Hostile Cache Implications for Small, Dense Linear Solves

Tom Deakin*, James Cownie*, Simon McIntosh-Smith*, Justin Lovegrove† and Richard Smedley-Stevenson†
*Department of Computer Science, University of Bristol, UK
Email: {tom.deakin, s.mcintosh-smith}@bristol.ac.uk
†Computational Physics Group, Atomic Weapons Establishment, Aldermaston, UK
Email: {Justin.Lovegrove, Richard.Smedley-Stevenson}@awe.co.uk

***Abstract*—The full assembly of the stiffness matrix in finite element codes can be prohibitive in terms of memory footprint resulting from storing that enormous matrix. An optimisation and work around, particularly effective for discontinuous Galerkin based approaches, is to construct and solve the small dense linear systems locally within each element and avoid the global assembly entirely. The different independent linear systems can be solved concurrently in a batched manner, however we have found that the memory subsystem can show destructive behaviour in this paradigm, severely affecting the performance. In this paper we demonstrate the range of performance that can be obtained by allocating the local systems differently, along with evidence to attribute the reasons behind these differences.**

***Index Terms*—finite element method; batched linear algebra; cache; memory allocation**

I. Introduction

The finite element method is a class of discretisation methods used in scientific computing. They are attractive because they can provide high accuracy for the solution. In general, the solution in the problem domain is sought as a combination of functions over a localised area. In the spatial domain, they are commonly used with unstructured meshes and can represent complicated and sometimes curved geometries.

However, this comes at a price. Finite element methods require more computation than other discretisation methods such as finite difference or finite volume. Those are typically computed with a simple stencil weighted average over neighbouring points, whereas for finite elements the solution is found by first producing and solving a linear system;

$$Ax = b \tag{1}$$

for a matrix A, known vector b and the unknown vector you're trying to solve for x. This system is produced and solved for each element. The matrix is also classed as *dense*, meaning most elements are non-zero (in comparison to sparse matrices which contain mostly zeros).

In general, multiple linear systems can be constructed and solved concurrently for independent elements. This provides parallelism and is exploited when writing a high performance finite element solver. Thus, many of these matrices are formed at once, allowing the resulting systems to be solved in parallel and the results stored directly. This approach is beneficial as storing the large global assembly matrix can often be prohibitive in terms of memory footprint. For small problems, or those of low dimensionality, this may be viable, but for large or highly dimensional problems (such as S_N transport [1]) a global-matrix-free approach is required. The global matrix representing the entire state is therefore not constructed due to its memory footprint. Instead, the local contribution from each element is calculated. In this formulation the problem therefore requires a solver to construct and then solve small linear systems in parallel.

There are natural comparisons to be made with batched BLAS routines in this goal [2]. Although a direct linear solve aligns more closely with LINPACK, the fundamental similarities of a "batching" approach for performance linear algebra operations on multiple matrices at once is clear. For the batched routines found in most BLAS libraries, a list of matrices is passed to the API. This therefore requires that the matrices (for our purposes: systems) are created in advance. Recall that this is prohibitive for the global-matrix-free approach where it is not possible to store all these matrices (otherwise we could have just used a standard parallel linear solver library).

Therefore small linear systems are created, solved and destroyed in parallel. Importantly these systems are small enough to fit in close levels of cache. A typical 3D linear Lagrangian hexahedral (i.e. a cube) discontinuous finite element forms an 8-by-8 matrix. In FP64 double precision this only requires 512 bytes of storage, well within the usual 32 KiB first level cache sizes of many modern processors. Do not forget also about the size of the register file, for example with Intel AVX-512 there are 32 512-bit registers, equivalent to 2 KiB of storage. Higher element orders are of course used: cubic elements would form a 64-by-64 matrix and fill the L1 cache with a footprint of 32 KiB. What is clear however, is that cache hierarchies are large enough to contain the linear systems being solved concurrently. Thus the cache performance becomes the limiting factor in this approach [3], [4].

This is a stark difference from the case where storing all the

local matrices would exceeded main memory capacity, so the advantages of this approach are clear. It does however require different ways to analyse performance as typical performance models are ill equipped for providing in detail information for cache-resident applications.

In this paper we:

- Highlight the performance challenges of cache-resident direct linear solves.
- Experiment with memory allocation schemes to improve the performance and show marked differences in the achievable performance of the linear solves.
- Show scaling results highlighting the interaction between low-level caches in a many-core CPU.

