Mini-app Driven Optimisation of Inertial Confinement Fusion Codes

R. F. Bird*, P. Gillies†, M. R. Bareford‡, J. A. Herdman† and S. A. Jarvis*

*Department of Computer Science
University of Warwick, UK
Email: R.Bird@warwick.ac.uk

†High Performance Computing
AWE plc
Aldermaston, UK

‡EPCC,
University of Edinburgh, UK

***Abstract*—In September 2013, the large laser-based inertial confinement fusion device housed in the National Ignition Facility at Lawrence Livermore National Laboratory, was widely acclaimed to have achieved a milestone in controlled fusion – successfully initiating a reaction that resulted in the release of more energy than the fuel absorbed. Despite this success, we remain some distance from being able to create controlled, self-sustaining fusion reactions. Inertial Confinement Fusion (ICF) represents one leading design for the generation of energy by nuclear fusion. Since the 1950s, ICF has been supported by computing simulations, providing the mathematical foundations for pulse shaping, lasers, and material shells needed to ensure effective and efficient implosion.**

The research presented here focuses on one such simulation code, *EPOCH*, a fully relativistic particle-in-cell plasma physics code, developed by a leading network of over 30 UK researchers. A significant challenge in developing large codes like EPOCH is maintaining effective scientific delivery on successive generations of high-performance computing architecture. To support this process, we adopt the use of *mini-applications* – small code proxies that encapsulate important computational properties of their larger parent counterparts. Through the development of a mini-app for EPOCH (called *miniEPOCH*), we investigate known time-step scaling issues within EPOCH and explore possible optimisations: (i) Employing loop fission to increase levels of vectorisation; (ii) Enforcing particle ordering to allow the exploitation of domain specific knowledge and, (iii) Changing underlying data storage to improve memory locality. When applied to EPOCH, these improvements represent a 2.02× speed-up in the core algorithm and a 1.55× speed-up to the overall application runtime, when executed on EPCC's Cray XC30 ARCHER platform.

I. Introduction

The UK has a long history of research into high-intensity laser plasma interactions. The UK's Central Laser Facility is home to some of the world's most advanced high power lasers, which can deliver Petawatt focused beams, with approximately 10,000 times more power than the UK National Grid, during picosecond pulses. Developments in the deployment of relativistically intense 'long' laser pulses (to compress fuel) and fast 'short' pulses (for ignition) present significant challenges in computational plasma physics. Plasmas with intense electromagnetic fields require fully kinetic models of particle distribution in 7 dimensions (3 space, 3 momentum and time); and point design for targets requires the coupling of relativistic kinetic models with long time-scale radiation hydrodynamics codes. As future gyrokinetic codes continue to develop to support plasma turbulence studies, in order to exploit facilities such as ITER, for example, the complexity of these simulations and the demands on the supporting supercomputers will also increase.

Particle-in-cell (PIC) codes are amongst the most widely used computational tools in plasma physics research, and help develop further understanding of both inertial confinement fusion (ICF) and laser-plasma interactions in general. The research presented here focuses on the Extensible PIC Open Collaboration simulation codebase, named *EPOCH* [1], which is a nationally funded, fully relativistic EM particle-in-cell plasma physics code, developed by a leading network of over 30 UK researchers.

A significant challenge in developing large codes like EPOCH is maintaining effective scientific delivery on successive generations of high-performance computing architectures. In EPOCH, collections of physical particles are represented using a smaller number of pseudoparticles; the fields generated by the motion of these pseudoparticles are calculated using a finite difference time domain on an underlying grid of fixed spatial resolution. The forces on the pseudoparticles due to the calculated fields are used to update the velocities of the pseudoparticles, and these velocities are then used to update their positions. Using this approach it is possible to reproduce the full range of classical micro-scale behaviour of a collection of charged particles. Like many codes of this type, EPOCH is Fortran-based and MPI parallelised; dynamic load balancing options exist and MPI-IO allows checkpoint re-start on an arbitrary number of processors. Legacy simulation codes designed and implemented in this way are now exhibiting poor utilisation of modern hardware features such as vector operations, and fail to fully exploit all levels of available parallelism – a problem which is exacerbated by the energy-efficient benefits available through heterogeneous computing.

The continued development, maintenance and future-proofing of EPOCH represents a significant software engineering challenge, EPOCH represents decades of development by skilled domain experts – the code is feature rich, but equally large and complex. Code porting to explore the potential benefits of new compute architectures represents a significant undertaking, and the resulting benefits of this effort may indeed be small. To help mitigate these problems, we adopt the use of *mini-applications* (commonly termed *mini-apps*) – small code proxies that encapsulate important computational properties of their larger parent counterparts [2, 3]. The existence of mini-apps is built on the premise that (i) although simulation codes may have millions of lines of source code, their performance is often dominated by a small subset of the code and, (ii) simulation codes may contain many physics models that are mathematically distinct, but in many cases exhibit similar performance characteristics. Mini-apps operate by encapsulating the most important computational operations and consolidating physics capabilities that have the same performance profiles; they will typically be orders of magnitude smaller than their parent code, and as a result be easier to port, easier to improve, easier to extend, and less likely to be subject to restrictive licensing governing their use or distribution.

