Deakin, T., Gaudin, W., & McIntosh-Smith, S. (2017). On the mitigation of cache hostile memory access patterns on many-core CPU architectures. In *High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Revised Selected Papers* (pp. 348-362). (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 10524 LNCS). Springer. https://doi.org/10.1007/978-3-319-67630-2_26

Peer reviewed version

Link to published version (if available):
10.1007/978-3-319-67630-2_26

On the Mitigation of Cache Hostile Memory Access Patterns on Many-core CPU Architectures

Tom Deakin[1], Wayne Gaudin[2], and Simon McIntosh-Smith[1]

[1] Department of Computer Science, University of Bristol, Bristol, UK
tom.deakin@bristol.ac.uk
[2] UK Atomic Weapons Establishment, Aldermaston, UK

Abstract. Kernels with low arithmetic intensity with memory footprint exceeding cache sizes are typically categorised as memory bandwidth bound. Kernels of this class are typically limited by hardware memory bandwidth. In this work we contribute a simple memory access pattern, derived from a widely-used upwinded stencil-style benchmark, which presents significant challenges for cache-based architectures. The problem appears to grow worse as CPU core counts increase, and the pattern in its initial form shows no benefit from the new high-bandwidth memory now appearing on the Intel Xeon Phi (Knights Landing) family of processors. We describe the memory access scenarios which appear to be causing lower than expected cache performance, before presenting optimisations to mitigate the problem. These optimisations result in useful effective memory bandwidth and runtime improvements by up to 4X on cache based architectures. Results are presented on the Intel Xeon (Broadwell) and Xeon Phi (Knights Landing) processors.

1 Introduction

For kernels (computational routines) with low arithmetic intensity, the Roofline model typically shows that memory bandwidth becomes the performance limiter [9]. Examples of such kernels can be seen in the STREAM [8] and GPU-STREAM [3] benchmarks, where in the later we have explored the achievable memory bandwidth of a highly diverse range of computer architectures. However for some memory bandwidth bound codes, an increase in the available memory bandwidth does not necessarily yield a proportionate improvement in performance, as the performance of such an application may depend on the degree to which it has been optimised — specifically, the degree to which its implementation is *actually* bandwidth bound (as opposed to *theoretically* bandwidth bound). Once such application is the SNAP performance proxy for modern deterministic transport codes from Los Alamos National Laboratory [10, 5].

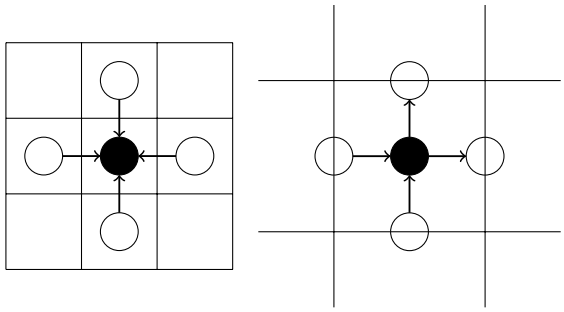
We have previously optimised SNAP to perform well on GPUs [2] and explored its scaling characteristics on large supercomputers [1]. GPU architectures are typically optimised for greater memory bandwidth relative to traditional CPU architectures. For SNAP we are able to demonstrate significant performance improvements, with the time to solution halved using NVIDIA K20X

GPUs compared to Intel Xeon (Haswell) CPUs at scale. The Intel Xeon Phi (Knights Landing) processor also offers a memory bandwidth increase relative to traditional multi-core CPUs, due to its on-package MCDRAM. For a bandwidth bound code therefore, assuming efficient vectorisation and memory access pattern optimisations have already been applied (as is the case with SNAP), running on the Knights Landing architecture utilising MCDRAM should provide a performance improvement utilising the extra memory bandwidth available. However, initial results for SNAP show that one Knights Landing achieves comparable performance to dual-socket Intel Xeon processors, which have around a quarter of the memory bandwidth relative to Knights Landing's MCDRAM [5]. We had also previously observed that the performance on Intel Xeon Phi (Knights Corner) co-processors was low, attaining only a small fraction of achievable memory bandwidth [2]. It is therefore the case that the SNAP application is not actually memory bandwidth bound on the Xeon Phi architecture; instead, some other bound is in place. There is however little tangible or actionable evidence as to what the limiting factor is (for the SNAP mini-app); the profiling tools suggest a good ratio of cycles per instruction, good vectorisation efficiency and stride-one data access patterns. As such, a 'glass ceiling' exists beneath the memory bandwidth limit in the Roofline model for this kernel. In this paper we present some intuition around the reasons for this limit to exist in this style of application; namely an upwind stencil. We then provide some solutions allowing a benchmark application to break through the ceiling and reach the memory bandwidth bounds as predicted under the Roofline model; this also comes with a significant improvement in the runtime of the benchmark.