II. Batched Gaussian Elimination

Gaussian elimination is a standard approach for solving linear systems explained in many introductory linear algebra text-books. It is used to find the solution vector x in the matrix equation $Ax = b$ where the matrix A and vector b are known. The procedure works in two phases. First, multiples of each row are subtracted from rows placed lower in the matrix to form an upper triangular matrix. Triangular linear systems are then trivial to solve; the bottom row is known as it contains only one non-zero entry thus defining the solution at this position. The known value can then be inserted into the higher rows reducing the higher rows by one unknown. Repeating this is the second phase. Gaussian elimination is summarised in Algorithm 1. Note we use the notation $A_{:,j}$ to mean the jth row of matrix A, and $A_{i,j}$ to mean the ith entry in the jth row.

Gaussian elimination is known as a direct linear solver, since when the procedure is followed once the solution is known. For large and particular sparse systems, this becomes prohibitively expensive and iterative methods are used instead. These use a technique which produces ever improving approximations to the solution. Much of their attraction comes from the ability to avoid the triangularisation (matrix inversion) step which may not be possible. For small dense invertible matrices however, direct solvers are sufficient. It is the subject of future work to consider the impact of using iterative methods to partially solve the local systems and rely on outer convergence criteria typically present in finite element codes to work towards the correct solution.

It is worth discussing the inherently serial nature of many parts of this algorithm. The outer loops over the rows must be performed sequentially. However, the updates to the lower rows and to the entries in a given row can be performed concurrently. The situation is similar for the backwards substitution step; here parallelism is somewhat limited by the length of the rows. For 3D hexahedral finite elements, the rows begin at length eight which is of sufficient length to vectorise. As we store the matrices in row-major order the memory access along a row is contiguous. To this end we do not exploit the changing parallelism of "lower rows" on CPUs and only exploit the fixed parallelism along each row. For backwards elimination we vectorise similarly, however note that due to

Input: $N \times N$ matrix A
Input: Right hand side vector b of length N
Output: Vector x of length N
`/* Triangularisation */`
foreach *row* j *in* A **do**
  if *diagonal entry* $A_{j,j}$ *is not zero* **then**
    foreach i *lower row* $j+1$ *to* N **do**
      $c = A_{j,i}/A_{j,j}$
      `/* Subtract multiple of jth row */`
      $A_{:,i} = A_{:,i} - (A_{:,j} \times c)$
      `/* Update right hand side */`
      $b_i = b_i - (b_j \times c)$
    end
  end
end
`/* Backwards substitution */`
foreach *row* j *from* N *to* 1 **do**
  `/* Set the solution */`
  $x_j = b_j/A_{j,j}$
  `/* Propagate known results */`
  foreach *row* i *from* 1 *to* $j-1$ **do**
    $b_i = b_i - x_j \times A_{j,i}$
    $A_{j,i} = 0$
  end
end

Algorithm 1: Gaussian elimination

the expanding nature of the available entries the vectors are not always full. In this way, each linear system provides work for one CPU core and uses the vector units. By solving multiple systems concurrently we can exploit the entire CPU; each core constructs and solves a number of independent linear systems. Our parallel implementation of this routine is available as an open source project on GitHub[1].

A. Results on different many-core CPUs

In this study we use a number of different many-core CPUs from Intel®, AMD®, IBM® and Marvell® (Arm®). Their details are summarised in Table I. These processors all have three levels of cache, with the capacities shown in the table. The L1 caches on all the processors are the same size, though they differ in their associativity. For all of the processors except Skylake, the higher levels of cache are inclusive of the lower levels, so data is replicated in all levels of cache. For Skylake the L3 cache is exclusive and does not contain a copy of L2; the L3 cache here typically operates as a victim cache.

The aggregate cache bandwidth is measured using cachebw [5] and are plotted in Figure 1. Observe the steep "cliff edges" where the total memory used exceeds a level of cache; all the processors have 32 KiB of L1 cache and a clear drop is seen after 2^{15} in the figure. The high core count

[1]https://github.com/UoB-HPC/batched_gaussian_elimination

TABLE I
PROCESSOR INFORMATION COLLATED FROM VENDOR TECHNICAL SPECIFICATIONS

Processor	**Type**	**Vendor**	**SKU**	**Cores**	**Clock** (GHz)	**L1 per core** (KB)	**L2 per core** (KB)	**L3 per socket** (MB)
Broadwell	2S CPU	Intel	Xeon E5-2695 v4	2 x 18	2.10	32	256	45
Skylake	2S CPU	Intel	Xeon Platinum 8176	2 x 28	2.10	32	1024	38.5
Rome	2S CPU	AMD	EPYC 7742	2 x 64	2.25	32	512	256
Power 9	2S CPU	IBM	-	2 x 20	3.20	32	512	9.8
ThunderX2	2S CPU	Marvell	-	2 x 32	2.50	32	256	28

2S indicates dual-socket system.