This paper makes a number of contributions:

- We present the development of a new mini-app (*miniEPOCH*) for the parent code EPOCH. To our knowledge this is the first publicly recorded mini-app explicitly targeting a finite difference time domain particle-in-cell (PIC) plasma physics code;
- We utilise *miniEPOCH* to explore known performance problems within EPOCH, and in particular (i) the increasing time-step duration during simulation runtime and, (ii) high levels of cache miss rates due to particle-store fragmentation;
- Using *miniEPOCH* we explore opportunities for code optimisation, including the utilisation of shared memory, exploiting increasing vector width and improving memory locality;
- Finally, we validate these EPOCH optimisations on ARCHER, a 1.6 PFlop/s Cray XC30, housed at the UK national supercomputing centre at EPCC. These improvements demonstrate a 2.02× speed-up in the core EPOCH algorithm and a 1.55× speed-up to the overall application runtime.

```
for all species do
    for all particles do

        ▷ Move particles.
        position ← position + momentum

        ▷ Update momentum based on field effects.
        e_cell ← ⌊position⌋
        for all neighbours of e_cell do
            calculate electric field effects
        end for

        b_cell ← ⌊position + 0.5⌋
        for all neighbours of b_cell do
            calculate magnetic field effects
        end for

        momentum ← momentum + electric and magnetic field effects

        ▷ Calculate and deposit currents.
        for all neighbour cells do
            calculate current
            deposit current
        end for

    end for
end for
```

Fig. 1. Pseudocode depiction of EPOCH's core particle-in-cell (PIC) algorithm.

The remainder of this paper is organised as follows: Section II provides additional background on EPOCH and its core algorithms; Section III documents an analysis of related work; Section IV details performance and algorithmic characteristics of EPOCH and the resulting mini-app implementation; Section V presents a detailed study of code optimisation using the mini-app, and the translation of these optimisations to the parent code; Section VI concludes the paper.

II. The Extensible PIC Open Collaboration Simulation Codebase (EPOCH)

EPOCH is a nationally funded, fully relativistic EM particle-in-cell plasma physics code, developed by a leading network of over 30 UK researchers. At the core of this simulation code-base are particle push and field update algorithms, developed by Hartmut Ruhl [4], extended to include advanced features such as collisions, ionisation and quantum electrodynamics-(QED) driven coherent radiation. EPOCH tracks the electric and magnetic fields generated by the motion of pseudoparticles, and is capable of reproducing the full range of classical microscale behaviour required to accurately simulate a collection of charged particles. Fig. 1 depicts the core PIC algorithm used in EPOCH, which typically accounts for over 80% of the application runtime, and consists of the following three steps:

1) Move the particle across the physical domain, proportional to particle momenta;
2) Update the particle's momentum, based upon the local electric fields, magnetic fields, and particle shape;
3) Deposit the generated current onto the grid, to act as an intermediary for particle-particle interactions.

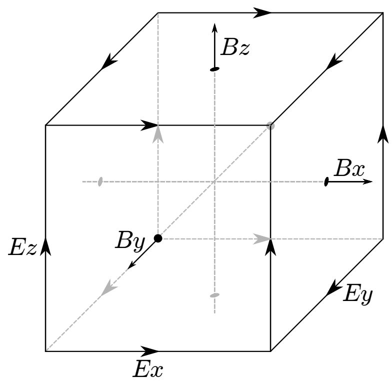

Fig. 2. A so-called Yee Grid, applying centered finite difference operators on staggered grids in space and time for each electric and magnetic vector field component.

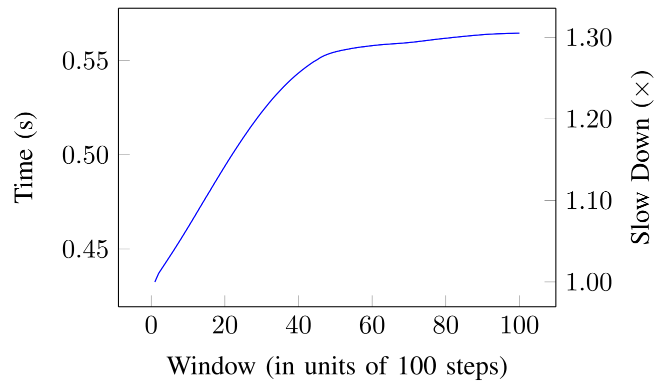

Fig. 3. The duration of each time step in EPOCH as a simulation progresses. Each simulation window, of which there are 100 in total, contains 100 steps.

These steps represent a considerable computational workload and are currently expressed as a single code kernel in which the particle loop spans approximately 500 lines of source code.

To distribute work between processing elements, EPOCH uses a static n-dimensional MPI domain decomposition, assigning a rectangular region of the physical domain to each processing element. Each MPI task is then responsible for a distinct region of the domain, transferring control of particles as they leave the space, and accepting any particles which may enter. This method of decomposition is known to exhibit poor performance for problems which display strong load imbalance, so in the interests of presenting a fair study, we limit this investigation to problems that remain well balanced throughout their operation [5].

In EPOCH, particles are densely packed, with a single particle spanning multiple grid cells (typically 3×3). EPOCH employs a Finite-Difference Time-Domain method (FDTD), and represents electrical and magnetic fields on a staggered Yee grid [6], as shown in Fig. 2. During the momentum update (step 2), a 25-point stencil in each of 3 dimensions is read per field, and used to update the particle momenta; the scale of these memory operations are a significant contributor to the overall application runtime. Once the current contributions have been calculated (step 3), they are then stored into a global array at indices determined by the grid vertices touched during particle movement. This *write* to multiple indices of a global array limits the possibility of particles simultaneously depositing current without the need for atomics or other concurrency control. It is this *current deposition* that most strongly differentiates PIC from alternative methods; unlike molecular dynamics, for example, PIC features no particle-particle interactions, instead approximating these using the Yee grid as an intermediary.

