of the benchmarks. The “memory” version is as described above, looping over the *M* matrices in each array. The “fast” version is designed to operate inside the cache and does many repetitions of *A* x *B* = *C* for the same matrices *A*, *B* and *C* (i.e. the value of *i* is fixed to 1). Although the “fast” form was retained in our standalone benchmark for completeness, it is not of so much interest as the “memory” form so we do not report results here.

The resulting code ‘nekbench.f’ compiles quickly and easily on all platforms, the only technical issues being finding appropriate compiler flags to manually enable C-style source preprocessing (since we retain the Nek5000 convention of using a “.f” suffix for source files as opposed to “.F”) and to promote default reals to double precision.

## 3.3 Kernel versions

All the kernels are implementing an extremely simple computation which, in naïve Fortran, would read:

```
double precision a(n1, n2), b(n2, n3), c(n1, n3)

do j = 1, n3
  do i = 1, n1

    c(i,j) = 0.0

    do k = 1, n
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do

  end do
end do
```

There are at least 10 different implementations of this kernel in the code, exploring the following types of optimisations:

- specific hard-coded versions for different values of n1, n2 and n3 so that these are constants at compile time;
- different loop orderings;
- loops unrolled by various amounts;
- matrix values stored explicitly in numerous temporary scalars;
- hand tiling into blocks for better cache reuse;
- calling the DGEMM library.

In order to have a manageable volume of data, we only report results for three versions:

- ‘std’ which is essentially the code as presented above (although written in an older F77 style)
- ‘m44’ which tiles the matrices to improve cache re-use use with tiles of size 4x4;
- ‘mxmd’ which used the DGEMM library function.

Although ‘m44’ did not turn out always to be the best performing kernel (particularly for small values of *N*), it is very representative of the types of optimisations which are often done by hand and, and more importantly it was the routine previously auto-tuned in [4].

# 4 CPU performance

Although it was commonplace over a decade ago to optimise code performance using manual recoding (e.g. loop unrolling or tiling), modern compilers are very sophisticated and can in principle apply many of these techniques automatically. As a result, it is interesting to investigate the performance of the Nek5000 kernels across multiple compilers at various optimisation levels. This was done on HECToR the UK national supercomputer, because it supports three compilers: GNU, PGI and Cray.

It should be noted that other compilers may produce superior performance to the results presented here. For example, the Intel compiler is specifically targeted at the x86 architectures used in all our CPU studies. In practice our choice was limited by the availability of compilers on each architecture, e.g. the Intel compiler is not supported on HECToR. However, this is not a major issue for these studies where we are more interested in the general performance variability observed across several different compilers than the maximum performance achievable across all compilers.

## 4.1 Experiments

We ran the stand-alone benchmark on a single core of HECToR. The nodes of HECToR contain 2 x 12-core 2.3 GHz AMD Interlagos processors. We deliberately took a somewhat naïve approach, compiling the code using either default levels of optimisation (i.e. no optimisation flags passed to the compiler) or a high level of optimisation (as recommended in the HECToR User Guide [6]). The only option we did not investigate was inter-procedural optimisation such as inlining since support for this varied so widely across compilers.

The results are show in Figure 1 – Figure 6 where, separately for each optimisation level, we show the six performance results obtained for the two kernels ‘std’ and ‘m44’ on each of the three compilers. We show results for all three cases for the smallest and largest matrix sizes *N*=6 and *N*=24 (note the difference in vertical scale between the two sizes). For comparison we also show the DGEMM performance which, as expected, was unaffected by choice of compiler or options since it is an external library.