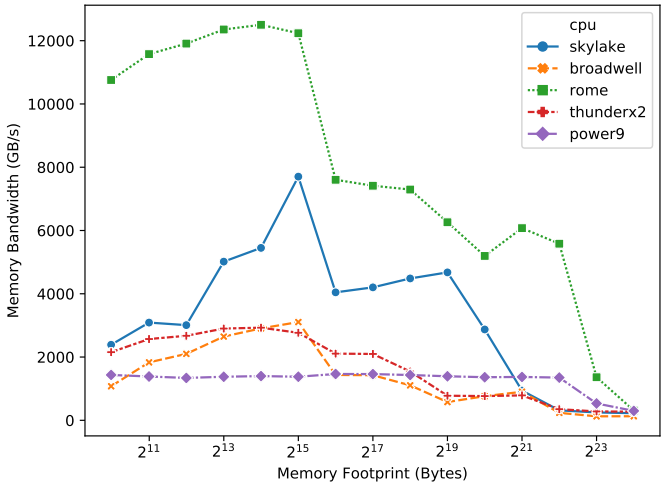

Fig. 1. Aggregate cache bandwidth of the processors

of Rome provides a high aggregate bandwidth despite having half the vector width (256-bit) of Skylake (512-bit).

We first explore the baseline performance of parallel batched Gaussian Elimination. We use OpenMP® code to share batches of 200 systems between all available cores, and we run 50 thousand iterations thus solving 10 million systems in total. A single shared array is allocated in the heap to store one matrix per physical core, each core operating only on its own part of the array; this is a standard approach to memory allocation.

The total runtime is shown in Table II for a variety of different matrix sizes which correspond to 1st–4th order finite elements. All cores are utilised and each computes a share of the batch of 200 matrices. Note that these are the baseline timings, and we will later show how the performance can be improved by changing the memory allocation scheme. Note that the 64-by-64 matrix is 32 KiB which will saturate the entire L1 cache of all of these processors, and we can clearly see a significant increase in runtime for a larger matrix that exceeds L1 cache capacity: the 125-by-125 matrix is 3.8X larger in size than the 64-by-64 matrix but the runtime exceeds that ratio. As such, the penalty for reading from more distant memory is clearly felt for this application. Note that although the memory footprint for an n-by-n matrix scale as $O(n^2)$, the

TABLE II
BASELINE RUNTIME (IN SECONDS) OF BATCHED GAUSSIAN ELIMINATION FOR DIFFERENT ORDERS ON VARIOUS MANY-CORE CPUS

	Processor				
Matrix size	Broadwell	Skylake	Power 9	ThunderX2	Rome
8-by-8	1.533	1.066	2.067	0.582	1.933
27-by-27	3.806	2.877	3.421	2.976	2.703
64-by-64	5.761	4.850	11.250	7.245	5.052
125-by-125	57.303	29.994	101.675	74.074	22.306

time complexity scales as $O(n^3)$, however due to vectorisation the number of steps required is reduced by the vector length, so one would expect performance to scale linearly with the memory footprint.

B. Matrix memory allocation schemes

We have found that the baseline runtimes shown in Section II-A can be improved upon in many cases simply by changing how the matrices are allocated, and, therefore, where the OS chooses to map the required pages. We show results for three different methods:

1) Each thread allocates its own matrix and vector, which are allocated in parallel.
2) The initial serial thread allocates one matrix and vector *per* thread.
3) Each array is allocated separately with space to store one matrix/vector per thread.

The baseline results correspond to method 3 which turns out normally to be the slowest of these methods.

Method 3 is the typical way we might program, allocating sufficient memory on the heap for our parallel work. Each thread will only access a fixed portion of the array with no overlap with data used by any other thread. Note too that since the data accessed will exceed cache lines no sharing of data within a cache line should occur. We can ensure this by aligning the start of the array to the beginning of a cache line, however in practice we found this has no noticeable effect on the runtime.

Methods 1 and 2 treat memory in the same way. We use a separate memory allocation for each thread's data, with each thread using different base pointers to access the memory. This involves many small memory allocations, however it may encourage the system to place each allocation in its own page

of memory. The methods differ solely in which thread calls the memory allocator. For method 2 the initial, serial, thread allocates the memory on behalf of the other threads. Note that memory is allocated (i.e. address space is reserved) but the data is not initialised; to take advantage of first-touch each thread will initialise its own data. Method 1 however calls the allocator in parallel, with each thread allocating its own pointer as well as subsequently initialising its own data.

Critically, the real computation is identical for all methods. Each thread only ever touches the data associated with it and so no sharing of data between the cores is algothmically required.