2 Stencil Patterns

A stencil access pattern describes the neighbouring data requirements for the solution of each cell in the mesh. For a typical 5-point stencil, each cell (i, j) requires values from cells $(i \pm 1, j \pm 1)$, and for a structured mesh this can be visualised as in Fig. 1a. The values in the centre of each cell are used in the solution of the cell centred value of the cell in question, a pattern typical of many computational fluid dynamic codes, such as in the Lattice-Boltzmann method.

The 5-point stencil however is applied differently in other fields, and we consider a particularly interesting and important variant in this paper. At first glance this application looks very similar to a standard 5-point stencil, with the cell centred solution of a particular cell using values from neighbouring cells. However the values from neighbouring cells are *edge* centred rather than cell centred as shown in Fig. 1b. The edge centred solutions are calculated using a simple finite difference relation given the cell centred value. Also, only data from half the edges are used for the solution, with the other edge solutions shared to downwind neighbouring cells; note the change in direction of the arrows in the figure. Specifically, for the sweep direction shown and an origin at the bottom left of the diagram, cell (i, j) requires values originally calculated in cells $(i-1, j)$



(a) Structured mesh (b) Upwinded sweep

Fig. 1: Applications of a typical 5-point stencil

and $(i, j-1)$, and the edge centred solutions in cell (i, j) are required by cells $(i+1, j)$ and $(i, j+1)$.

The data dependency is formed by *upwinding* whilst discretizing a partial differential equation in the spatial domain. It results in a *wavefront sweep*, an important programming pattern which appears in a variety of applications such as dynamic programming, LU factorisation and deterministic transport [7, 10].

The available parallelism is also different compared to a standard application of the stencil. The cells must now be computed in the order defined by the sweep; this is in contrast to the mesh in Fig. 1a where the cells can be computed concurrently as long as a copy of cell centred values is stored. Solution on a large distributed system also uses a standard halo exchange communication pattern for the standard approach, whereas outgoing edge data is sent to downwind neighbours as it becomes available here. The focus of this paper is the on node performance and so communication differences will not be discussed further.

Because the edge centred values are temporary, and only half are required to share between neighbouring cells, it is typical to optimise the memory footprint and store the incoming and outgoing edge values in the same memory location. This reduces the memory footprint and also encourages reuse of array elements.

The descriptions in the 2D case are analogous in three dimensions using a seven-point stencil, and it is the 3D case that we investigate in this paper.

3 Memory Access Patterns

The generalised memory access pattern of the upwinded stencil may be described as follows. Multiple values are calculated per cell, which are operated on in parallel through vectorisation, and are hence stored contiguously as the inner most dimension within the mesh array to allow for simple stride 1 memory access patterns.

The cell centred solution computation is somewhat similar to a STREAM Triad operation [8] and is calculated based on its previous value along with

weighted contributions from the upwind neighbour cells' edge centred solutions. Simple floating point arithmetic (typically fused multiply add) is used to combine the terms. This computation is clearly memory bandwidth bound as the computational intensity is very low, with a ratio of arithmetic to memory locations accessed of close to 1.0. Specifically the computational intensity in double precision, with eight bytes per element, is 1/8. Subsequent calculation of the edge centred values is also similar to STREAM Triad, and is just a simple finite difference operation requiring a single fused multiply add. Finally, a within cell SIMD reduction of the cell centred values is calculated and stored for each cell.