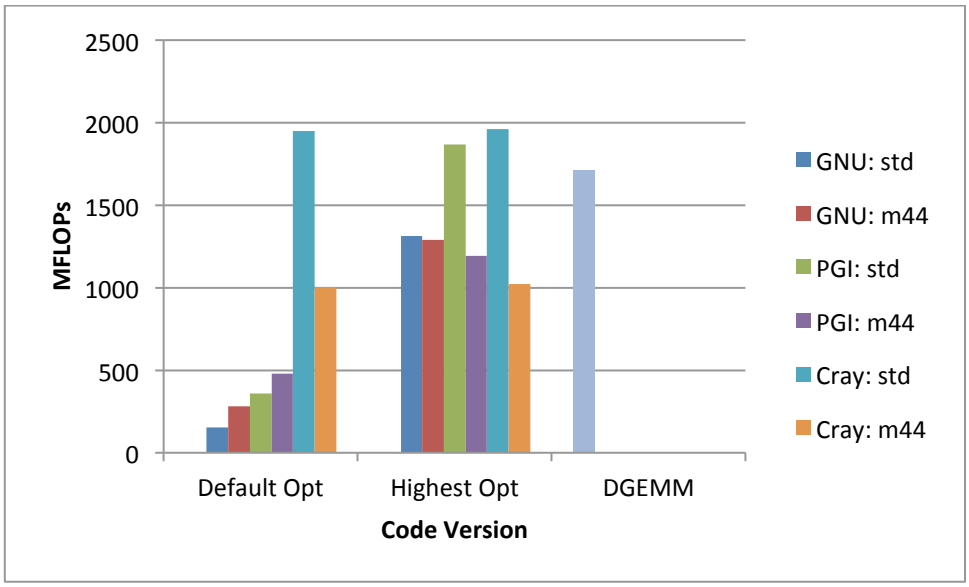


**Figure 1: Results for N=6 case 1**

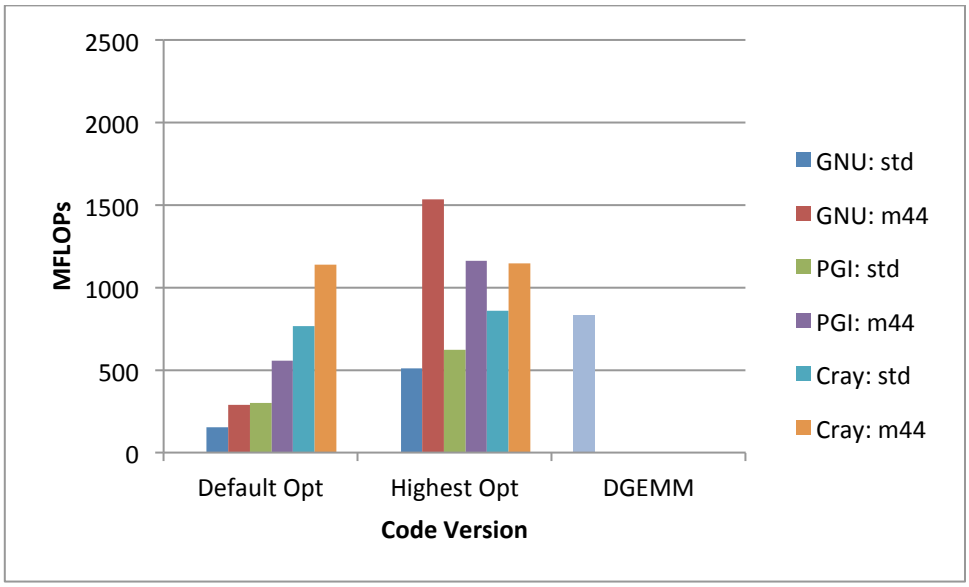


**Figure 2: Results for N=6 case 2**

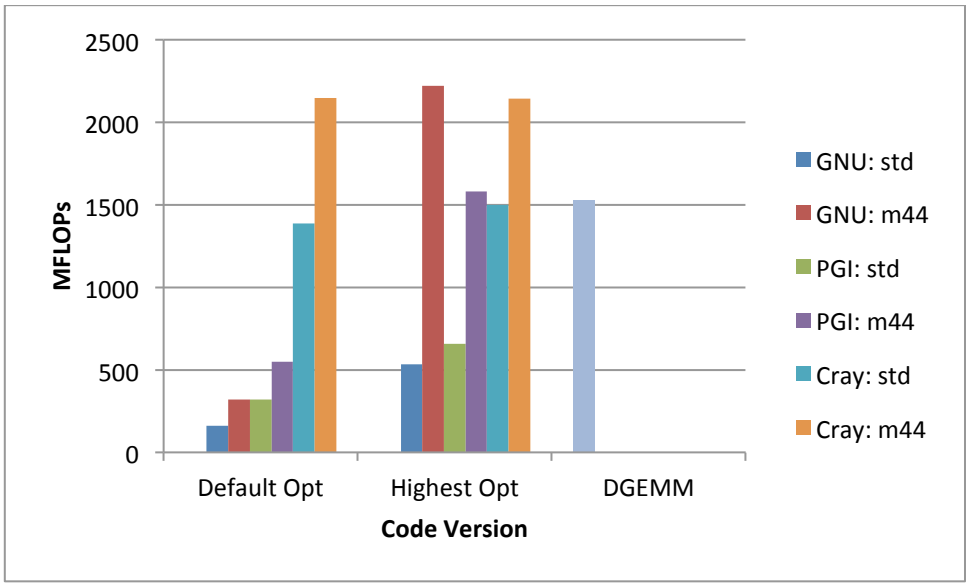


**Figure 3: Results for N=6 case 3**

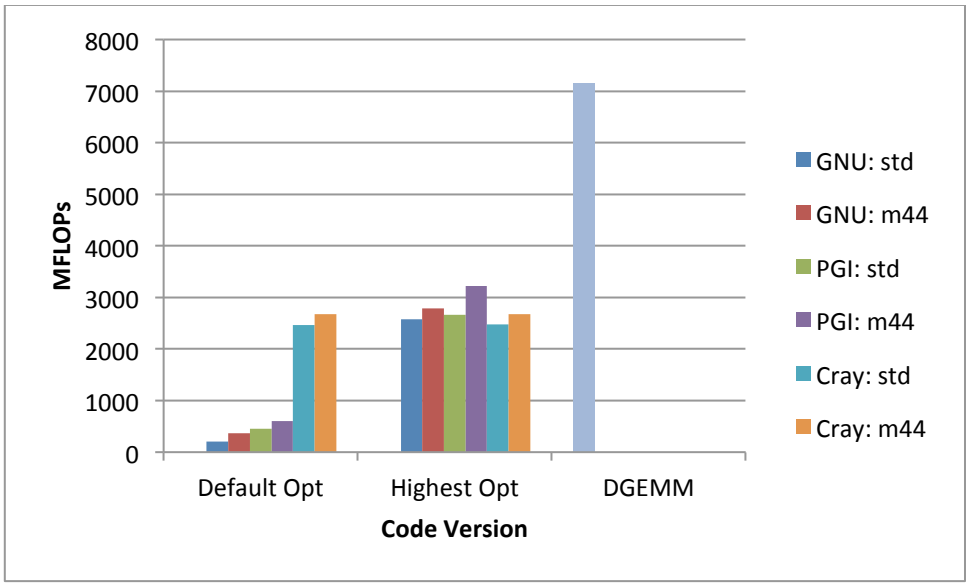


**Figure 4: Results for N=24 case 1**

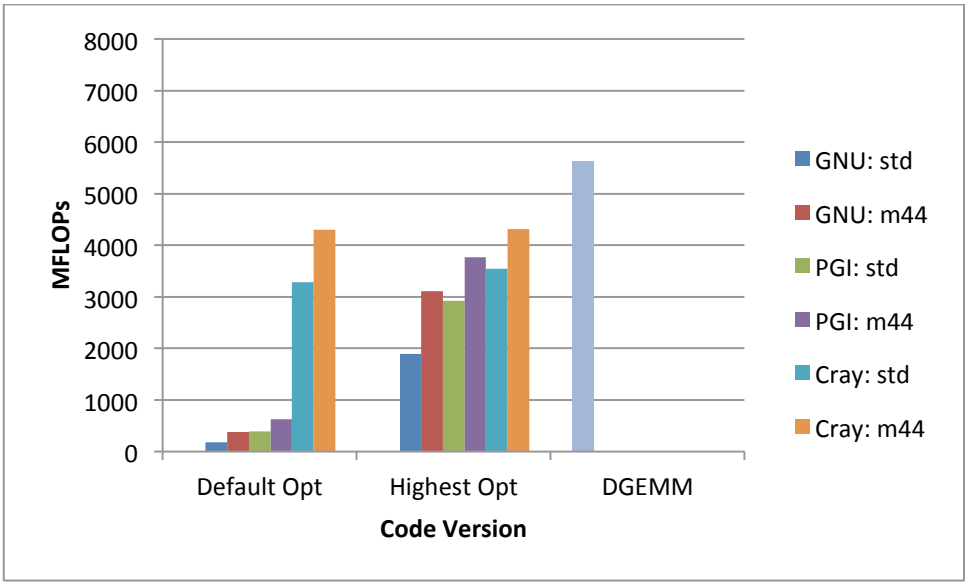


**Figure 5: Results for N=24 case 2**

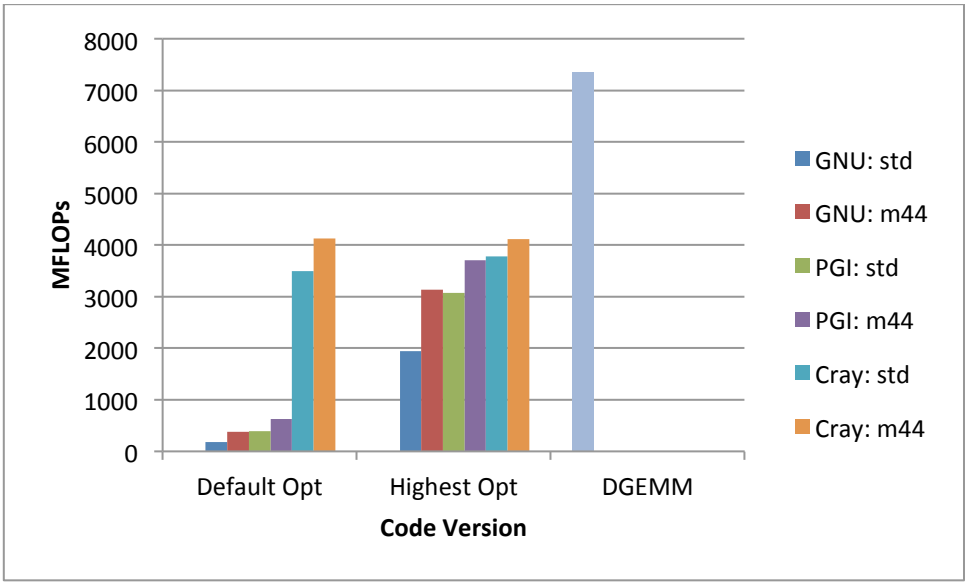


**Figure 6: Results for N=24 case 3**

## 4.2 Results

The results are dramatically different between the two matrix sizes so we will consider them separately.

### N = 6

The first thing to note is that the DGEMM results are not the fastest. Presumably this is because the library has not been targeted at matrices that are so small and/or of such unusual shapes. In all cases the Cray results are almost identical at the two optimisation levels, which simply indicates that the Cray compiler has high optimisation as a default setting (since it is a dedicated HPC compiler).

Cases 2 and 3 are very similar: m44 is always substantially faster than std for all compilers and optimisation levels. At high optimisation there is a substantial difference between the two kernels with GNU std being the slowest across all six versioos, but GNU m44 the fastest.

For case 1, m44 outperforms std at the low optimisation levels of the GNU and PGI compilers. However, for high optimisation the situation is the reverse of the other two cases with std always outperforming m44, and with Cray giving both the slowest (m44) and the fastest (std) results across all six versions.

### N = 24

The results for the larger matrices are much more consistent than for the smaller ones. Here, DGEMM is always the fastest overall and m44 always outperforms std for a given compiler and optimisation level. For case 1, PGI m44 is the fastest whereas it is Cray m44 for cases 2 and 3. However, there is a lot less variation across all the highly optimised results than for the smaller *N* = 6 case.

## 4.3 Summary

We see significant variations in performance across different combinations of compiler, optimisation level, matrix size and case. In some situations hand optimisation is beneficial, whereas in others it actually gives worse performance than naïve code. No single kernel or library routine performs the best across all the different matrices, indicating that some sort of auto-tuning might be beneficial in selecting the best choice of code for each specific problem. This is backed up by the results of [4] where auto-tuning is found to increase the performance of m44 by more than a factor of two (albeit on a different platform).

# 5 OpenACC Acceleration

Accelerators are seen by many people as the only way to increase CPU performance to the levels required for exascale. The current most popular accelerator technology is the GPU, developed for and funded by the massive worldwide computer games industry. The market leader at present is NVIDIA with their current generation Tesla GPUs installed in a number of petascale HPC systems. Motivated by this we choose to investigate the performance of the Nek5000 kernels on the Tesla architecture.