We run these three methods on our different CPU architectures and compute the relative improvement relative to the baseline results (Method 3) on that architecture previously shown in Table II. This is plotted in Figure 2, where each chart shows the improvement for each matrix size.

On the Intel Broadwell and Skylake processors, there is a marked improvement from having separate memory allocations per thread. For the smallest matrix size the improvement can be up to 6X. The magnitude of improvement however reduces as the matrix size increases until the matrix size exceeds L1 cache capacity where we observe little difference in runtime. On the ThunderX2, a 2X improvement can still be found for the smallest matrix size even for the large matrices. The IBM Power 9 processor does not seem to benefit from this approach, however note from Table II that it did not offer compelling performance compared to the other processors in this study.

We can also see differences between the memory allocation methods 1 and 2. These only differ in which thread calls the allocator; one array per thread is allocated in either method. Recall too that each thread is the first to touch the data it requires, so although in the case of method 2 the initial thread calls the allocator (creating the pointer to the data) that allocated data is only used by the designated thread. Figure 2 shows that for the two smallest matrix sizes there are marked differences between these approaches on the Intel processors. The other platforms do not show this difference.

We used the Numastat Linux tool[2] to check for page migrations across sockets, however all allocation methods show similar results. Therefore we do not believe the difference is coming from typical NUMA-related issues, even though the symptoms are similar.

We can use profilers to measure the average number of cycles taken per instruction (CPI). This is reported by `uarch-exploration` in Intel Advisor for example. Here we observe clear differences for the allocation methods, with method 1 showing 0.4 CPI, method 2 showing 1.2 CPI and method 3 4.8 CPI (for a 12-core single socket Skylake node where the profiler was available). This increase can be attributed to congested access to L3 cache in the slower methods which the profiler reports. This is unusual because the data is small enough to be resident in L1 cache. Therefore this indicates that the processor is encountering false sharing behaviour, most noticeably in method 3. This method does require *pages* to be shared between cores. A possible mitigation is to pad the data to try to ensure pages are not shared, and, while we found this produced a 2X speedup on the Broadwell processor, this is still far short of the 6X improvement offered by method 1.

TABLE III
SELECTED PERF C2C OUTPUT NORMALISED BY RUNTIME ON BROADWELL FOR THE 8X8 MATRIX SIZE

	Normalised sample count		
Record	Method 1	Method 2	Method 3
Total records	904,107.37	810,994.12	793,611.31
Load Operations	441,758.21	383,101.35	294,611.10
Load Fill Buffer Hit	13,329.22	117,091.45	187,793.55
Load L1D hit	421,217.03	237,674.09	51,754.34
Load L2D hit	49.76	70.49	298.38
Load LLC hit	7,143.54	28,254.95	54,760.38
Load Local HITM	4,030.49	24,271.39	47,559.99

The Linux Perf C2C tool[3] can also provide in depth information about cache lines being falsely shared between cores by measuring loads that "hit in a modified cache line" (HITM). We collected these statistics on the Broadwell processor. This measurement is made by sampling the running application and so due to the variation in runtime observed between the different methods we must compare the results carefully. We found that by normalising the statistics to the runtime we can interpret the data in terms of instructions per second. This yields a similar number of samples collected for all three options. We show some of the key output in Table III.

The load operations row shows the number of loads executed per second, so the fastest method 1 shows more loads per second as the it completes the problem in less time. The other rows in the table show where those loads were satisfied from. It is clear that method 1 has the vast majority (95%) of these operations satisfied from L1 cache, whereas methods 2 and 3 have 62% and 18% respectively from there. We also see a large number of fill buffer hits for 2 and 3 which are known as secondary L1 misses where the request for the first miss is still in flight and a second load to the same data has been issued. This shows that for this kernel, L1 misses give severe latency penalties due to the high reuse of the small amount of data.

The data here also highlights a false sharing issue in method 3. There are a high number of last level cache (LLC) hits, meaning that the data has had to be loaded all the way from the inclusive L3. This also corresponds to a comparatively high number of HITM samples recorded, meaning that cache lines have changed invoking the cache coherency protocols. This should be unlikely as cores do not modify cache lines since they only operate on adjacent chunks of the array spanning multiple cache lines. It is our working hypothesis that the shared array in method 3 may be invoking