Unlike STREAM operations, the arrays containing the data are of different sizes. The cell centred values are stored in a mesh sized array and are therefore large in size. The weights used are in very small arrays, and there is one weight for each cell centred value which are the same in every cell and are therefore shared and independent of the mesh size. The arrays containing the edge centred values are allocated on Cartesian planes in the mesh, and are therefore smaller than the total mesh size, existing in one less dimension than the mesh; this is a memory footprint optimisation as incoming values are overwritten in this array by the outgoing equivalent. The threads operate on disparate slices of the arrays via a final outer dimension so as to avoid false sharing. All data access is stride 1.

In summary, there are three operations we consider within the kernel:

1. Calculate cell centred values with low computational intensity, streaming through the large array.
2. Calculate outgoing edge centred solutions.
3. Reduce cell centred values within each cell.

4 Cache Based Architectures

The memory hierarchy of CPU style architectures typically consists of a set of data caches between the execution units and the main memory. Using the Intel Xeon (Broadwell) CPU as an exemplar, its hierarchy consists of three levels of cache and main DDR memory. Level 1 (L1) and Level 2 (L2) caches are available to each core independently and are 32 KB (for data) and 256 KB in size respectively. Level 3 (L3) cache is 55 MB for the high end E5-2699 v4 model, and is shared between all cores on the socket.

The Intel Xeon Phi (Knights Landing) Processor has a different memory hierarchy, although there are many similarities. Level 1 caches are available per core with a capacity of 32 KB. A 1 MB L2 cache is shared between pairs of cores, which are organised on tiles, with the entire device consisting of some number of tiles. High bandwidth MCDRAM and standard DDR are also available, and their position in the hierarchy depends on the mode the processor is booted into. The MCDRAM is available either as a directly mapped cache for DDR, or else as a separate memory space; it is this latter mode that we consider here where all data is resident in the high bandwidth memory.

For both processors, when a load instruction is encountered the cache hierarchy is always checked. Data not present in the caches is moved down through

the hierarchy, with data being evicted according to the cache policy. For a store instruction, both processors operate a "read for ownership" policy, whereby the data is first loaded into cache before being written. Both processors also support non-temporal store instructions which bypasses this mechanism, writing directly to the highest level in the memory hierarchy (MCDRAM or DDR).

Hardware (and software) prefetchers also operate in the background to assist the movement of memory through the cache hierarchy. If a data access pattern is detected, the prefetcher will move data down the hierarchy in advance of when the memory is predicted to be used. Therefore the required data will hopefully be in a fast, low level cache in time and hence increase the throughput of the processor as it need not stall waiting for the data to arrive.

5 Investigation into Memory Bandwidth Issues

In order to practically investigate the performance issues of this style of stencil operation a small benchmark code was written. This code, named *mega-stream*, was distilled from the SNAP mini-app which encounters the previously described performance issues on cache based architectures. The simple computational kernel is shown in Listing 1. We display the Fortran version in print as it allows for compact representation of the multi-dimensional array accesses; the C version is similar and we ensure that the memory layout matches that of the loop nest in both cases to ensure stride 1 access for the innermost loop. The kernel consists of five nested loops with the three operations from Sec. 3; the cell centred computation on line 6, the outgoing edge centred solutions on lines 7–9, and the reduction over the innermost dimension on line 10. OpenMP work-sharing threads are employed on the outermost loop (`Nm`) and compiler auto-vectorisation via the OpenMP `simd` clause is employed on the innermost loop (`Ni`).

The data reuse of the `x`, `y` and `z` arrays are of some interest. Note that each of these arrays is missing one of the three middle indices; `x` is missing the `l` index, `y` is missing the `k` index, and `z` is missing the `j` index. All three arrays have the innermost index `i`. Using the `z` array as an example, the subsection of the array used is the same for all `j`, and as such one would hope that for a given `k` and `l` the associated `Ni` values remain in cache for the duration of the `Nj` loop. Note that none of the writes can be hoisted above loops as the updated values of one iteration are used in line 6. This pattern can be visualised with Fig. 1b whereby this array carries data in the j-axis (in jkl-space) between adjacent cells. There is no reuse of the `r` array.

