6th European Conference on Computational Mechanics (ECCM 6)
7th European Conference on Computational Fluid Dynamics (ECFD 7)
11–15 June 2018, Glasgow, UK

ENERGY CONSUMPTION OF ALGORITHMS FOR SOLVING THE COMPRESSIBLE NAVIER-STOKES EQUATIONS ON CPU'S, GPU'S AND KNL'S

Satya P. Jammy[1], Christian T. Jacobs[1], David J. Lusher[1] and Neil D. Sandham[1]

[1] University of Southampton, SO17 1BJ, United Kingdom. S.P.Jammy@soton.ac.uk

Key words: Energy efficiency, Finite difference methods, Graphics Processing Unit (GPU), Knights Landing (KNL)

Abstract. In addition to the hardware wall-time restrictions commonly seen in high-performance computing systems, it is likely that future systems will also be constrained by energy budgets. In the present work, finite difference algorithms of varying computational and memory intensity are evaluated with respect to both energy efficiency and runtime on an Intel Ivy Bridge CPU node, an Intel Xeon Phi Knights Landing processor, and an NVIDIA Tesla K40c GPU. The conventional way of storing the discretised derivatives to global arrays for solution advancement is found to be inefficient in terms of energy consumption and runtime. In contrast, a class of algorithms in which the discretised derivatives are evaluated on-the-fly or stored as thread-/process-local variables (yielding high compute intensity) is optimal both with respect to energy consumption and runtime. On all three hardware architectures considered, a speed-up of ~ 2 and an energy saving of ~ 2 are observed for the high compute intensive algorithms compared to the memory intensive algorithm. The energy consumption is found to be proportional to runtime, irrespective of the power consumed and the GPU has an energy saving of ~ 5 compared to the same algorithm on a CPU node.

1 Introduction

As the performance of high-performance computing (HPC) systems tends towards exascale, the characteristics of hardware are not shaped by floating-point operation (FLOP) count and memory bandwidth alone. Energy costs continue to rise in an era where climate change is a factor in many decision making processes, and power consumption and energy efficiency are therefore prominent concerns of hardware manufacturers and end-user software developers alike. According to a report by the US Department of Energy [1], power consumption of a potential exascale machine is predicted to increase by a factor of three relative to HPC system designs in 2010. Indeed, an exascale-capable machine built with the technology existing in 2010 would be expected to cost more than \$2.5 billion USD in power expenditure alone [1] and consume at least a gigawatt of power [10].

From the perspective of numerical simulations, it is quite probable that in addition to the hardware wall-time restrictions commonly seen in institutional, national and international HPC systems, simulations will also be constrained by energy budgets. It is therefore paramount that numerical codes and their underlying solution algorithms can be optimised in both respects. Recent effort has focussed on designing efficient solvers with the help of tiling, and improving data locality, etc (e.g. [5; 23]). However, it is possible to introduce more radical algorithmic changes to the codebase. Recent work [12] using the automatic source code generation capabilities of the OpenSBLI framework [10] has demonstrated that varying the computational and memory intensity of explicit finite difference algorithms for solving the compressible Navier-Stokes equations in three dimensions on a representative test case, has a significant impact on simulation runtime on an Intel Ivy Bridge CPU node. However, not much is known about how these algorithms behave across various architectures with respect to their energy efficiency and power consumption.

This paper investigates the performance, in terms of both runtime and energy efficiency, of six algorithms (five of which are from [8]) that solve the compressible Navier-Stokes equations on an Intel Ivy Bridge CPU node, an NVIDIA K40c GPU, and an Intel Xeon Phi Knights Landing processor (denoted KNL for brevity). The Methodology section introduces the numerical method and the algorithms under consideration, along with the automatic source code generation framework (OpenSBLI) used to generate C code implementations for each algorithm. The Hardware and Execution section describes the various backend programming models and the hardware used, along with a brief discussion of how the power consumption and energy usage were measured. The Results section demonstrates that the algorithms requiring minimal memory access are consistently the best-performing algorithms across the three different architectures. The paper closes with some concluding remarks in the Summary and Conclusions section.