One of the major drawbacks of programming NVIDIA GPUs has been that the programming model, CUDA [7], is not a recognised standard and is completely proprietary to NVIDIA. Although there has been a large volume of research into porting HPC kernels to GPUs, the HPC community is nervous about investing substantial software development effort in converting applications to use a programming language that is not portable between different architectures. There is an ongoing effort to include accelerators in the existing OpenMP standard for shared-memory directives, but this is likely to be a long process as it requires significant updates to the standard. To address this, a number of HPC hardware and software vendors got together to produce an interim standard for accelerator directives, OpenACC [8], based on their own experiences and guided by the direction of the OpenMP efforts.

## 5.1 Porting to the NVIDIA Tesla GPU

The HECToR system used previously was useful for CPU studies as it supported three different compilers. Unfortunately, HECToR does not have GPU accelerators so we had to port the benchmark to a different system. We chose a small internal EPCC development system, Hydra, since it is easily accessible and supports OpenACC via the PGI compiler. Hydra comprises a 24-core shared-memory frontend (4 x 6-core Xeon X5650 2.67 GHz CPUs) with an NVIDIA Tesla C2050 GPU accelerator. All studies were performed with PGI version 12.3 (the version is significant as OpenACC is still a relatively new standard and compiler capabilities are constantly improving).

The basic idea of OpenACC is that the applications developer does not have to be concerned with the details of the underlying hardware. At the most basic level the programmer has to manage data transfer between host and GPU (analogous to declaring data as shared or private in OpenMP), and indicate which loops should be parallelised. Fine control can in principle be exercised over the distribution of loops to threads; OpenACC has three levels of control over *gangs*, *workers* and *vectors*. These correspond roughly to the CUDA concepts of thread blocks, thread warps and individual threads. Although the way in which loop iterations are split among each of these levels must be specified explicitly in CUDA, in OpenACC the programmer can choose to let the compiler make the decision. Here we deliberately take a very naïve approach and give complete freedom to the compiler. We will investigate how good the compiler's default choices are in the next section were we will use auto-tuning technology to search for optimal settings of these parameters.

One of the main restrictions in OpenACC is that you cannot call functions or subroutines from within an accelerated region. In order to give the compiler as much freedom as possible to parallelise over all loops, we created a new accelerated kernel which itself performs the entire loop over the *M* matrices (previously we made *M* separate calls to each kernel). This is essential in exploiting the power of the GPU because, for example, multiplying two 6x6 matrices does not require enough operations to occupy the many hundreds of GPU threads.

The code is shown below. Note that this is not exactly the code as contained in the benchmark. The real kernel is slightly more complicated in that it performs an additional loop *outside* the loop over the *M* matrices (i.e. outside the 'imat' loop) to ensure that the correct number of repetitions is performed so the elapsed time is kept under control. However, this is inside the data transfer region (the 'acc data' directive) so does not affect the logic of the loop and is omitted here for simplicity.

```
!  Simple parallel OpenACC version of the Nek5000 kernel

      double precision a(n1, n2, m), b(n2, n3, m), c(n1, n3, m)
      double precision tmp

!$acc data copyin(a,b) copyout(c)

!$acc kernels loop independent
        do imat = 1, m
!$acc loop independent
           do j = 1, n3
!$acc loop independent
              do i = 1, n1

                 tmp = 0.0
!$acc loop
                 do k = 1, n2
                    tmp = tmp + a(i,k,imat)*b(k,j,imat)
                 end do

                 c(i, j, imat) = tmp

              end do
           end do
        end do
!$acc end kernels
      end do

!$acc end data
```

The outermost ‘data’ region ensures that the input matrices *A* and *B* are copied to the GPU, and that the result *C* is copied back to the host. The outermost loop over ‘imat’ is marked as being the place to start parallelisation using the ‘kernels’ directive. All of the four loops are marked as parallelisable. The compiler is given additional information that the computation of every element of *C* is independent using the ‘independent’ clause on the first three loops. Parallelisation of the innermost loop requires a reduction operation. Although this can in principle be indicated using a ‘reduction’ clause, reductions over array elements is not possible with the 12.3 version of the PGI compiler used here. Following the advice produced by the compiler itself, an explicit temporary reduction scalar ‘tmp’ is introduced which solves the problem.

## 5.2 Performance

The performance of the three CPU kernel versions considered previously (‘std’, ‘m44’ and ‘DGEMM’) was compared to OpenACC across the same four values of *N*. The results are presented in Figure 7 - Figure 10 showing all three cases for each kernel version. Only the PGI compiler was considered as we required OpenACC support.

The first observation is that the DGEMM performance is typically not any better than the compiled source. This simply indicates that the default BLAS library on Hydra is not very well optimised. Although we could have installed a better version this was not done as we are more concerned here with compiler performance.

The performance of the simple OpenACC kernel is quite impressive. For the smallest value of *N* it depends quite strongly on the shapes of the matrices: 18 GFLOPs for case 1 compared to 8 GFLOPs for cases 2 and 3. As *N* increases, the performance of cases 2 and 3 increase more rapidly than case 1. At *N* = 12, the figures are 27, 19 and 22 GFLOPs respectively. For larger values of *N* we see case 3 becoming the fastest, and cases 1 and 2 having similar (but poorer) performance to each other. For *N* = 18 we see 19 GFLOPs for cases 1 and 2, and 23 GFLOPs for case 3. Performance increases again for *N* = 24 where these figures are now 25 and 30 GFLOPs.