A known performance issue observed during the operation of EPOCH is that the duration of each time step increases as the simulation progresses. This problem is demonstrated in Fig. 3, where we see a sharp increase in time-step duration until approximately 4,000 steps, after which time-step duration stabilises. This observation is counter-intuitive, as there is no change to the kernel during runtime. This issue has persisted through many generations of EPOCH; in Section V we build on our knowledge and understanding gained in previous research [7] to address this time-step scaling problem, as well as using the new EPOCH mini-app to explore further code optimisation opportunities.

III. Related Work

Despite the benefits of mini-apps being identified as early as 1991 [2], the re-emergence and use of mini-apps has only gained greater traction in recent years [3]. There are now examples of mini-app use in co-design, code optimisation and porting, and in the exploration of new programming languages and paradigms. We highlight some of the more relevant work to this research.

The use of mini-apps for code optimisation is exemplified in the work by Karlin et al. [8, 9], in which the authors use the mini-app LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) to demonstrate the optimisation of multi-material hydrodynamics simulations, increasing both the performance of their mini-app (which solves a Sedov blast problem) and associated parent codes, such as Lawrence Livermore National Laboratory's (LLNL) arbitrary Lagrangian-Eulerian multi-physics code ALE3D. By using their mini-app to better focus optimisation efforts, they managed to increase LULESH's vector instruction utilisation by a factor of 8, reduce the number of memory reads by 62%, and reduce the overall application memory footprint by 19%. These improvements were then mapped back to ALE3D to achieve a 20% reduction in overall application runtime [10, 11, 12], performance gains that had remained undetected until the mini-app investigation.

Further studies demonstrate the use of mini-apps to rapidly investigate both hardware platforms and programming paradigms. Lavallée et al. [13] demonstrate how they were able to use the mini-app called HYDRO, developed by CEA Saclay, to investigate multiple hardware platforms available through the PRACE Tier-0 Research Infrastructure; similarly, the size of the mini-app greatly aided their development of several code variants of HYDRO, including those employing MPI, OpenMP, CUDA, OpenCL, HMPP and UPC – a task that would have been infeasible using the full production application. The availability of such a diverse range of code implementations allowed the authors to evaluate a variety of heterogeneous hardware and assess its viability as a future platform for the parent application RAMSES [14], an EU-funded computational astrophysics package used for the study of large-scale structure and galaxy formation. Such studies demonstrate the importance of mini-apps as a tool to enable studies in code portability, scaling, and performance.

While particle-in-cell codes are well understood [15, 16, 17, 18], the application of the mini-app software engineering methodology to the field of plasma physics and PIC remains largely unexplored. Previous work has been undertaken with the aim of providing flexible, concise environments for the development of PIC codes [19, 20]. GTC [21], at only 8,000 lines of code, is one such example of this; however, GTC is not associated with any parent code *per se*, and any findings associated with this code must still be translated to larger production codes in this code class through additional research. To the best of our knowledge, the research presented here is the first to develop and apply a PIC mini-app, which is associated with a large, production-code equivalent.

Mini-apps are increasingly being developed within open source frameworks, thus allowing the HPC community to benefit from their development. Projects such as Mantevo [3, 22] and the UK Mini-App Consortium (UKMAC) [23] exist to provide centralised repositories where collections of mini-applications can exist, selected to represent key scientific areas supported by high-performance computing simulation.

Finally, as point of clarification, the work presented here focuses on fully relativistic PIC codes, where Maxwell's equations are solved using FDTD methods; this is not the same as those solutions which utilise gyrokinetic equations, or those that employ Fourier transforms to operate in the time domain.

IV. Implementation and Optimisation

The original version of EPOCH uses a linked list to store its particles. While this itself does not present a problem, naïve implementations of linked lists offer no guarantees of contiguous memory access. This means that as data elements are inserted and deleted, memory allocations take place without consideration for locality of existing data, and considerable memory fragmentation can occur. This fragmentation can significantly impact performance, as modern hardware is optimised for contiguous memory loads, with each issued memory load fetching an entire cache line. By aligning data and promoting the grouping of data within cache lines, memory locality can be improved and one can reduce the effective bandwidth required to load the same amount of data from main memory. This effect can be seen when particles move across physical processor boundaries; as particles exit they are deleted and new particles added at arbitrary memory addresses. This means that although the initial particle allocation may be contiguous, the memory access pattern degrades over time, as a function of particle movement.

```
for all species do

    for all particles do
        ▷ Move particles.
        position ← position + momentum
    end for

    ▷ Optional Particle Sort.
    for all particles do
        ▷ Update momentum based on field effects.
        e_cell ← ⌊position⌋
        for all neighbours of e_cell do
            calculate electric field effects
        end for

        b_cell ← ⌊position + 0.5⌋
        for all neighbours of b_cell do
            calculate magnetic field effects
        end for

        momentum ← momentum + electric and magnetic field effects
    end for

    for all particles do
        ▷ Calculate and deposit currents.
        for all neighbour cells do
            calculate current
            deposit current
        end for
    end for

end for
```

Fig. 4. Pseudocode depiction of EPOCH's modified particle-in-cell (PIC) algorithm.

A second candidate for improving memory locality and, as a result, effective memory bandwidth, is to group data in memory such that it will cause subsequent loads to common addresses. In so doing, memory is more likely to be resident in cache when it is required and will reduce cache eviction and thrashing. To achieve this in miniEPOCH, we implement a particle sort to group spatially local particles in memory. We then exploit domain specific knowledge to further improve this. During the second step of the algorithm, the momentum of particles is updated based on the surrounding magnetic and electric fields. Given the staggered nature of the Yee grid (Fig. 2), it is possible to group particles which share a common vertex, and in so doing promote reuse within the 3-dimensional stencil update.