2 Methodology

The governing equations are the non-dimensional compressible Navier-Stokes equations with constant viscosity. The convective terms are written in the skew-symmetric formulation of [4] and the Laplacian form of the viscous terms is used to improve the stability of the equations [14]. The spatial derivatives are evaluated using a fourth-order central finite-difference method and a three-stage low storage Runge-Kutta method is used to advance the solution variables in time.

Pseudo-code for the spatial and temporal discretisation of the governing equations is outlined in Figure 1. Most of the computational time is spent evaluating the residual. This consists of (a) evaluating the primitive variables (u_i, P, T), (b) evaluating the spatial derivatives (63 derivatives are to be computed), and (c) to compute the residual of the equations. The algorithms considered in this work differ in the way the spatial derivatives are evaluated and the residuals are computed. The other steps are evaluated the same way across all the algorithms, as in [8]. Note that all the steps in Figure 1 are applied on the entire domain (grid size) except the application of boundary conditions, which are applied over each boundary face.

```
initialise-the-solution-variables
for each-timestep do
  save-state ($\vec{Q}^n$)
  for each-runge-kutta-substep do
    evaluate $R(\vec{Q}^k)$
    evaluate $\vec{Q}^{k+1}$
    n+1-level-solution-advancement $\vec{Q}^{n+1}$
    apply-boundary
  end for // runge-kutta-substep
end for // timestep
```

Figure 1: Generic algorithm for the numerical simulation of the compressible Navier-Stokes equations using a multi-stage Runge-Kutta timestepping scheme.

The different algorithms are outlined below. The source code for the implementation of each algorithm is generated automatically using the OpenSBLI framework [6]. Three categories of algorithms are considered in the present work, (a) conventional algorithms, (b) intermediate storage algorithms and (c) minimal-storage algorithms. The `conventional algorithm` follows the approach frequently used in large-scale finite difference codes [14] and is considered to be the baseline (BL) algorithm.

The `intermediate storage algorithms` stores only the velocity gradients into work arrays and the rest are recomputed. Two variants of this type are considered depending on the way the derivatives that are not stored are computed. In Store Some (SS) algorithm, the derivatives are evaluated to thread/process-local variables. Where as in the Recompute Some (RS) algorithm these are directly substituted in the residual of the equations. The implmenetation details are described in [8].

The `minimal storage algorithms` do not store any partial derivatives to work arrays. They are either recomputed on-the-fly, or evaluated as thread-/process-local variables. Three different variants are considered, Recompute All (RA), Store None (SN) and Store None version 2 (SN2). The RA algorithm follows a similar approach to that of the RS algorithm, where as th SN and SN2 algorithms follow the approach similar to that of SS algorithm for all the partial derivatives in the equations as described in [8]. The SN2 algorithm is a not reported in [8], in this data locality is improved by evaluating the derivatives in groups. First, the groups that contain velocity derivatives (u_i) are evaluated in the order of the index i, and then the group of non-velocity derivatives is evaluated.

A summary of the algorithms is presented in Table 1 with the operation count per timestep. This is obtained by counting the number of operations for each expression in the algorithm using the `count_ops` functionality in SymPy and adding them together. This sum is then multiplied by the total number of grid points (256^3 used in this work) and the number of sub-stages in a Runge-Kutta iteration (3 in the current work) to arrive at the operation count per timestep.

Operation count is an indication of the compute intensity of the algorithms. It should also be noted that, as different compilers optimise differently, the actual operation count

	BL	RS	SS	RA	SN	SN2
Local variables	0	0	53	0	63	63
Extra arrays	63	9	9	0	0	0
Operations ($\times 10^9$)	48	68	54	145	69	69

Table 1: Number of process-/thread-local variables, work arrays and operation count used by each algorithm, for a three stage Runge-kutta time step.

performed by the resulting binary executable would vary. Expensive divisions are avoided by evaluating rational numbers and the inverse of constants at the start of the program. A complete list of the optimisations performed is reported in [8].

3 Hardware and execution

OpenSBLI uses OPS's source-to-source translation capabilities to tailor and optimise the algorithm for their execution on various architectures.