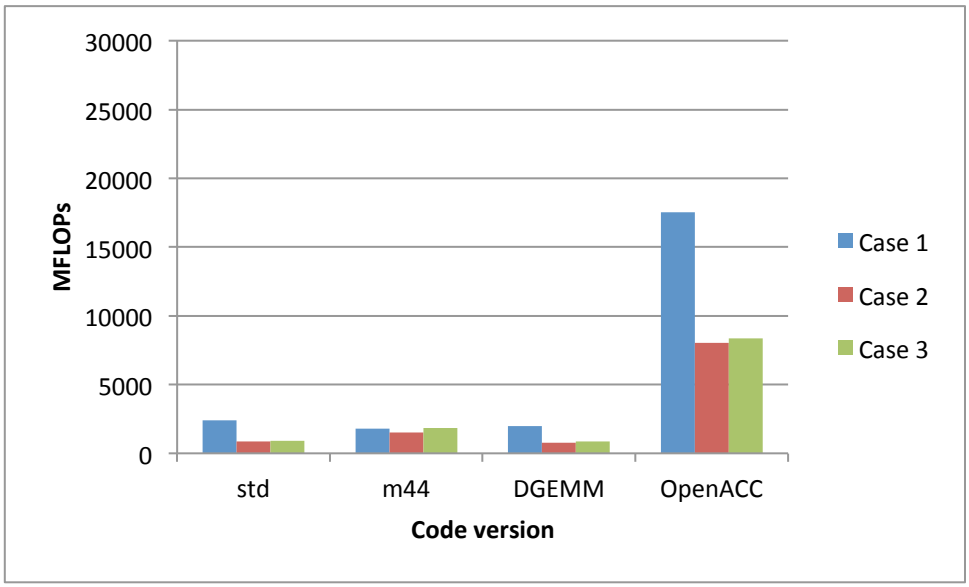


**Figure 7: Results for N = 6 with the PGI compiler**

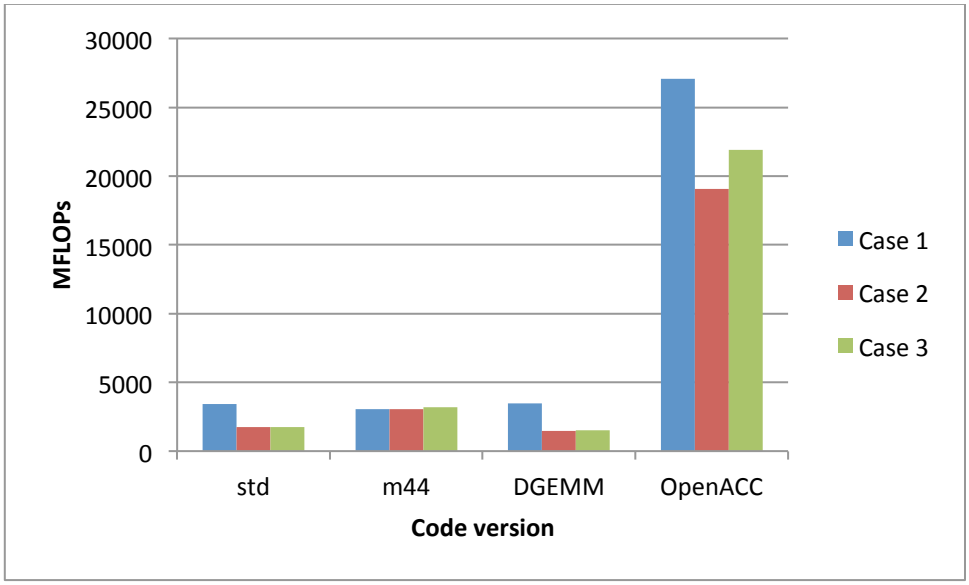


**Figure 8: Results for N =12 with the PGI compiler**

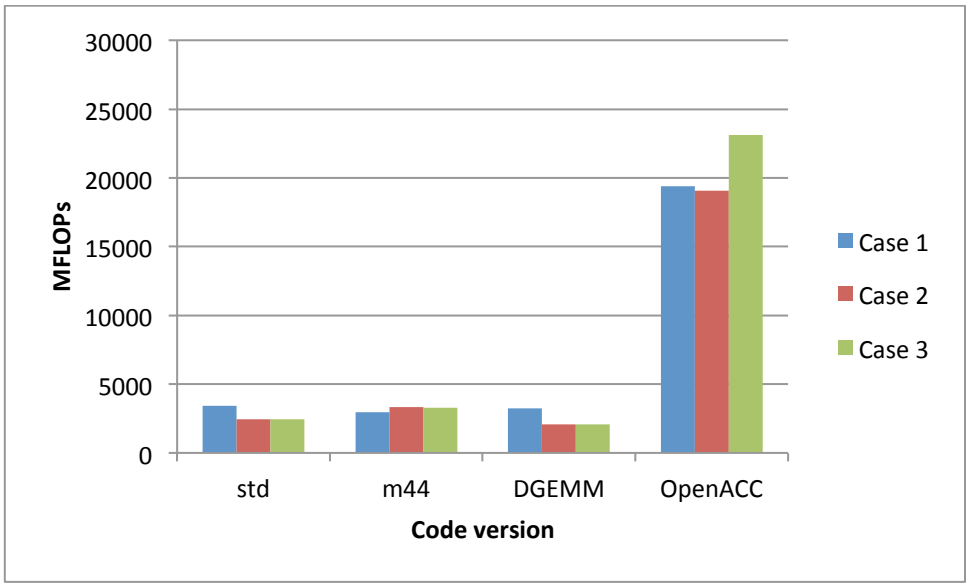


**Figure 9: Results for N = 18 with the PGI compiler**

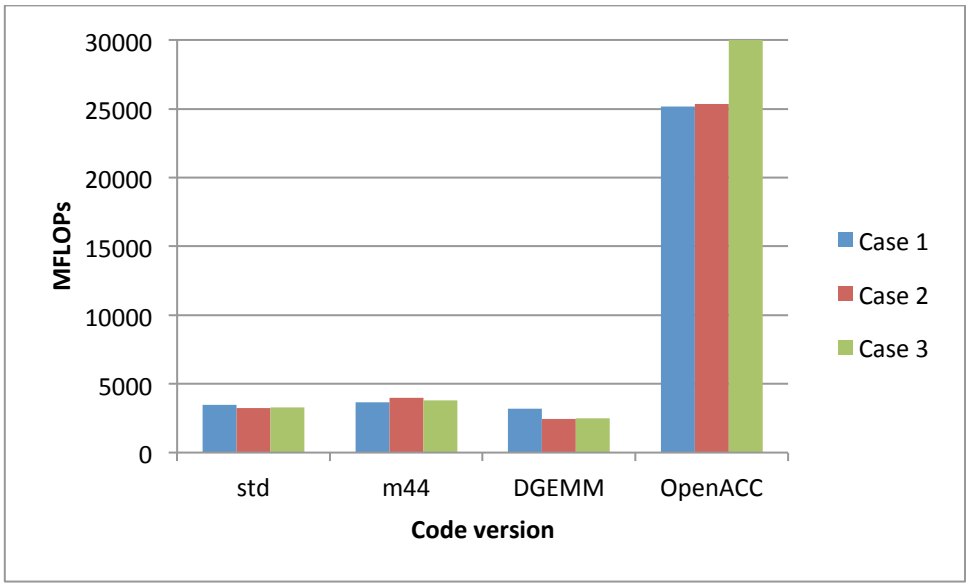


**Figure 10: Results for N = 24 with the PGI compiler**

## 5.3 Summary

Overall we see performance improvements for the OpenACC version of between 5 and 9 when compared to 'm44' on the CPU. Although this may seem impressive for such a simple parallelisation, it is important to note that we are comparing to the performance of a single Xeon core. Even a very simple OpenMP parallelisation of the CPU calculation (e.g. across the outermost loop over the *M* matrices) would easily exploit all 24 Xeon cores and we would expect that this would outperform the OpenACC version. Similarly, a hand-coded CUDA version for the GPU would probably outperform our OpenACC version, or we could aim for ultimate performance by calling a GPU-enabled version of DGEMM from the CUBLAS library.

Despite this, the results are promising in that the insertion of five simple compiler directives (one for data movement, and one for each of the four loops) enabled the compiler to generate GPU code achieving some tens of GFLOPs for all but the smallest matrices. Other than the introduction of a single scalar variable, no source changes were required. Comparing to similar naive Fortran compiled using the CPU compiler (i.e. the 'std' kernel), performance improvements on the GPU vary between factors of 6 and 12 across the various tests. All the OpenACC versions also produced correct answers.

# 6 Auto Tuning

Lead by Prof. Mike O'Boyle, the compiler group from the School of Informatics at the University of Edinburgh [9] performs leading research into compiler auto-tuning for both serial and parallel programs. Under the NAIS project [9], EPCC collaborates with this group to investigate these novel compilation techniques as applied to real scientific applications. Nick Johnson at EPCC has recently started a project with Dominik Grewe and Alberto Magni to look at OpenACC codes. This presents CRESTA with an important opportunity to evaluate auto-tuning technologies for the co-design vehicles: all that is required is a simple OpenACC version of the Nek5000 kernel.