A further performance limiting factor in EPOCH, is its inefficient use of vector instructions (SIMD). Typically, it is desirable to vectorise over the most computationally intense code regions in order to fully exploit SIMD. In EPOCH this means vectorising over the *particles loop*. However, in the current expression of the algorithm, such auto-vectorisation of particles is not possible due to a classical update dependency within the current deposition. This update dependency is key to the PIC algorithm, so where the original code expresses the kernel as a single large loop, we adapt this in favour of expressing the code as three discrete steps. We explore this in our mini-app using loop fission, with pseudocode for this shown in Fig. 4. As well as promoting vectorisation, splitting

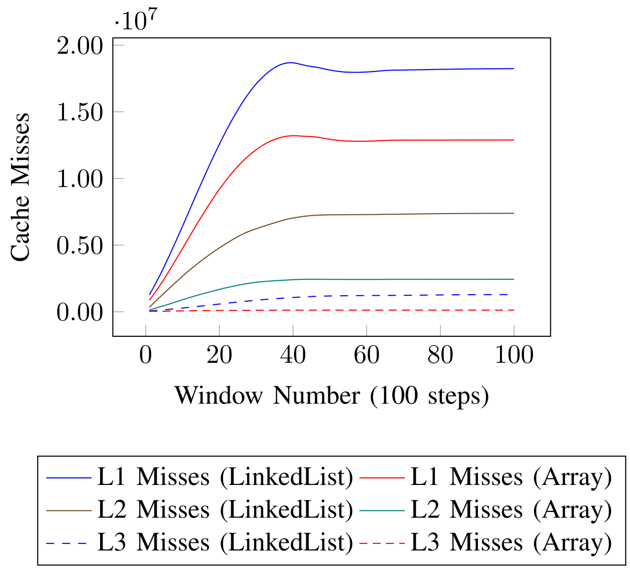

Fig. 5. Cache misses during a simulation consisting of 100 simulation windows, each window containing 100 steps.

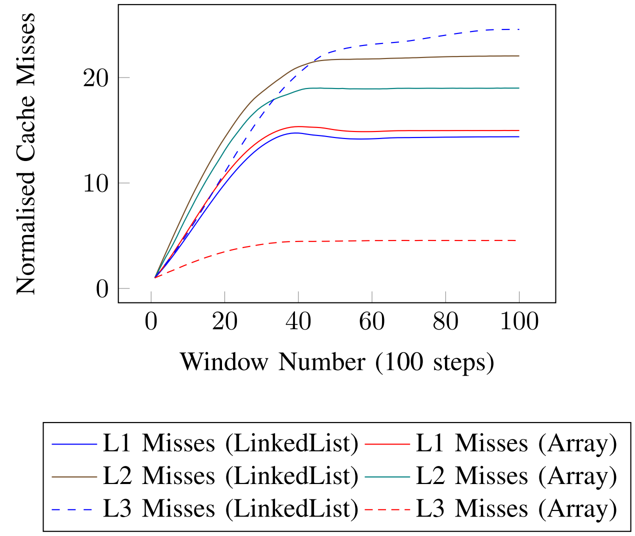

Fig. 6. Normalised cache misses during a simulation consisting of 100 simulation windows, each window containing 100 steps.

the code in this way also allows us to sort particles directly after they have been moved, giving us a stronger guarantee regarding particle reuse during the field-effect stencil. Hereafter we will refer to these three components as the *move-*, *stencil-* and *current- kernels*.

While the sort itself does increase the amount of computation required for the execution, this is mitigated by two factors: 1) Much of the computation for the sort can be done while the particle is in cache from the particle move; 2) For particles that are grouped together, we can avoid recomputing shared properties, which was not previously possible. Further to this, it is also possible to employ the sort after the *stencil* kernel, allowing us to place guarantees on the order in which the *current* kernel updates global memory, which can in turn be exploited to remove the classical update dependency and achieve vectorisation. As auto-vectorisation of all three kernels is possible, we can further increase vector efficiency by performing scalar replacement on arrays where possible, and employing SIMD lane indexing to ensure all SIMD temporary arrays have coalesced data accesses.

A. Experimental Setup

Throughout this work we report numerical results for the periodic interaction of two densely packed electron streams. Each pseudoparticle is assumed to have an individual mass, and wide spanning third order b-spline stencil. Particle probes are disabled and, unless otherwise stated, a typical problem size of 128^2 grid-cells per core is used, with 32 particles per species, per cell, on a fully packed node. The results detailed in Section V were obtained from ARCHER, a 1.6 PFlop/s Cray XC30 housed at the UK national computing centre at EPCC. ARCHER features 4,920 dual socket Intel Xeon E5-2697v2 Ivy Bridge nodes, with 24 cores per node. Intel 15.0 was used, with the highest level of code optimisation enabled (`-O3`), and with platform specific code generation (`-xHost`) enabled. PAPI 5.3.2.1 was used to gather the results of selected performance counters, including those used for recording cache misses and vector instruction counts.

V. Results

During the initial investigation of the increasing duration of time-steps in an EPOCH execution, it was believed that the primary contributor to the poor time-step scaling was increasing fragmentation of the linked list. As the particles move between MPI ranks, it was expected that the particle store would become more fragmented. As previously discussed, this was due to new particles being added to the store as they entered the domain, whilst others were removed as they leave. Fig. 5 shows the typical cache miss rates for an unsorted, linked-list implementation of EPOCH, while Fig. 6 shows this data expressed as relative differences. We can clearly see that as the simulation progresses, the cache-miss-rate increase is strongly correlated with overall runtime, and peaks at approximately 4,000 steps. After 4,000 steps, the cache miss rates increase at a much reduced rate. While this data strongly suggests that the increase in runtime is caused by the change in cache miss rates, it provides no clue as to its origin.