On the CPUs and KNLs a variety of programming models such as MPI, MPI+OpenMP, OpenACC, inlined MPI, and Tiled may be used. As it is beyond the scope of this paper to compare different programming models, we select the MPI model on the CPUs and KNL. This yields a typical manually-parallelised solver with a similar parallel performance as reported in [11]. The CPU and KNL simulations are performed on the UK National Supercomputing Service (ARCHER). The CPU system comprises Intel Ivy Bridge processors, and the KNL system comprises Intel Xeon Phi Knights Landing 7210 processors.

Similarly, for GPUs one can use CUDA, OpenCL, OpenACC or MPI_CUDA for multiple GPUs. Our experience [9] is that CUDA performs as well as OpenCL, so the CUDA backend is used for the execution of algorithms on GPUs. The simulations were performed on a single NVIDIA Tesla K40c installed on a local desktop computer. This comprises 2,880 CUDA stream cores, and runs at a clock speed of 745 MHz with 12 GB of on-board memory [12].

3.1 Performance measurement

For measuring the performance and energy consumption by the algorithms on various architectures, the performance measurement libraries for those architectures were used. On the CPU and KNL, the PAT MPI library (v 2.0.0) [3] was used. PAT MPI library uses the Cray PAT Application Programming Interface (API), giving easy control and requiring minimal modifications to the source code to collect the performance parameters. It should be noted that the actual performance measurement is done by the Cray PAT library and PAT MPI library calls ensure that the number of MPI processes reading the hardware performance counters is minimised, and the master (rank 0) collects and outputs the performance data [3].

Using the PAT MPI library involves adding calls to the `pat_mpi_monitor` in the source code at the point where performance measurements should be taken and the quantities that are to be monitored are controlled by a Makefile. More details can be found in [2; 3].

In the present work, for all the algorithms on CPU and KNL; the simulation runtime, instantaneous power (W) and cumulative energy (J) were recorded at the start and end of each timestep using the `pat_mpi_monitor` function [3].

On the GPU the NVIDIA Management Library (NVML) [13] was used. The function `nvmlDeviceGetPowerUsage` was inserted into the code at the start and end of each iteration to measure the power consumption.

3.2 Compilation

After the insertion of performance measurement calls, the code was tailored to the target hardware backends using OPS's source-source translation capabilities. For the CPU and KNL architectures the Intel C/C++ compiler (v 17.0) was used, and the NVIDIA C compiler (v 8.0.44) was used for the GPU. On all architectures we used the -O3 compiler optimisation flag combined with architecture-specific flags (-xHost, -xMIC-AVX512 and sm_35 compute architecture for CPU, KNL and GPU, respectively) to generate binary code specific to each architecture. All the dependant libraries and the algorithms are compiled using the same optimisation flags on each architecture.

4 Results

The simulation was set-up using 256 grid points in each direction. The BL, RS, SS, RA and SN algorithms were validated by [6; 8] using a Cray compiler on the ARCHER CPU node. The results of the algorithms using the current compilers on different architectures are compared to the solution dataset of [7]. The minimum and maximum error in the solution variables between [7] and the current runs was found to be of the order of machine precision on all three architectures.

To evaluate the energy efficiency of the algorithms, the simulations are run for 500 iterations with energy measurement enabled. Each algorithm is repeated five times on each architecture and the performance results presented here are averaged over the five runs. In the following sections the runtime, power and energy usage of the algorithms are investigated for each architecture.

4.1 CPUs

The CPU simulations are run using 24 MPI processes on a single ARCHER compute node. Table 2 show the runtimes and speed-up relative to the BL algorithm on the CPU. The variation between the runtimes for different runs for each algorithm was less than 1% from the average.

Table 2: Runtime (s) and speed-up (relative to BL) for all algorithms on the CPU.

Algorithm	BL	RS	SS	RA	SN	SN2
Runtime (s)	1216.39	715.51	690.30	558.70	549.73	559.86
Speed-up	1.00	1.70	1.76	2.18	2.21	2.17

All other algorithms outperform the BL case and a speed-up of (∼1.7-2.2) relative to BL was achieved as reported in [8]. On a CPU, the runtimes of the minimal storage algorithms (SN, RA and SN2) are within 2% of each other and are faster than the intermediate storage algorithms. It is worth noting that in [8] the SS and SN algorithms are within $\sim$ 1-2% on the same ARCHER CPU hardware. The difference in the runtime of the current simulations compared to the results reported by [8] was likely due to different compilers being used (the Cray C compiler version 2.4.2 in [8], versus the Intel C compiler in the present work), which caused different optimisations to be performed at compile-time.