A model for the memory bandwidth of the routine in Listing 1 can be constructed by simply counting the number of bytes moved under some basic assumptions. We assume that a write counts as a single memory movement; in particular, read for ownership is not required. Indeed, this is an architecture specific design decision and in theory the computation does not require this. We also assume that once we have read a memory location, further reads and writes are free. Specifically, once we read a location in the `x`, `y` or `z` arrays, these are cached and the update is free. It is typical that such assumptions are made

```
1    DO m = 1, Nm
2      DO l = 1, Nl
3        DO k = 1, Nk
4          DO j = 1, Nj
5            DO i = 1, Ni
6              r(i,j,k,l,m) = q(i,j,k,l,m) + a(i)*x(i,j,k,m) +
   ↪  b(i)*y(i,j,l,m) + c(i)*z(i,k,l,m)
7              x(i,j,k,m) = 0.2*r(i,j,k,l,m) - x(i,j,k,m)
8              y(i,j,l,m) = 0.2*r(i,j,k,l,m) - y(i,j,l,m)
9              z(i,k,l,m) = 0.2*r(i,j,k,l,m) - z(i,k,l,m)
10             total(j,k,l,m) = total(j,k,l,m) + r(i,j,k,l,m)
11           END DO
12         END DO
13       END DO
14     END DO
15   END DO
```

Listing 1: The mega-stream kernel

on memory bandwidth models since they represent the best-case behaviour and form an upper-bound for performance. The model is therefore the total of all reads and writes under these assumptions: `r` is written to once per element, `q` is read once per element, `x`, `y` and `z` are read and written once per element (assuming future updates are free), `a`, `b` and `c` are read once per element, and `total` is read and written once per element. The data are double precision floating point elements which are of size 8 bytes. Therefore the estimated (modelled) amount of data moved is:

$$\begin{aligned}(N_i * N_j * N_k * N_l * N_m + N_i * N_j * N_k * N_l * N_m + \\ 2 * N_i * N_j * N_k * N_m + 2 * N_i * N_j * N_l * N_m + 2 * N_i * N_k * N_l * N_m + \\ N_i + N_i + N_i + 2 * N_j * N_k * N_l * N_m) * 8 \text{ bytes}\end{aligned} \quad (1)$$

The estimated memory bandwidth is therefore the data moved divided by the runtime of the kernel. The benchmark runs the kernel 100 times and takes the minimum kernel time to calculate the bandwidth.

5.1 List of Experimental Platforms

The Intel Xeon Phi (Knights Landing) used for our experiments is a 7210 64-core at 1.30 GHz. The processor was configured in Flat/Quadrant mode, and has 16 GB MCDRAM with 96 GB DDR (unused). The mesh is clocked at 1.6 GHz resulting in a rate of 6.4 GT/s for MCDRAM. Transparent huge pages were enabled on the system. The code was compiled with the Intel Compiler 17 update 2 specifying the `-O3 -xMIC-AVX512` flags. We ran from MCDRAM using the `numactl` tool with one OpenMP thread per physical core, pinned using

the `OMP_PROC_BIND` environmental variable. The STREAM Triad benchmark achieves 448 GB/s on this system.

We also use an Intel Xeon E5-2699 v4 (Broadwell) 22-core dual-socket node from a Cray XC40 supercomputer. This processor is clocked at 2.2 GHz and has 128 GB DDR. The code was compiled with the Intel Compiler 17 update 1 specifying the `-O3 -xCORE-AVX2` flags. We ran with one OpenMP thread per physical core, pinned using the `OMP_PROC_BIND` environmental variable and the `aprun` command. The STREAM Triad benchmark achieves 128 GB/s on this system.

The default problem size sets `Ni`=128, `Nj`=`Nk`=`Nl`=16, and `Nm`=64. The `q` and `r` arrays are therefore sized 256 MiB, `x`, `y` and `z` are 16 MiB, and `a`, `b` and `c` are 1 KiB each. The model predicts moving 612 MiB to/from main memory for each kernel execution for the default problem size.