Unfortunately, the existing kernel is written in Fortran. Although there is no reason in principle why it could not be auto-tuned, the OpenACC auto-tuning framework is still in the research phase and has only been verified on C codes. As a result we had to port the Nek5000 kernel benchmark to C.

## 6.1 Nek5000 OpenACC kernel in C

The C code is shown below. It is a direct translation of the Fortran version except with the order of the array indices reversed to preserve the same layout in memory. The only other difference is that the array sizes are explicitly declared in the data copy clauses. In the real version of the code this kernel is contained in a function and, despite the arrays being declared statically with compile-time constant sizes, the compiler complains that it does not know the extents of the *A*, *B* and *C* arrays in the data directive unless they are explicitly restated. Presumably this is because the C language does not support arrays very well, and they are perhaps converted to simple pointers during the function call at which point the true array extents are forgotten. As before, results are verified against an equivalent kernel executed on the host CPU.

```
/* Simple parallel OpenACC version of the Nek5000 kernel */


double a[NMAT][N2][N1], b[NMAT][N3][N2], c[NMAT][N3][N1];
double tmp;

#pragma acc data \
copyin (a[0:NMAT][0:N2][0:N1], b[0:NMAT][0:N3][0:N2]) \
copyout(c[0:NMAT][0:N3][0:N1])

#pragma acc kernels loop independent
for (imat=0; imat < NMAT; imat++)
{
  #pragma acc loop independent
  for (j = 0; j < N3; j++)
  {
    #pragma acc loop independent
    for (i = 0; i < N1; i++)
    {
      tmp = 0.0;

      #pragma acc loop
      for (k = 0; k < N2; k++)
      {
        tmp += a[imat][k][i] * b[imat][j][k];
      }

      c[imat][j][i] = tmp;
    }
  }
}
```

## 6.2 Default performance

Before the auto-tuning was attempted, it was important to verify that this code executed correctly with default settings and performed similarly to the Fortran version. All the test cases verified correctly against the host code. The performance of this new C version was compared to the previous Fortran results – see Figure 11.