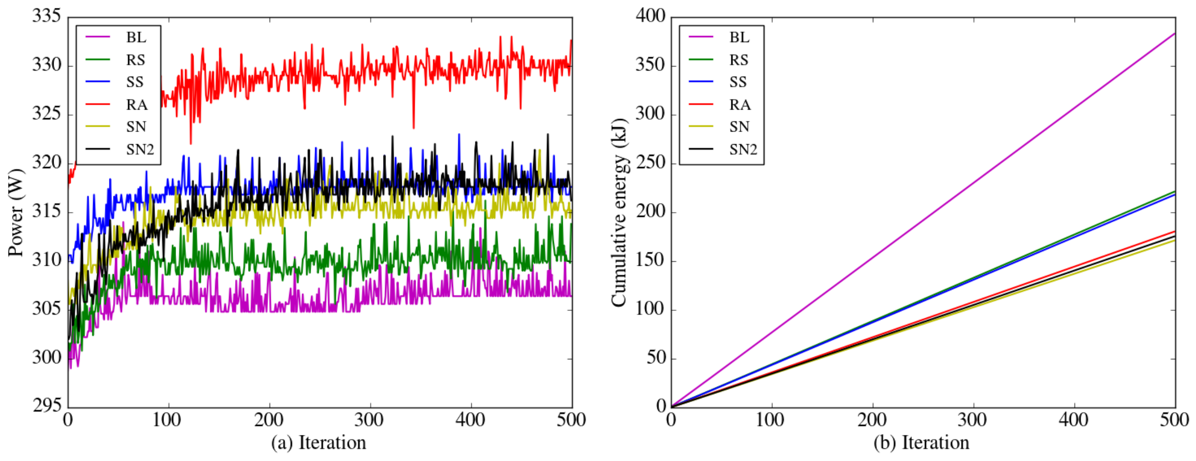

Figure 2: (a) Power consumption per iteration (W), and (b) cumulative energy (kJ) for the different algorithms on the CPU node.

Figure 2(a) shows the power consumption of the algorithms for each iteration. The steady rise in power at early times, which is common to all runs, is due to Dynamic Voltage and Frequency Scaling (DVFS) seen in other studies of power consumption on CPUs [2]. The BL algorithm consumes the least amount of power per iteration whereas RA consumes the most. The higher power consumption for the compute intensive algorithms is expected due to the larger number of operations being performed. Note that SN and SN2 consume less power than RA since they reuse some of the thread-/process-local variables. With respect to the intermediate storage algorithms, RS consumes less power relative to SS. This is quite different to the behaviour of the minimal storage algorithms (where more calculations lead to increased power consumption), and demonstrates that in order to be energy efficient, a balance needs to be achieved between the number of computations performed and the amount of local-storage used.

Figure 2(b) shows the evolution of cumulative energy for each algorithm. As this represents both power consumption and run-time, the lower the value, the more energy efficient the algorithm is. In this figure the different classes of algorithms can be easily identified, irrespective of the power consumption behaviour. BL uses the highest amount of energy, whereas all the compute intensive algorithms are close to each other, consuming the least amount of energy. The intermediate storage algorithms consume more energy than the compute-intensive algorithms but still far less energy than BL. The energy saving of an algorithm, defined as the ratio of energy used by the BL algorithm to the energy used by that algorithm are 1.0, 1.73, 1.76, 2.12, 2.24, 2.18 for BL, RS, SS, RA, SN and SN2 respectively.

From the performance of different algorithms on the CPU we can conclude that: (a) the more computations that an algorithm performs with minimal read/write to RAM, the less energy is consumed; (b) energy consumption of an algorithm is directly proportional to the speed-up relative to BL; (c) the traditional way of writing large-scale finite difference codes (i.e. BL) is not optimal both in terms of runtime and energy consumption; (d) high power consumption by an algorithm does not necessarily imply that the algorithm is energy inefficient; and (e) improving the data locality of the thread-/process-local variables has negligible effect on both runtime and energy consumption, which is expected as CPUs have relatively large caches.