5.2 Baseline Performance

Throughout this investigation we will quote results for the default inputs unless specified otherwise. The initial estimated memory bandwidth as a percentage of STREAM Triad is shown in Fig 2 labelled "Baseline". The performance on Knights Landing, or lack thereof, is rather striking and certainly highlights the need for an investigation; note that this kernel, which has stride 1 access patterns very reminiscent of STREAM, only achieves 16.4% of STREAM bandwidth (74 GB/s) when running solely out of the MCDRAM. On Xeon the achieved memory bandwidth is, at 65.1% of STREAM, perhaps on the low side for a stride 1 access code. While this is not necessarily low enough to cause concern, it is clear there is scope for improvement on CPUs too. The important corollary is that these observations are similar to the measurements we make with the SNAP application itself. Note too that the runtime of the kernel is similar on both architectures, whereas if it was memory bandwidth bound the advantages of MCDRAM on the Knights Landing should result in a speedup of around 3.5x (the ratio of their achieved bandwidth on STREAM).

We aligned all the arrays to 2 MB page boundaries (matching the page size of Knights Landing) to minimise any latency of unaligned loads and stores. This is a common optimisation step for memory bandwidth bound codes when examining vectorisation, and was performed as part of due diligence in the early stages of development. The alignment is performed at allocation via the C11 `aligned_alloc` library call. Alignment in this fashion also means that peel loops are not required (even though they would have been generated by the compiler).

Whilst the bandwidth reported by the application is estimated, it is possible to compare to a measured memory bandwidth obtained through hardware counters via a tool such as Intel vTune Amplifier XE. For the Knights Landing run, the tool was reporting near peak memory bandwidth use to MCDRAM, indicating that more memory must be moving than our model predicts, and as such this movement is considered wasteful by the model. This observation also hints at the underlying problem that is resulting in lower effective bandwidth. The Roofline analysis in the Intel Advisor tool initially shows that the innermost

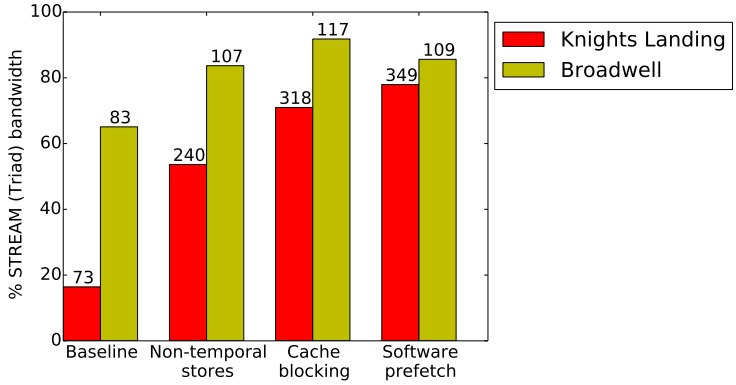


Fig. 2: Estimated memory bandwidth of the default problem size shown as a percentage of STREAM (Triad) memory bandwidth on Xeon and Xeon Phi as optimisations are applied (inclusively). Achieved memory bandwidth numbers are shown above each bar.

`Ni` loop lies on the MCDRAM bandwidth line, and this could be interpreted that the application is indeed memory bandwidth bound as expected, however in this paper we should optimisations which improve the runtime by 4X. However, analysis of the entire kernel as a whole is not shown in current versions of this tool and so inference from the analysis should be used with care.

Figure 3 shows the estimated memory bandwidth of the baseline code with dashed lines for a variety of problem sizes, and explores the ranges of the different loops, leaving the others fixed at the default size. The "inner" line varies the `Ni` range, the "middle" varies the `Nj`, `Nk` and `Nl` ranges identically, and the "outer" line varies the `Nm` range. As such the three dashed lines represent an exploration of the problem space spanned by the set of nested loops. For example, setting the middle loops to 8 yields the following configuration: `Ni`=128, `Nj`=`Nk`=`Nl`=8, `Nm`=64. Varying the number of iterations each thread performs by increasing `Nm` alone shows little change in the achieved bandwidth from the default case, as shown by the outer dashed red line. The other dashed lines show the loop extents of the four innermost loops have a more dramatic effect. It is clear in the figure that there is a large variation in bandwidth, up to 145% (excluding the first data point of the "middle" dashed green line), with bandwidths of 49.9–122.3 GB/s depending on the input size. Where the iteration space of the middle jkl loops are set to 4 (leftmost point of the "middle" dashed green line) the baseline performance for this input exceeds the memory bandwidth available from MCDRAM, and as such must already be taking advantage of the cache hierarchy; indeed, the total problem is only 179 KiB per core, so can be fully resident in L2 cache.