[2]https://man7.org/linux/man-pages/man8/numastat.8.html

[3]https://man7.org/linux/man-pages/man1/perf-c2c.1.html

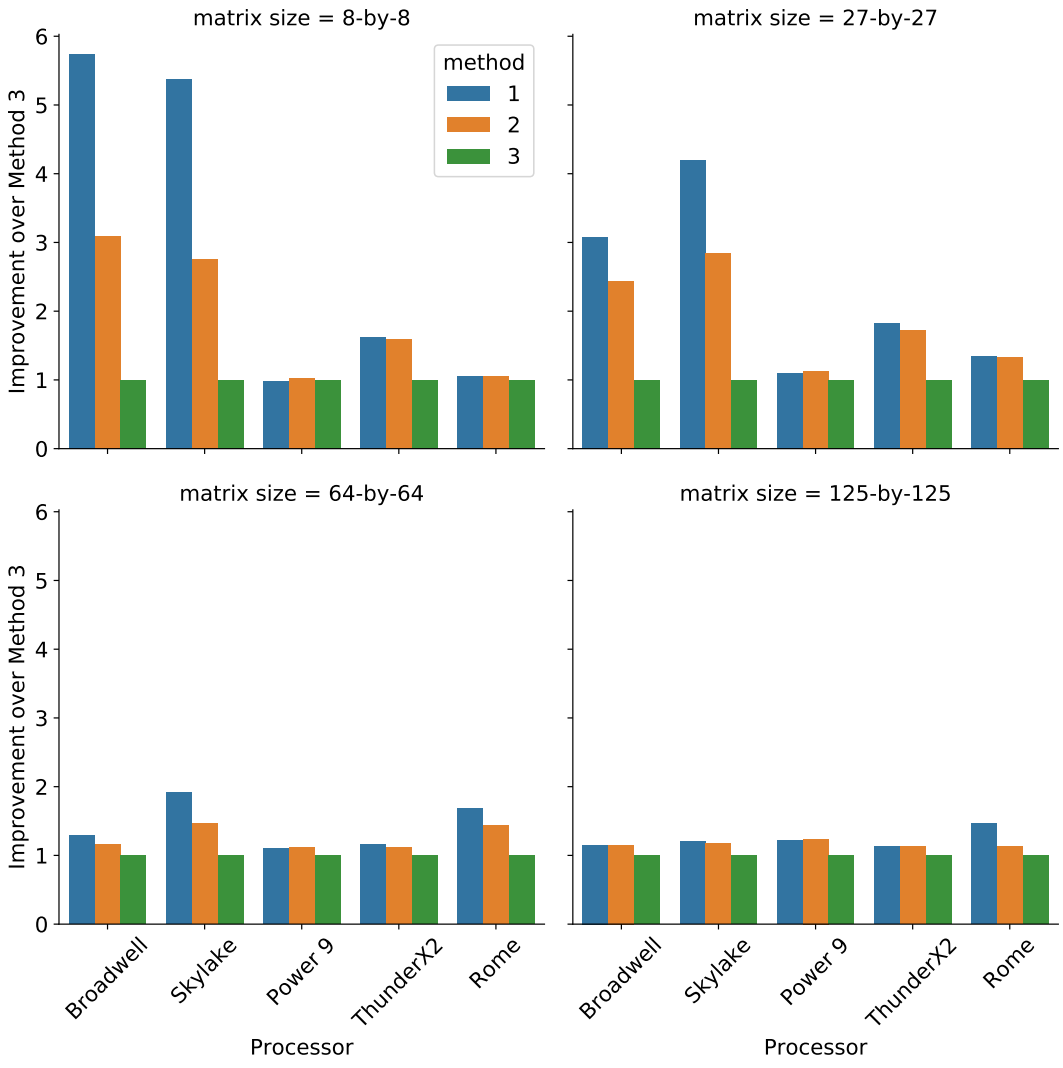

Fig. 2. Relative improvement over the baseline implementation of different memory allocation methods

hardware prefetchers which evict useful data in favour of data from nearby cores. This would explain the high number of HITM records as those lines are indeed modified by their original owner. Cache hierarchies are designed be hidden from application developers, and the prefetchers are designed to predict the future data required by the core in advance in order to minimise the latency costs (which are severe as seen in Figure 1). Among otheer things this prediction will rely on temporal locality, so data near to what is required now will be prefetched. However, that is not true of this workload. Another example of such conflicting memory access requirements occurs where there are small amounts of highly reused data combined with a large data set which is streamed through the memory [6]. It is a key message that when these cache mechanisms work against the needs of the application, it is a significant challenge to understand and mitigate these effects.

It is also useful to record the final runtimes to allow for comparisons between architectures, hence Table IV shows the runtime from method 1. For the smallest matrix size, the Intel processors provide the best performance. On the AMD Rome we find a large fixed overhead independent of the number of matrices in the batch which may suggest that the GCC OpenMP runtime needs additional optimisation on this platform, since each batch requires OpenMP threads to synchronise. For the larger matrix sizes however, we see the benefits of the L2 cache bandwidth on Rome as observed in Figure 1: 1.9X more L2 bandwidth than Skylake translates into 1.6X faster solves for batched Gaussian Elimination.