4.2 KNLs

Simulations were run in parallel on the KNL using 64 MPI processes. Hyper-threading was not enabled, in order to keep the configuration on the KNL similar to that of the CPU. It was also found to have negligible effect on runtime in this case. As the memory used by the algorithms for the grid size considered can fit in the 16GB of on-chip high bandwidth memory (MCDRAM memory), the entire MCDRAM is utilised in the cache memory.

Table 3: Runtime (s) and speed-up (relative to BL) for all algorithms on the KNL.

Algorithm	BL	RS	SS	RA	SN	SN2
Runtime (s)	739.61	425.02	426.05	415.59	410.96	401.99
Speed-up	1.00	1.74	1.74	1.78	1.80	1.84

Table 3 gives the runtime of the algorithms for each run and the speed-up achieved relative to BL. Similar to the CPU runs, the variation between individual runs of the same algorithm is $\sim$ 1-2% from the mean. A speed-up of $\sim$ 1.8 is achieved for all the algorithms relative to BL. No significant variation in speed-up/runtime is seen between the low and intermediate storage algorithms; this might be due to the relatively small grid size and also the use of high band-width memory. However, the trend is similar to that seen in the CPU case (i.e. BL is the slowest, the compute intensive algorithms are the fastest, and intermediate storage algorithms are in-between). With increased grid size the differences might be more pronounced. It is also interesting to note that the SN2 algorithm, which is the same as SN except that the local variable evaluation is reordered, was 2.5% faster than the SN algorithm, unlike the CPU case where SN2 was slower than SN.

Figure 3(a) shows the power consumption per iteration. Unlike CPUs, there is not much effect of DVFS at early times. The BL algorithm consumes the highest amount of power on KNL, which might arise from the exchanges between the MPI ranks, which can be reduced for hybrid MPI and OpenMP codes. The other algorithms consume a similar amount of power per iteration.

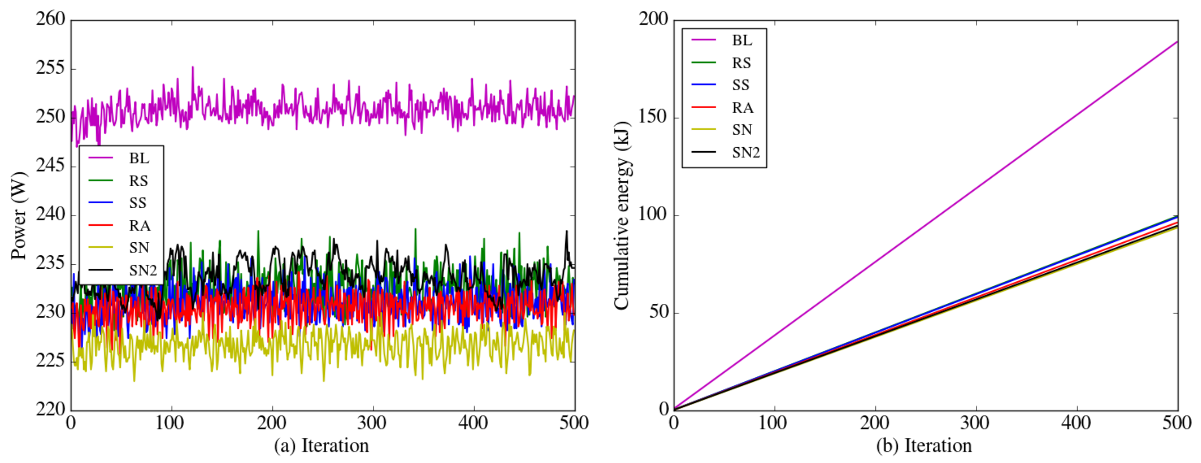

Figure 3: (a) Power consumption per iteration (W), and (b) cumulative energy (kJ) for the different algorithms on the KNL.

Figure 3(b) shows the evolution of cumulative energy consumption for each algorithm. Similar to the CPU case, the BL algorithm consumes the highest amount of energy and all other algorithms consume $\sim$ 50% less. The energy savings are 1.0, 1.9, 1.91, 1.96, 2.02 and 2.0 for BL, RS, SS, RA, SN and SN2, respectively.