To investigate this further, we used the mini-app to implement an alternative, array-based, particle store. This allows for contiguous access to particles, avoiding any fragmentation. This greatly improved the time-step scaling of the mini-app, with Fig. 5 showing the representative changes in cache misses for a version of EPOCH with these changes applied. It is clear that this change in implementation only partially addresses the problem, with cache hit rates still scaling poorly as time-steps progress. The readers attention is however drawn to the marked decrease in L3 cache misses, which explains the substantial benefit to runtime scaling seen in Fig. 8 for the array version

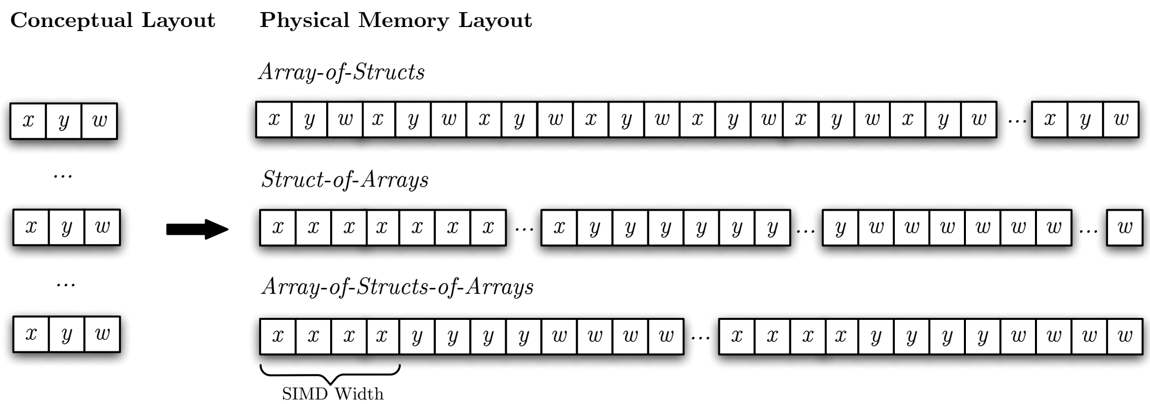

Fig. 7. A comparison of Array-of-Structs, Structs-of-Arrays, and Array-of-Structs-of-Arrays data layouts.

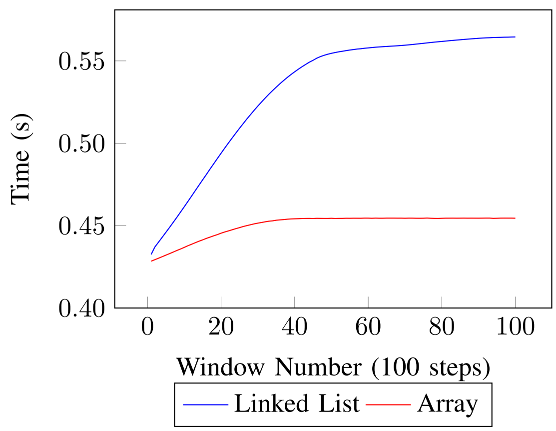

Fig. 8. EPOCH Time-step duration, during a simulation consisting of 100 simulation windows, each window containing 100 steps.

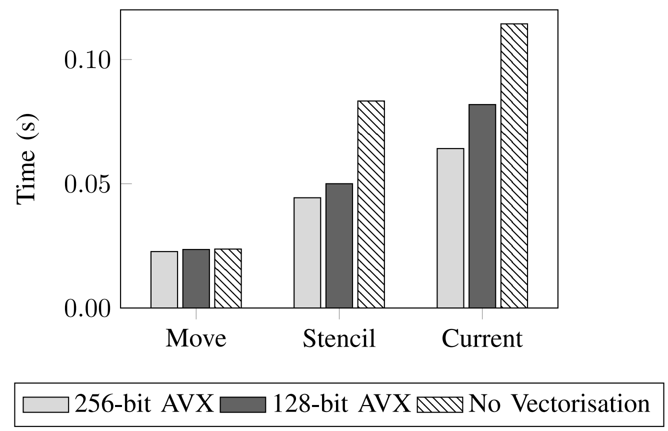

Fig. 9. SIMD scaling of miniEPOCH AoS kernels for 256-bit AVX, 128-bit AVX, and for the code operating without vectorisation.

of EPOCH. Whilst it is clear from Fig. 8 that an array version of EPOCH benefits the overall runtime, the cache miss figures in Fig. 5 identify that a secondary problem still exists. Further analysis of the cache miss rates directed our attention to the large stencil required when applying electromagnetic effects to the particle momenta. Increasing disorder in the particle list may explain the poor cache and memory behaviour, it would increase the probability of cache pressure, and the probability of spilling during this stencil-gather. Our strategy here was to periodically sort the particle store, and in so doing investigate the effect particle disorder had on both cache misses and runtime. Such a sort remedied the problem, and allowed for near perfect time-step scaling with a maximum deviation of 0.03% over ten thousand time-steps.