It is usual to run the STREAM benchmark with one thread per physical core, and the results presented throughout the paper for the mega-stream benchmark

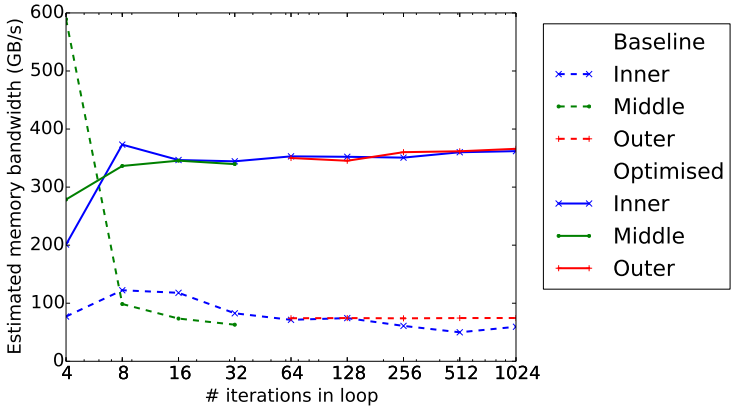


Fig. 3: Estimated bandwidth for a range of inputs by varying one dimension. The dashed lines show the baseline performance with the solid lines showing the performance the optimisations applied.

also assume one thread per physical core. However, higher memory bandwidth is achieved for the baseline code by using the hyperthreads on Knights Landing and increasing `Nm` to match the number of hyperthreads, so that each thread performs the same amount of work. Using 2 and 4 hyperthreads per physical core increases the bandwidth to 138 and 114 GB/s respectively, an increase from the initial 73 GB/s attained from 1 thread per physical core. These results are still much lower than the expected memory bandwidth limits of this processor, however using the hyperthreads for the baseline may allow for some memory latency hiding thus increasing the estimated memory bandwidth. However, we have had to increase the problem size in order to take advantage of the hyperthreads which may not be applicable to all applications.

5.3 Improving the Performance

In order to improve the performance, we seek to minimise the wasteful data movement observed above and ensure that data is only moved the minimum number of times, something which is captured by our model. We take an incremental approach of three steps:

1. Ensure data which *is not* re-used *is not* in cache.
2. Ensure data which *is* re-used *is* in cache.
3. Ensure data is in the cache in time for use.

Whilst these optimisations sound obvious it is critical to observe that the code itself already has "good" memory access patterns; it is stride 1 access, should be very predictable for a cache lookahead engine, and we have confirmed that the code is vectorised well by the compiler. Indeed, the STREAM kernels are typically optimised though the first and last of these steps, however there

is no data reuse. Note too that the STREAM benchmark kernels require little user intervention as the compiler performs this optimisations on our behalf; for mega-stream the programmer had to intervene. There are few mechanisms for explicitly controlling what is present in a CPU's cache hierarchy, as the memory subsystem is managed automatically by the processor itself, primarily based on data locality. As such, controlling what data is in the cache is more a result of side effects of the instruction stream rather than from any explicit description of the memory location. This is somewhat similar to programs being NUMA-aware in the sense that they, for example, ensure data is allocated and used on a core within a NUMA region, however this is achieved without any form of annotation or mechanism to explicitly state that this was the intended effect. GPU architectures tend to be very different in this regard, as they provide a scratchpad memory whose contents are explicitly controlled by the programmer.

Non-temporal Stores The `r` array is large in size, there is no data-reuse within the kernel and it is only written to, so the previous value is not required. Therefore there is little use in the `r` array consuming cache space, or worse, evicting data that would benefit from caching.