The following can be concluded from the results: (a) BL is the worst performing in terms of energy and runtime; (b) the energy consumption is proportional to the speed-up achieved by the algorithm relative to BL; (c) the effects of re-ordering of derivatives in the SN2 are apparent and became more pronounced with larger grid sizes; (d) high power consumption by an algorithm does not necessarily imply that the algorithm is energy inefficient, and measuring power alone is not the best way to interpret the energy efficiency of an algorithm.

4.3 GPUs

Table 4: Runtime (s) and speed-up (relative to BL) for all algorithms on the NVIDIA Tesla K40 GPU.

Algorithm	BL	RS	SS	RA	SN	SN2
Runtime (s)	496.29	255.52	231.25	234.29	297.68	220.45
Speed-up	1.00	1.94	2.15	2.12	1.67	2.25

Table 4 shows the runtime of the algorithms on GPUs. Similar to the CPU and KNL, all algorithms perform better than BL. For the current grid size used, no specific trend can be found between low storage and intermediate storage algorithms. The fact that RA is slower than SS, which is not seen on other architectures, shows the difficulty of coding numerical methods on a GPU as these architectures are sensitive to data locality. This can be inferred from the runtimes of SN and SN2, with the only difference between them being improving data locality which gives a $\sim$ 30% reduction in runtime. The order in which we write the evaluations while performing compute-intensive operations does indeed have an effect on the runtime on a GPU as shown in [1] .

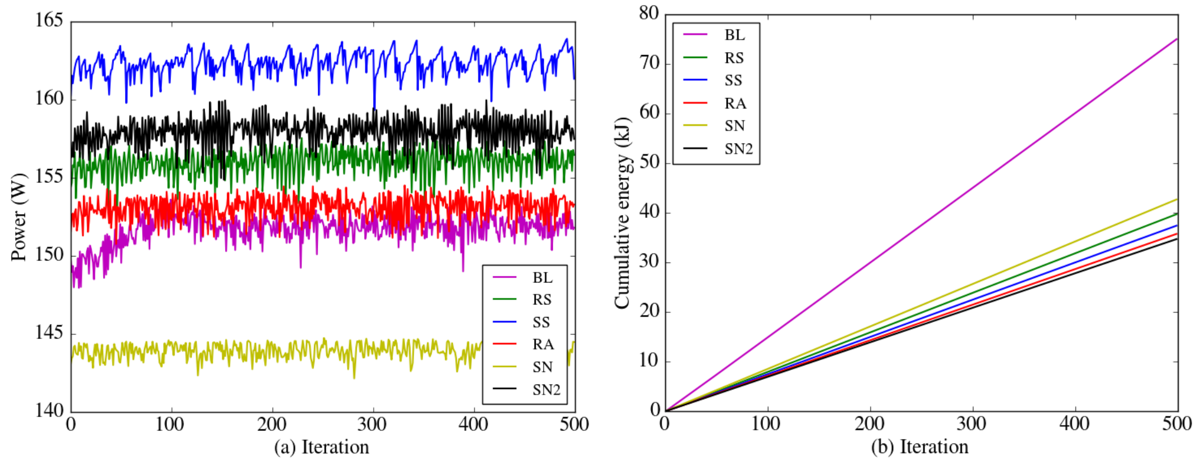

Figure 4: (a) Power consumption per iteration (W), and (b) cumulative energy (kJ) for the different algorithms on the GPU.

Figure 4(a) shows the power consumed per iteration. On the GPU, the SN algorithm uses less power due to the lack of data locality. If we ignore the SN algorithm from the current discussion, then BL uses less power and interestingly SS uses more power. The power usage by the intermediate storage algorithms is higher than that of the most computationally-intensive algorithm (RA), again indicating that data locality has a critical impact on GPU performance.

Figure 4(b) shows the evolution of cumulative energy. Similar to the CPU and KNL, the BL algorithm consumes the highest amount of energy. If we set aside the SN algorithm from the discussion, the SN2 algorithm uses the least amount of energy. Even though the runtime of RA is higher than the runtime of SS by 1.5%, the energy consumption of RA is less than SS by 5%. This means that RA algorithm can be improved further by writing the large expression in such a way that data locality is enhanced. The energy savings of the algorithms are 1.0, 1.89, 2.01, 2.1 and 2.16 for BL, RS, SS, RA and SN2 respectively, showing a similar trend to the CPU and KNL.