On the AMD Rome platform we also find some significant differences in the performance of unvectorised code. The Intel 2020 compiler did not produce vector instructions here, and we saw runtimes for method 1 of 0.27s, 0.93s, 7.35s and 54.00s for the four matrix sizes respectively. Comparing the results presented in Table IV obtained with the GCC, which does vectorise, we see that for the small matrix size there is

TABLE IV
RUNTIME (IN SECONDS) OF BATCHED GAUSSIAN ELIMINATION FOR DIFFERENT ORDERS ON VARIOUS MANY-CORE CPUS USING ALLOCATION METHOD 1

	Processor				
Matrix size	Broadwell	Skylake	Power 9	ThunderX2	Rome
8-by-8	0.267	0.198	2.113	0.360	1.845
27-by-27	1.234	0.685	3.112	1.628	2.024
64-by-64	4.478	2.529	10.212	6.270	3.012
125-by-125	49.880	25.087	83.106	65.762	15.286

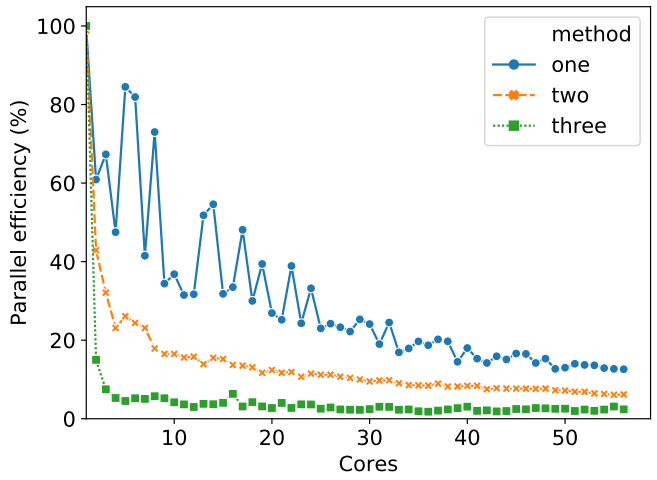

Fig. 3. Strong scaling for 8-by-8 matrices on Skylake shown as parallel efficiency

significant benefit in running scalar floating point operations, yet for the larger size vector operations are critical for high performance. On Intel platforms, the Intel compiler produces both scalar and vectorised versions of the loops, with Intel Advisor showing the serial (remainder) loop was chosen for the backwards substitution.

It is a significant challenge to understand the nature of caches in further detail in real hardware. It is the subject of future work to explore the use of cache simulators in order to further understand the performance here. However, such simulators are often simplified from the true nature of the hardware, or do not include modelling of prefetchers whose behaviour is rarely documented.

III. SCALING BEHAVIOUR

The work within each batch is independent and so this gives concurrency to exploit multiple cores. The amount of memory allocated varies with the core count to give each parallel thread its own memory to use, and no sharing of this memory between cores should occur. As a result the work is trivially paralleliseable.

We run the batched Gaussian Elimination on the Skylake processor and strong scale from one to 56 cores, across both sockets. The parallel efficiency compared with the fastest code run with a single thread is shown in Figure 3.

For method 3 it is clear that there is very limited scalability as core count increases. This allocation approach has a shared array between all cores. This scaling behaviour also strengthens the observations regarding false sharing in Section II-B. It is clear that this allocation approach causes significant overheads limiting the benefits of using additional computational resources.

Allocation method 2 alleviates this significantly and some scaling occurs. We see a marked difference between methods 1 and 2 once again. Recall that both call the allocator once per thread, the difference being whether that call is made by the initial thread, or by each thread independently. The parallel *allocation* performed by 1 (as well as the parallel *initialisation* present in both methods) significantly improves the performance. In part this may be due to non-uniform memory access (NUMA) allocation policies since with only one thread allocating the data, it may be allocated in one domain (though "first touch" allocation should avoid this). What is less clear however is that even in this case, the memory is only touched and used for its entire lifetime on the correct socket so that migration should occur to remove the limits it would otherwise cause on the performance.

Even method 1 shows limited scalability, despite being the fastest approach. Intel processors typically undergo aggressive frequency scaling when sockets become heavily utilised in order to remain within thermal limits. Clock scaling also occurs when AVX-512 instructions are used, as is the case for some of the computation. Thus, as more cores are used, the processor frequency may throttle down. Using `perf stat` to measure the clock speed, we find that the serial code ran with a clock frequency of 3.4 GHz, which is higher than the advertised clock speed shown in Table I. When using all cores, the clock speed was measured at 2.8 GHz, 17% slower thus limiting the ability to strong scale.