The Streaming SIMD Extensions (SSE) instruction set introduced the notion of non-temporal stores with the `MOVNTQ` instruction. This instruction hinted that the cache hierarchy should be avoided and the data should be stored directly in main memory [4]. Additionally this avoids the "read for ownership" policy and it is no longer necessary to read `r` before writing to it, thus saving the unnecessary read of this entire array; previously every element was read before being written and therefore writing to this array caused 512 MiB of data movement (for the default problem size) instead of 256 MiB (its total size).

We can encourage the Intel compiler to generate such instructions for the target architecture via compiler directives, specifically by decorating the inner loop with the directive `#pragma vector nontemporal(r)`. Note however that this is not a portable solution as this is a directive specific to the Intel compiler; at the time of writing the authors have been unable to find similar directives for other compiler vendors. Intel architectures require non-temporal instructions to occur on aligned memory locations. The improvement is shown in Fig. 2, where the achieved memory bandwidth is much better, with 3X faster runtime than the baseline on Knights Landing, a significant improvement. However, the percentage of STREAM bandwidth that mega-stream achieves on Knights Landing is still relatively low, and therefore further improvements are required.

On Broadwell we see a comparatively small 1.3X improvement in runtime from non-temporal stores. We interpret this as the larger caches per core mitigating the effects on Broadwell relative to Knights Landing. Broadwell's 55 MB L3 cache is shared between all 22 cores on a socket, which results in around 2.5 MB of L3 cache per core; significantly more in this last level of cache than the 512 KB per core in last level (L2) cache on Knights Landing. As such, the effects of reducing the cache pollution from the `r` array are less pronounced. Indeed, the CrayPat profiler reports that, on Broadwell, the baseline is achieving 76.1%

L1 cache hits, with non-temporal stores increasing this to 85.6%. However, the L2 cache hit rate has reduced; on Broadwell the baseline achieved 7.3% misses, whereas adding the non-temporal directive increased this to 40.1%. This increase is likely down to highlighting cache miss behaviour of other data arrays rather than obtaining a high hit rate for the `r` array as a side effect of cache pollution.

Cache Blocking There is reuse of the `x`, `y` and `z` arrays, although the reuse pattern is somewhat complex for the middle three jkl loops. We want to ensure that data remains in cache whilst there is some temporal locality of the elements of these arrays. For the default problem size on Knights Landing, each core is accessing a 256 KiB contiguous portion of each of these three arrays. Each pair of cores share a 1 MB L2 cache, and assuming there is no sharing of data each core has approximately 512KB of L2 cache available. The combined total of the local portion of these arrays (768 KiB) therefore exceeds the capacity of its available L2 cache, and therefore the data will at some point be evicted from this level of cache; as there is no L3 cache on Knights Landing the data will fall back to MCDRAM. Therefore the data will need to be read from main memory multiple times, whereas our bandwidth estimate assumes that the data remains resident in cache, as it should, and as the programmer might expect, due to the temporal locality and predictable access patterns.

As discussed above, there are no explicit mechanisms for controlling what is in a CPU cache, and therefore we must use other techniques to ensure that only the appropriate data remains resident in the cache. We therefore implement a *cache blocking* scheme, alternatively known as tiling, with the aim of decreasing the amount of data required in cache at any given time. By controlling which tiles are in use at any one time, we can also prevent cacheable data evicting other data that we want to retain in the caches. In many applications this is done by tiling the spatial dimension in one or more dimensions, however in this benchmark we have multiple values per spatial cell and so in contrast we tile in this extra dimension. We shall use one core's portion of the `x` array as an example; by default this is 256 KiB in size. Each `x(:,j,k)` element is accessed `Nl` times, once for each iteration of the `l` loop. By breaking the first index of the array into blocks, where each block is the size of a cache line, we can reduce the amount of memory kept in cache for the duration of the `l` loop; we can then ensure that all of these accesses are made from cache. An extra loop over the blocks is inserted between lines 1 and 2 in Listing 1; we also modify the inner loop in line 5 to index within a block. The arrays are allocated and initialised with an extra dimension, again keeping the order of the extents matching the loop nesting. By splitting the `Ni` dimension into blocks of eight, which corresponds to eight double precision elements forming a 64 byte cache line, the portion of the `x` array to be kept in cache for reuse drops from 256 KiB to just 16 KiB. The reduction in size is the same for the `y` and `z` arrays, and therefore the portion of *all* three arrays which is reused totals 48 KiB, which can certainly remain inside the 512 KB cache. The performance improvements from blocking are shown in Fig. 2, where the optimisations are applied inclusively.