We can conclude the following for GPU: (a) the performance of BL is inefficient both in terms of runtime and energy; (b) data locality is inversely related to the power consumed; (c) care should be taken while optimising algorithms on GPU to improve data locality; (d) reading and writing to arrays should be minimised, even though the computational intensity of an algorithm increases; (e) all the algorithms consume half the amount of energy relative to BL.

5 Summary and conclusions

The route to exascale presents additional challenges to the numerical modeller. Not only is it crucial to consider runtime performance of the algorithms that underpin finite-difference codes, it is also important for model developers to consider the energy efficiency of their codes. This paper has highlighted the benefits of using an automated code generation framework such as OpenSBLI to readily vary the computational and memory intensity of the finite difference algorithms in order to evaluate their performance.

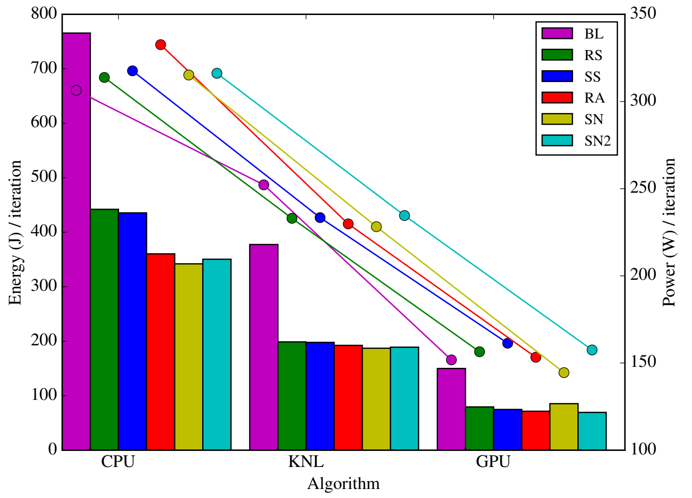

Figure 5: Energy per iteration (bars) and the power consumed per iteration (lines) on different architectures.

To summarise the findings, Figure 5 shows the energy consumed per iteration and the average power consumption per iteration for each algorithm on the three architectures considered. The CPU node uses more power per iteration than a KNL or GPU. GPUs use about 50% less power than a CPU node and the KNL system is in-between. The trend is similar for energy consumption per iteration on three architectures, which agrees with the findings of [5], in which a second order central finite difference scheme was used for solving the acoustic diffusion model equation; CPUs use more energy and GPU use less and Xeon Phi (Knights Corner) processors are in-between.

Figure 6 shows a comparison of speed-up of the algorithms and the energy saved by the algorithms on different architectures. The energy saved is in a direct proportion to the speed-up relative to the baseline algorithm.

The conclusions from the present work include: (a) Across all architectures, the low storage/ high compute intensity algorithms are more energy efficient and gives the best performance; (b) Runtime reduction reduces the energy used by the algorithm; (c) For all the algorithms considered CPUs are the least energy efficient and GPUs are the most energy efficient, with the KNL architecture in-between; (d) For the high compute intensity algorithms, the energy saving of KNL and GPU are ~ 2 and ~ 5 compared to the CPU node; (e) For optimising the simulations on GPU, care should be taken to improve data locality.

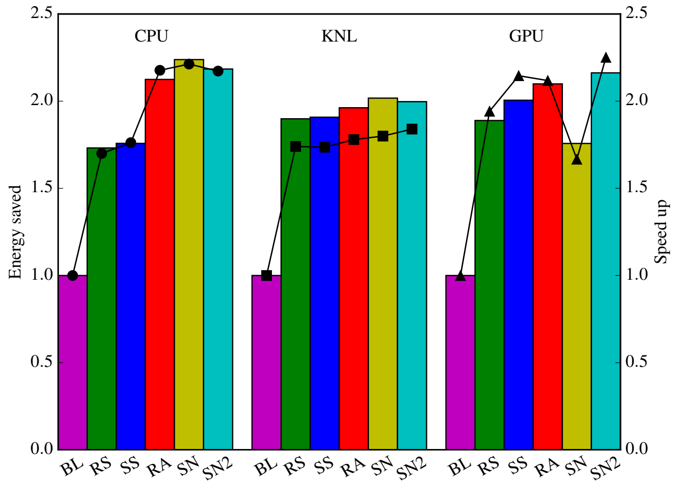

Figure 6: Energy saved (bars) and the speed-up (lines) of algorithms relative to the BL algorithm for each architecture.