Strong scaling results for batches of the larger 64-by-64 matrices across the 56-cores of the Skylake processor are shown in Figure 4. These matrices are 32 KiB in size which is also the L1 cache size of this processor. This means that the vector and matrix cannot both fit in the L1 cache. Similar trends to the smaller size can be observed although the scaling is much improved for all methods. Recall from Figure 2 that method 1 provided a 2X performance improvement over the baseline method 3, and the scaling behaviour of the former is also better for this matrix size. We observe that method 2 more closely tracks method 1 meaning the difference of which thread calls the allocator is minimised.

Method 1 therefore offers improved scalability across the cores of the many-core processors. The scalability is better for larger matrix sizes, where cores have more work to do on each system in turn. It is clear that having one shared array containing the matrices is not a scalable approach for any problem size due to the unusual false sharing behaviour.

IV. CONCLUSION

Batched linear algebra routines on small, dense matrices form a crucial part of finite element solvers. Being able to

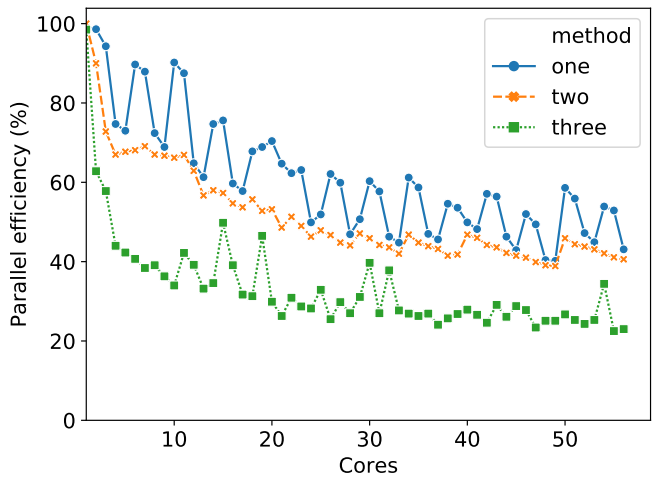

Fig. 4. Strong scaling for 64-by-64 matrices on Skylake shown as parallel efficiency

perform many such operations concurrently on matrices is important to achieving high performance on many-core CPUs. Unlike traditional BLAS routines, the matrices cannot exist in advance, and the small linear systems are created, solved and thrown away with only the solution vector saved.

This work shows that it is important to consider how such systems are allocated. One approach demonstrated performance improvements of up to 6X through mitigating an unusual form of false sharing behaviour instigated by the processor. Memory access is isolated in contiguous chunks between cores, and so any false sharing behaviour is through a misstep of the cache architecture.

Parallel scaling of the work across cores is also affected by the way the linear systems are allocated, and some allocation methods results in almost no parallel scaling.

These memory effects are most noticeable on Intel processors. Although some CPUs from AMD, Marvell/Arm and IBM still benefit somewhat from these methods, they demonstrate more modest speedups on these systems which shows that the problems we observe in the cache are less inherent in their designs. The high aggregate cache bandwidth of the AMD Rome CPU, in part from the high number of cores, still provides compelling performance for the larger matrix sizes considered in this study, despite the limits to scalability.

It is important also to highlight that these matrices are small and so are often neglected by linear algebra libraries. Additionally, due to the requirement for the matrices to be constructed and solved "on the fly" libraries are typically unable to help, so developers have to create their own implementations of these routines. It is hoped that library solutions such as MFEM [7] can assist developers by providing some of that functionality while taking on board the issues identified in this study.

In summary, we find that in order to improve the execution time and scale the parallel execution of the linear solve operations within the batch we need to minimise the risk that those linear systems appear to the CPU to share memory.

Acknowledgment

This work used the Isambard UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/P020224/1).

This work was carried out using the computational facilities of the Advanced Computing Research Centre (ACRC), University of Bristol — http://www.bristol.ac.uk/acrc/.

Access to the Cray XC50 supercomputer 'Swan' was kindly provided through the Cray Marketing Partner Network.

References

[1] T. Deakin, S. McIntosh-Smith, J. Lovegrove, R. Smedley-Stevenson, and A. Hagues, "UnSNAP: A Mini-App for Exploring the Performance of Deterministic Discrete Ordinates Transport on Unstructured Meshes," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. Belfast: IEEE, sep 2018, pp. 598–606. [Online]. Available: https://ieeexplore.ieee.org/document/8514920/

[2] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. Higham, J. Hogg, P. V. Lara, Z. Mawussi, and S. Relton, "A Proposed API for Batched Basic Linear Algebra Subprograms," pp. 1–20, 2016.