Having arrived at a significantly improved array-based version of miniEPOCH, efforts could then be focused on optimising EPOCH to ensure that it was better able to utilise current and future hardware. At the outset of this study, EPOCH was unable to exploit vector operations in its main kernel. The reason for this was a classic update dependency when accumulating currents, with multiple particles possibly having to write to the same array location concurrently. Given the large size of the initial kernel (approximately 500 lines of code), loop fission could be used to great effect in order to separate out much of the non-dependant computation which was SIMD-parallel safe. The previously mentioned sort, combined with iterating over particles on a per vertex basis, allows us to hoist loop invariant calculations so that they could be performed once per vertex. This offered a decrease in the overall required computation, and allowed the compiler to better predict memory access patterns into global memory.

Through code modifications first tested in the mini-app, we were able to ascertain that the restructured fissioned kernel was able to successfully exploit vector instructions. Fig. 9 shows the SIMD scaling of kernel runtime for 256-bit AVX (4 doubles), 128-bit AVX (2 doubles), and for the code operating without vectorisation. We see reasonable SIMD scaling for all kernels, except for *move*. This is because of its memory-stream-like structure, and the lack of floating point operations per second (FLOPS) to hide the memory accesses. The *stencil*

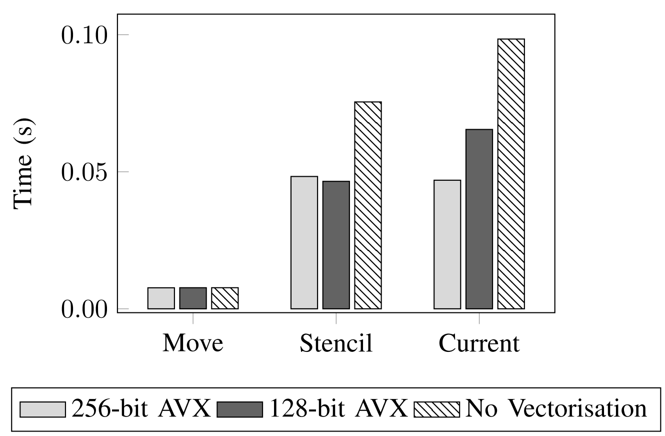

Fig. 10. SIMD scaling of miniEPOCH SoA kernels for 256-bit AVX, 128-bit AVX, and for the code operating without vectorisation.

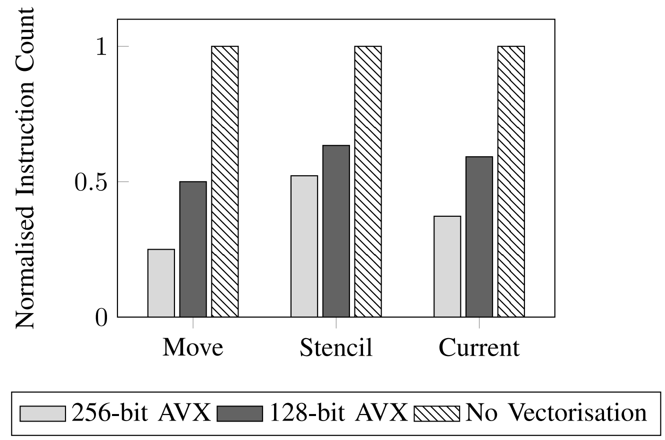

Fig. 11. Normalised instruction counts per kernel for different SIMD widths for the *Struct-of-Arrays (SoA)* memory layout.

and *current* kernels represent the majority of the time spent in miniEPOCH, and demonstrate significantly improved performance with SIMD width. Such improvements will yield further gains if the current trend of increasing SIMD width continues.

We can further improve performance by considering alternate memory layouts used for the array storage. Not only does this change how data is accessed, it also determines the number of concurrent memory streams the processor has to track during pre-fetching. Typically, particles are stored in an array of custom objects (or structs), a technique known as an *Array-of-Structs (AoS)*. By storing particles as an AoS, a single memory stream is required, with each particle loaded bringing with it all particle properties from main memory. This effect holds regardless of the number of properties used in the given kernel, and can represent a significant overhead for kernels which require few fields. When processing multiple array elements, as is typical in SIMD, loads must be gathered from memory, and any writes scattered, incurring a performance cost and increasing the latency of the memory operations. Alternative approaches include a *Struct-of-Arrays (SoA)* and a more complex hybrid, *Array-of-Structs-of-Arrays (AoSoA)* (which aims to combine the benefits of both SoA and AoS). A brief overview of these memory layouts is found in Fig. 7, where in our experiments *SIMD Width* was typically 4, as we were operating on doubles with 256-bit AVX.

For the SoA data layout, single particle properties for multiple particles are stored together in an array. This means that under SIMD operation, single properties from multiple particles can be loaded in one contiguous and aligned load, at the expense of tracking a different memory stream per property required. This eliminates any potential for gather/scatters, and is often favourable when only a few particle properties are required. With AoSoA, groups of N elements of each property are stored together, in order, where N is typically a function of vector length. This approach attempts to combine the benefits of both SoA and AoS, but comes at the expense of vastly increased complexity and an indexing overhead. In the current kernel configuration, a particle has 6 properties and stores 1 intermediary value (all types are `doubles`). The *move* kernel requires 5 such properties, the *stencil* kernel requires 6 and the *current* kernel requires 4. To investigate enhanced SIMD scaling, we ported the mini-app to each of the alternative memory layouts – this provides an excellent example of where mini-apps allow rapid code exploration, which might not otherwise be possible on full production codes. Fig. 10 shows the vector scaling of the SoA implementation, which compared to Fig. 9 shows that the performance is favourable in the kernels requiring fewer particle field accesses, and generally favourable overall. This result is largely due to the more effective use of data loaded using SoA, as no bandwidth is wasted. Fig. 11 shows the relative difference in instruction counts for varying SIMD widths, as recorded by PAPI. For good vector scaling one would expect to see the number of instructions executed decrease as a function of SIMD width. We see that both the `move` and `current` kernels scale as expected, but that the `stencil` kernel is only able to partially benefit from the increase in SIMD width.