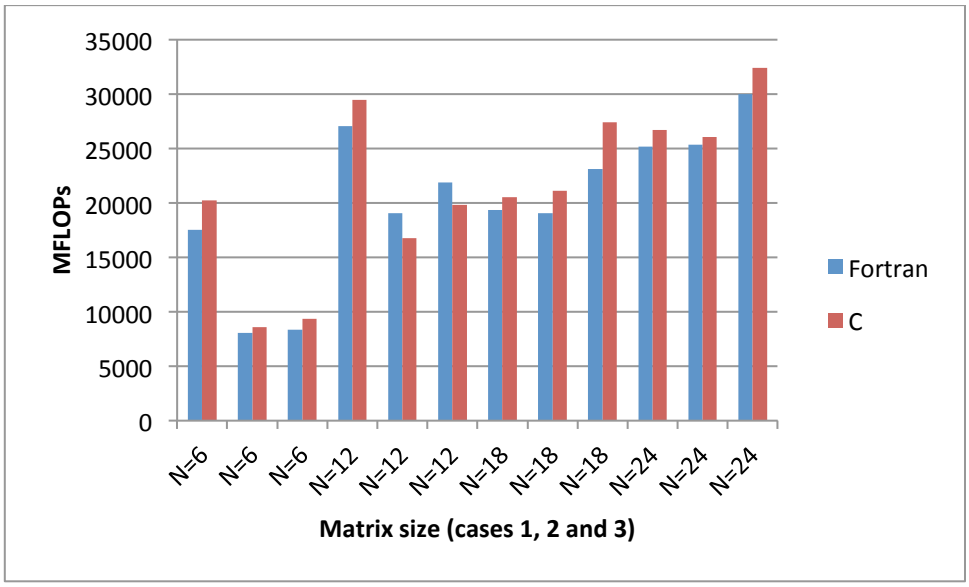


**Figure 11: Comparison of OpenACC kernel for different languages**

The agreement between the two codes, although not perfect, is very good. It is clear that the two versions have roughly the same absolute performance and also exhibit the same general trends across different matrix sizes and shapes. Except for cases 1 and 2 at $N$ = 12, the C version is always slightly faster than Fortran. Perhaps it is because the array dimensions in the C kernel are fixed at compile time as opposed to being passed as runtime arguments in Fortran, but this is pure speculation.

## 6.3 Auto-tuned performance

The auto-tuning framework compiles and executes a code many thousands of times. It is therefore essential to make it as simple as possible to compile and execute, and to minimise screen output. The C version of the OpenACC kernel driver was designed so that it only ran a single test case, with the parameters selected at compile time by passing different flags to 'make'. This is important as we want to find optimal OpenACC parameters for each kernel, so we must compile each combination of matrix size and shape separately. Output was reduced to a single performance figure, and verification indicated by a true or false return value from main. The benchmark was then handed on as a black box to Mike O'Boyle's group who ran it through their auto-tuning tools.

The auto-tuning was performed on the outermost three of the four loops. It was found that the compiler was not parallelising the innermost loop so this was left untouched. This is not an issue in practice as this loop involves a reduction operation over a very small loop with a maximum trip count of 24 (this loop is always over $N$ and never $N^2$). It would therefore be unlikely that parallelisation would have any overall benefit given that there is ample scope for parallelism at the three outer loop levels.

The three parallel loops were auto-tuned over various choices of the 'gang' and 'vector' parameters (the highest and lowest levels of control respectively). Although there is an additional 'workers' parameter that in principle controls parallelism between these two levels, it is somewhat redundant on the NVIDIA architecture. For values of 'vector' larger than the warp size (which is 32), a loop is automatically split into multiple 'workers' so there is little benefit in varying this parameter explicitly.

The results are shown in Figure 12 (relative improvement) and Figure 13 (absolute performance).