6 Acknowledgements

SPJ and CTJ were supported by a European Commission Horizon 2020 project entitled "ExaFLOW: Enabling Exascale Fluid Dynamics Simulations" (671571). DJL was supported by an EPSRC Centre for Doctoral Training grant (EP/L015382/1). The authors acknowledge the use of the UK National Supercomputing Service (ARCHER), with computing time provided by the UK Turbulence Consortium (EPSRC grant EP/L000261/1). The authors would also like to thank the NVIDIA Corporation for donating the Tesla K40c GPU used in this work.

References

[1] S. Ashby, P. Beckman, J. Chen, P. Collela, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegel, F. Streitz, A. White, and M. Wright. The Opportunities and Challenges of Exascale Computing: Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee. Technical report, U.S. Department of Energy, Office of Science, 2010.

[2] M. Bareford, N. Johnson, and M. Weiland. On the Trade-offs between Energy to Solution and Runtime for Real-World CFD Test-Cases. In *Proc. of the Exascale Applications and Software Conf. 2016*, number 6, 2016.

[3] M. R. Bareford. The PAT MPI Library (Version 2.0.0). Online https://github.com/cresta-eu/pat_mpi_lib, 2016.

[4] G. A. Blaisdell, E. T. Spyropoulos, and J. H. Qin. The effect of the formulation of

nonlinear terms on aliasing errors in spectral methods. *Applied Num. Mathematics*, 21(3):207–219, 1996.

[5] M. Hernández, B. Imbernón, J. M. Navarro, J. M. García, J. M. Cebrián, and J. M. Cecilia. Evaluation of the 3-D finite difference implementation of the acoustic diffusion equation model on massively parallel architectures. *Computers and Electrical Engineering*, 46(January):190–201, 2015.

[6] C. T. Jacobs, S. P. Jammy, and N. D. Sandham. OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. *J. of Comp. Science*, 18:12–23, 2017.

[7] S. P. Jammy, C. T. Jacobs, and N. D. Sandham. Enstrophy and kinetic energy data from 3D Taylor-Green vortex simulations. University of Southampton ePrints repository 10.5258/SOTON/401892, 2016.

[8] S. P. Jammy, C. T. Jacobs, and N. D. Sandham. Performance evaluation of explicit finite difference algorithms with varying amounts of computational and memory intensity. *J. of Comp. Science*, In Press.

[9] S. P. Jammy, G. R. Mudalige, I. Z. Reguly, N. D. Sandham, and M. Giles. Block-structured compressible navier-stokes solution using the ops high-level abstraction. *Int. J. of Comp. Fluid Dynamics*, 30(6):450–454, 2016.

[10] R. Miller. Exascale Computing = GigaWatts of Power. Online http://www.greendatacenternews.org/articles/share/178272/, 2010.

[11] G. R. Mudalige, I. Z. Reguly, M. B. Giles, A. C. Mallinson, W. P. Gaudin, and J. A. Herdman. Performance Analysis of a High-level Abstractions-based Hydrocode on Future Computing Systems. In *Proc. of the 5th Int. workshop on PMBS '14, Supercomputing 2014* , 2014.

[12] NVIDIA. Tesla K40 GPU Accelerator: Board Specification. Technical Report BD-06902-001_v05, NVIDIA, November 2013.

[13] NVIDIA. NVML: Reference Manual. Technical Report vR352, NVIDIA, May 2015.

[14] N. Sandham, Q. Li, and H. Yee. Entropy Splitting for High-Order Num. Simulation of Compressible Turbulence. *J. of Comp. Physics*, 178(2):307–322, 2002.