Fig. 12 shows the overall performance as a result of each memory layout. Again we observe that as fewer particle properties are required, SoA performs better and, as more particle properties are required, so AoS outperforms SoA. AoSoA pays the cost of masked hardware instructions due to the sparse particle grouping used, a cost which will be much reduced in future hardware generations and has already been much reduced on the Intel Xeon Phi product range. We see that each kernel benefits differently from the change in array layout, largely due to cache pressure and required memory bandwidth.

As with all mini-app optimisation studies, our goal was to map our improvements back to the parent code in order to facilitate improved scientific investigation. Fig. 13 shows the overall runtime for an optimised version of EPOCH, as well as the a comparison against the original EPOCH code base and an array-based implementation. The array versions shows a small increase over the original implementation due to the lack of memory fragmentation and increased memory locality.

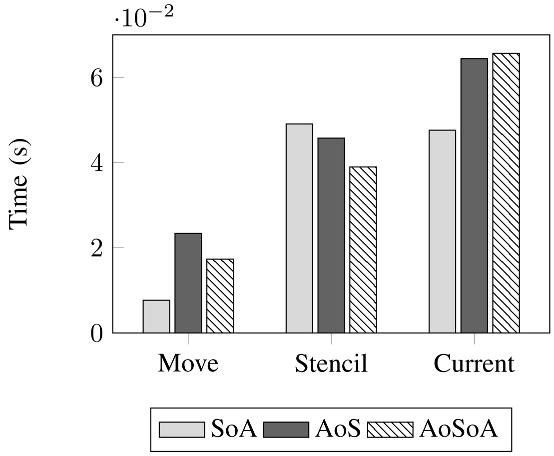

Fig. 12. Kernel runtime for different data layouts.

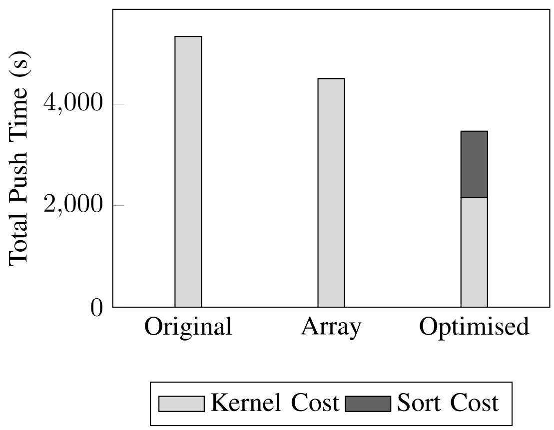

Fig. 13. Overall runtime for the original and optimised versions of EPOCH for a 10,000 step run. The additional *sort* cost is highlighted for clarity.

The optimised version of EPOCH represent a considerable improvement in code performance. It includes all previously discussed improvements, including spatial sort, increased vectorisation, loop-invariant code hoisting, array scalarisation, and coalesced temporary SIMD arrays. These optimisations deliver a notable improvement to production code runtimes, and successfully demonstrate the value a mini-app based optimisation investigation. These improvements, recorded on ARCHER, a 1.6 PFlop/s Cray XC30, demonstrate a 2.02× speed-up in the core EPOCH algorithm and a 1.55× speed-up to the overall application runtime.

VI. Conclusions and Future Work

Despite recent successes at the large laser-based inertial confinement fusion device at the National Ignition Facility at LLNL, we remain some distance from being able to create controlled, self-sustaining fusion reactions. Inertial confinement fusion (ICF) represents one leading design for the generation of energy by nuclear fusion and computing simulations supporting ICF continue on some of the world's most powerful supercomputers. The research presented here focuses on *EPOCH*, a fully relativistic particle-in-cell plasma physics code, developed by a leading network of over 30 UK researchers. A significant challenge in developing large codes like EPOCH is maintaining effective scientific delivery on successive generations of high-performance computing architecture. To support this process, we adopt the use of *mini-applications* – small code proxies that encapsulate important computational properties of their larger parent counterparts.

Through the development of *miniEPOCH*, we investigate known time-step scaling issues within EPOCH and explore possible optimisations. In particular, this work:

- Presents the development of a new mini-app (*miniEPOCH*) for the EPOCH code. We believe that this is the first mini-app explicitly targeting a finite difference time domain particle-in-cell (PIC) plasma physics code;
- Utilises *miniEPOCH* to explore known performance problems with EPOCH, and in particular (i) the increasing time-step duration during simulation runtime and, (ii) high levels of cache miss rates due to particle-store fragmentation;
- Employs the mini-app to explore opportunities for code optimisation, including the utilisation of shared memory, exploiting increasing vector width and improving memory locality;
- Validates these findings on ARCHER, a 1.6 PFlop/s Cray XC30, housed at the UK national supercomputing centre at EPCC. These improvements demonstrate a 2.02× improvement to the core EPOCH algorithm and a 1.55× speed-up to the overall application runtime.

In future work we hope to reduce the performance impact of the additional particle sort; highlighted in Fig. 13. Whilst the current implementation of the sort remains largely unoptimised, a algorithmic change would likely yield significant improvement. The current implementation performs a full particle sort despite large portions of the data remaining sorted. A sorting algorithm which exploits this feature could greatly improve overall application performance. Additionally, a trade off between sort-cost and the performance improvement could be achieved, by tracking sorted data regions and sorting periodically, rather than requiring a fully ordered sort every timestep. Unfortunately such investigation fell outside of the scope of this work and is thus reserved for future work

Finally, as part of this work an OpenMP port of EPOCH has also been developed. The results of this port demonstrate good on-node scaling, and future work will build on this codebase to develop an Intel Xeon Phi code version. This version of miniEPOCH would be able to utilise the increased SIMD width offered by the Intel Xeon Phi, and could facilitate a study assessing the viability of heterogeneous and accelerated hardware platforms for PIC codes.