[3] T. Yamaguchi, K. Fujita, T. Ichimura, A. Naruse, J. C. Wells, C. J. Zimmer, T. P. Straatsma, M. Hori, L. Maddegedara, and N. Ueda, "Low-Order Finite Element Solver with Small Matrix-Matrix Multiplication Accelerated by AI-Specific Hardware for Crustal Deformation Computation," pp. 1–11, 2020.

[4] T. Deakin, S. McIntosh-Smith, J. Lovegrove, R. Smedley-Stevenson, and A. Hagues, "Reviewing the Computational Performance of Structured and Unstructured Grid Deterministic S N Transport Sweeps on Many-Core Architectures," *Journal of Computational and Theoretical Transport*, vol. 49, no. 3, pp. 121–143, apr 2020. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/23324309.2020.1775096

[5] M. Martineau, P. Atkinson, and S. Mcintosh-Smith, "Benchmarking the NVIDIA V100 GPU and Tensor Cores," in *HeteroPar workshop at EuroPar International Conference on Parallel and Distributed Computing*, Turin, Italy, 2018.

[6] T. Deakin, W. Gaudin, and S. McIntosh-Smith, "On the Mitigation of Cache Hostile Memory Access Patterns on Many-Core CPU Architectures." Frankfurt: Springer International Publishing, 2017, pp. 348–362. [Online]. Available: http://link.springer.com/10.1007/978-3-319-67630-2_26

[7] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, "Mfem: A modular finite element methods library," *Computers and Mathematics with Applications*, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0898122120302583

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: HOSTILE CACHE IMPLICATIONS FOR SMALL, DENSE LINEAR SOLVES

Two benchmark codes are used in this paper:

- **cachebw** measures the cache bandwidth of CPU (and GPU) processors using a Triad kernel.
- **batched_gaussian_elimination** implements a Gaussian Elimination solve on batches of small matrices in the manner seen in finite elements codes.

A. Description

1) Check-list (artifact meta information):

- **Compilation:** We used the following compilers:
  - For the Intel Broadwell, Intel 19 update 4. The compiler was invoked via `make COMPILER=INTEL`.
  - For the Intel Skylake, Intel 19 update 4. The compiler was invoked via `make COMPILER=INTEL`.
  - For the IBM Power 9, GCC 8.1. Due to this platform using non-standard GCC `-march` flags, the benchmark was built as follows: `g++ -std=c++11 -O3 -mcpu=native batched_gaussian_elimination.cpp -fopenmp -o batched_gaussian_elimination`.
  - For the Marvell ThunderX2, Cray CCE 10.0.1. The compiler was invoked via `make COMPILER=CRAY`.
  - For the AMD Rome, GCC 9.1. The compiler was invoked via `make COMPILER=GCC`.
- **Data set:** The benchmark only required the problem size to be set at run time. The binary accepts three arguments:
  1) the finite element order. We used 1–4 inclusive to generate matrices ranging in size between 8-by-8 to 125-by-125.
  2) the number of matrices to be solve in a parallel batch. For all runs we set this to be 200.
  3) the number of batches to solve. We set this to 50,000 for all runs.
- **Run-time environment:** As the benchmark used OpenMP, we ensured thread binding by setting `OMP_PLACES=cores OMP_PROC_BIND=true` and setting `OMP_NUM_THREADS` equal to the number of physical cores.
- **Hardware:** Hardware used is detailed in the main body of the paper, in Table I.
- **Publicly available?:** All code is publicly available on GitHub.

2) How software can be obtained (if available): Source code the benchmarks used in this paper are available online:

- **cachebw** is available at https://github.com/UoB-HPC/cachebw.
- **batched_gaussian_elimination** is available at https://github.com/UoB-HPC/batched_gaussian_elimination.

B. Installation

Details on building the code are provided in Section A-A1.

C. Experiment workflow

The codes were built and run on the various platforms in the configurations outlined in the main body of the paper.

The **cachebw** code uses the `test.sh` script included in the repository to run the benchmark on various input sizes in order to measure all levels of cache.

The Perf C2C tool was invoked as follows: `perf c2c report -NN -g -c offset,pid,tid,iaddr --stdio`

D. Evaluation and expected result

The **batched_gaussian_elimination** code outputs the run-time for the three allocation methods.

The **cachebw** code outputs the memory bandwidth in GB/s, along with information on the size of the input data.