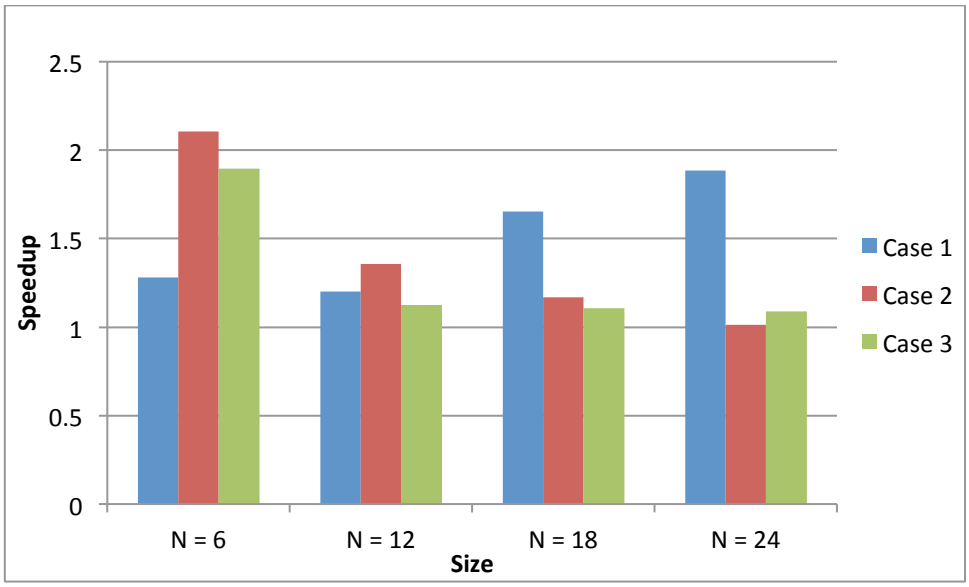


**Figure 12: Speedup from Auto-tuning**

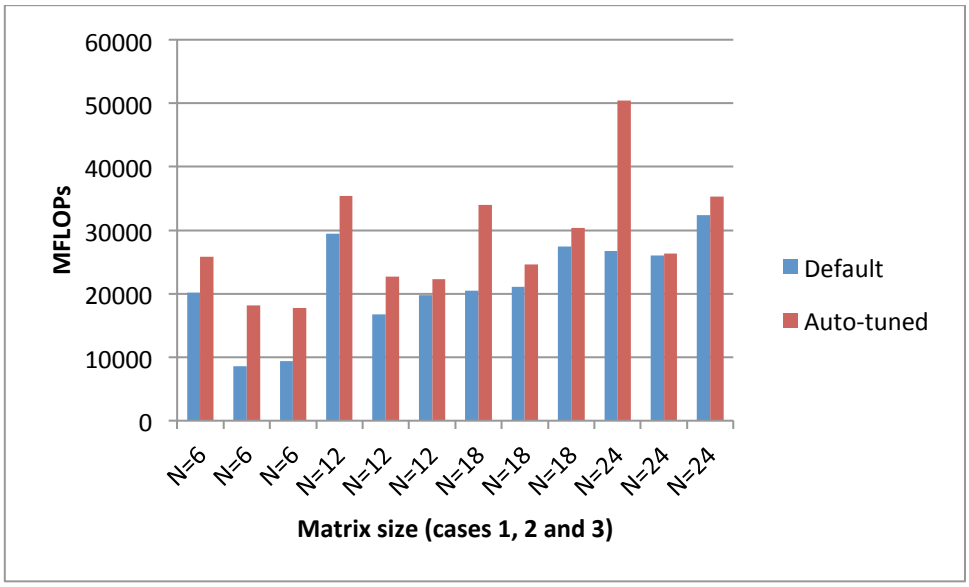


**Figure 13: Performance of Default and Auto-tuned Kernels**

## 6.4 Summary

The benefits of auto-tuning for this code are obvious, with the performance of every kernel being improved to some extent. The amount of improvement varies significantly across the different tests, between factors of 1.01 to 2.10. The similarities in default performance between the original Fortran and new C kernel would suggest that the optimised parameters from the C kernel could also be used in the Fortran version. However, this has not yet been investigated.

As OpenACC is such a new technology the auto-tuning done here is quite basic, looking only at the space of high-level loop scheduling options with the actual source code remaining fixed. Many more GPU optimisations can be investigated for codes written at a lower level using CUDA. For example, in [11] the full CHILL framework is extended to include CUDA and the auto-tuned performance of matrix-matrix code approaches that of the CUBLAS library. A similar study (although not performed using CHILL) was carried out in [12] for 3D Fast Fourier Transforms, where the auto-tuned code out-performed the vendor-supplied CUFFT library. However, because OpenACC is much more portable and easier to use than CUDA, we believe that auto-tuning the higher-level OpenACC parameters will become increasingly important in the future as HPC applications developers move away from CUDA.

# 7 Conclusions and Further Work

This study deliberately took a very naïve approach using simple kernels to investigate the accelerator capabilities of today's compilers and to see how much performance improvement could be gained from compiler auto-tuning. The results were overall very positive, with significant speedups achieved on an NVIDA GPU compared to the single CPU versions of the Nek5000 benchmarks. This only required the addition of a few simple OpenACC directives (assuming OpenACC support in the compiler). These speedups could be improved further by up to a factor of 2 by auto-tuning the loop distribution parameters. However, the performance achieved was still well below the peak of the GPU, and probably even inferior to a simple multicore CPU version.

As this is an initial study it is clearly incomplete at present, and there are many possibilities for further work.

- Profiling of the OpenACC code to identify further optimisation opportunities.
- Comparison to performance of an OpenMP multicore version.
- Comparison to CUDA or library (e.g. CUBLAS) implementations.
- Evaluation of Fortran OpenACC performance using optimal C parameters.
- Development of a single kernel routine that calls the optimal OpenACC version based on its input parameters.
- Integration back into Nek5000.

Perhaps the major success of this work has been that the initial results have been sufficiently promising that we will be creating a CRESTA co-design team to further investigate GPU acceleration of the entire Nek5000 parallel application.

# 8 Acknowledgements

We would like to thank Mike O'Boyle for his assistance in this work, and Dominik Grewe and Alberto Magni for undertaking the auto-tuning investigations. Thanks go to Nick Johnson for acting as the interface between CRESTA and the auto-tuning group and for his advice on OpenACC and the design of the C kernel.

# 9 References


[1] State of the art and gap analysis, Project Deliverable D3.1.

[2] CRESTA benchmark suite, Project Deliverable D2.6.1.

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

[4] J. Shin, *et al*., "Autotuning and Specialization: Speeding up Nek5000 with Compiler Technology", presented at the International Conference on Supercomputing, 2010.

[5] A. Tiwari *et al*., "A scalable autotuning framework for compiler optimization", Proceedings of the 24th International Parallel and Distributed Proecessing Symposium (2009).

[6] HECToR user guide http://www.hector.ac.uk/support/documentation/userguide/, accessed on June 6th 2012.

[7] The CUDA Toolkit http://www.nvidia.com/content/cuda/cuda-toolkit.html/.

[8] OpenACC Home Page http://openacc.org/.

[9] Institute for Computing Systems Architecture, School of Informatics, The University of Edinburgh http://www.inf.ed.ac.uk/research/icsa/.

[10] NAIS: The Centre for Numerical Algorithms and Intelligent Software http://www.nais.org.uk/.

[11] G. Rudy, *et al*., "A programming language interface to describe transformations and code generation" in Proceedings of the 23rd international conference on Languages and compilers for parallel computing (LCPC'10).

[12] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs" in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09).