Acknowledgments

This research is supported in part by the EPSRC grant "A Radiation Hydrodynamic ALE Code for Laser Fusion Energy" (EP/I029117/1), by The Royal Society, through their Industry Fellowship Scheme (IF090020/AM) and by AWE through the Warwick-hosted Centre for Computational Plasma Physics. Use of ARCHER was supported by several Resource Allocation Panel (RAP) awards, including "High-level Abstractions for Performance, Portability and Continuity of Scientific Software on Future Computing Systems". Professor Stephen A. Jarvis is an AWE William Penney Fellow.

References

[1] T. D. Arber, K. Bennett, C. S. Brady, M. G. Ramsaym, N. J. Sircombe, P. Gillies, R. G. Evans, H. Schmitz, A. R. Benn, and C. P. Ridgers, "Contemporary particle-in-cell approach to laser-plasma modelling," *Plasma Physics and Controlled Fusion*, 2015.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[3] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," *Sandia National Laboratories, Techical Report SAND2009-5574*, 2009.

[4] Ruhl, H, "Classical Particle Simulations with the PSC Code," http://www.physik.uni-muenchen.de/lehre/vorlesungen/wise_09_10/tvi_mas_compphys/vorlesung/Lecturescript.pdf.

[5] H. Nakashima, Y. Miyake, H. Usui, and Y. Omura, "OhHelp: A Scalable Domain-Decomposing Dynamic Load Balancing for Particle-in-Cell Simulations," in *Proceedings of the 23rd International Conference on Supercomputing*. ACM, 2009, pp. 90–99.

[6] K. S. Yee *et al.*, "Numerical solution of initial boundary value problems involving Maxwells equations in isotropic media," *IEEE Trans. Antennas Propag*, vol. 14, no. 3, pp. 302–307, 1966.

[7] R. F. Bird, S. J. Pennycook, S. A. Wright, and S. A. Jarvis, "Towards a Portable and Future-proof Particle-in-Cell Plasma Physics Code," *International Workshop on OpenCL*, 2014.

[8] I. Karlin, J. McGraw, J. Keasler, and B. Still, "Tuning the LULESH Mini-app for Current and Future Hardware," in *Nuclear Explosive Code Development Conference Proceedings (NECDC12)*, 2012.

[9] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 919–932.

[10] H. C. Problem, "Technical Report LLNL-TR-490254," *Lawrence Livermore National Laboratory*.

[11] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney *et al.*, "LULESH Programming Model and Performance Ports Overview," *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep*, 2012.

[12] A. Nichols, "Users manual for ALE3D: An arbitrary Lagrange/Eulerian 3D code system," *Lawrence Livermore National Laboratory*, 2007.

[13] P.-F. Lavallée, G. C. de Verdière, P. Wautelet, D. Lecas, and J.-M. Dupays, "Porting and optimizing HYDRO to new platforms and programming paradigms–lessons learnt."

[14] R. Teyssier, "Cosmological Hydrodynamics with Adaptive Mesh Refinement: a new high resolution code called RAMSES," *Astronomy & Astrophysics*, vol. 385, no. 1, pp. 337–364, 2002.

[15] C. K. Birdsall and A. B. Langdon, *Plasma physics via computer simulation*. CRC Press, 2014.

[16] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas (1994-present)*, vol. 15, no. 5, p. 055703, 2008.

[17] H. Burau *et al.*, "PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster," *IEEE Transactions on Plasma Science*, vol. 38, no. 10, pp. 2831–2839, 2010.

[18] F. Rossi, P. Londrillo, A. Sgattoni, S. Sinigardi, and G. Turchetti, "Robust Algorithms for Current Deposition and Dynamic Load-balancing in a GPU Particle-in-Cell Code," in *ADVANCED ACCELERATOR CONCEPTS: 15th Advanced Accelerator Concepts Workshop*, vol. 1507, no. 1. AIP Publishing, 2012, pp. 184–192.

[19] J. Payne, D. Knoll, A. McPherson, W. Taitano, L. Chacon, G. Chen, and S. Pakin, "Design and development of a multi-architecture, fully implicit, charge and energy conserving particle-in-cell framework," *Bulletin of the American Physical Society*, vol. 58, 2013.

[20] G. Chen, L. Chacón, and D. C. Barnes, "An efficient mixed-precision, hybrid CPU-GPU implementation of a nonlinearly implicit one-dimensional particle-in-cell algorithm," *Journal of Computational Physics*, vol. 231, no. 16, pp. 5374–5388, 2012.

[21] Z. Lin, T. S. Hahm, W. Lee, W. M. Tang, and R. B. White, "Turbulent Transport Reduction by Zonal Flows: Massively Parallel Simulations," *Science*, vol. 281, no. 5384, pp. 1835–1837, 1998.

[22] D. W. Barnette, S. D. Hammond, J. H. Laros, and J. Jayaraj, *Using Miniapplications in a Mantevo Framework for Optimizing Sandia's SPARC CFD Code on Multi-Core Many-Core and GPU-Accelerated Compute Platforms*, Dec 2012.

[23] "UK Mini-App Consortium (UKMAC)," http://uk-mac.github.io.