On Knights Landing this is a good improvement in memory bandwidth and a 1.3X runtime improvement over applying non-temporal stores alone; achieving 71.0% of STREAM bandwidth (318 GB/s) once we apply both cache blocking and non-temporal stores, compared to 53.6% for the non-temporal stores alone. The variation in performance between different problem sizes has also vanished, and all inputs now achieve similar results (not shown for brevity).

The Broadwell architecture again improves a little with this optimisation, but not significantly; the fraction of STREAM bandwidth we have achieved increases from 83.7% to 91.8%. We believe this is down to the large caches holding the `x`, `y`, `z` arrays entirely in L3 cache and avoiding going to main memory entirely, but improvements can be seen at lower levels of the cache hierarchy. Profiler output from the CrayPat tool shows that L1 cache misses have fallen from 14.4% to 7.1%, a significant saving. Additionally L2 cache misses have reduced from 40.1% to 15.5%. What is clear though, is that the cache hit rates are generally high on Broadwell throughout these optimisation stages, and as such the impact on runtime is minimal. The Intel vTune profiler was not available on the platform so we are unable to provide L3 cache statistics.

This cache blocking technique applied in isolation does improve the performance of the baseline by around 1.7X on Knights Landing, however 4X over the baseline is demonstrated with both non-temporal stores and cache blocking combined.

Software Prefetching Finally a small improvement is available by ensuring that the prefetching of data is suitably early to hide the associated latency of memory movement. On profiling the cache blocked version, it can be seen that there are L2 cache misses for the read of the `q` array, indicating that the data is not there in time to be read when it is required. Therefore we can use Intel software prefetch intrinsics to generate prefetch instructions earlier in the instruction stream. We found that prefetching with a depth of 32 vector instructions was sufficient, and the intrinsic was inserted inside the `j` loop; software prefetching typically requires some experimentation in determining a suitable prefetch distance. This experimentation was done by firstly enabling automatic software prefetching in the compiler via the `-qopt-prefetch=3` flag, with the distance reported by the compiler used as the starting value for the prefetch distance. This results in a 10% boost in performance on Knights Landing, taking us up to 77.9% of STREAM bandwidth (349 GB/s).

Interestingly on the Broadwell architecture the use of the same software prefetch actually reduces the performance, from 91.8% to 85.6%. We could not find a suitable value for the prefetch distance which did not reduce performance from that achieved via cache blocking and non-temporal stores alone. Using software prefetchers may cause conflicts with the compiler automatically inserting these instructions, however no improvements could be found by turning off compiler prefetching. Note too that hardware prefetching will not function on the data stream if manual prefetching instructions are issued; therefore on the Broadwell architecture the hardware prefetchers alone seem sufficient for this

benchmark; after all the benchmark was achieving over 90% of STREAM Triad bandwidth after the cache blocking optimisation.

Summary Listing 2 shows the code changes described above applied to the kernel originally shown in Listing 1. Again we show the Fortran kernel for brevity; note the inclusion of the compiler directive for non-temporal stores, the software prefetch intrinsic, and the additional loop and corresponding index.

```
1   DO m = 1, Nm
2     DO n = 1, Ni/8
3       DO l = 1, Nl
4         DO k = 1, Nk
5           DO j = 1, Nj
6             CALL MM_PREFETCH(q(1+32*8,j,k,l,n,m), 1)
7             !DIR$ VECTOR NONTEMPORAL(r)
8             DO i = 1, 8
9               r(i,j,k,l,n,m) = q(i,j,k,l,n,m) + a(i,h)*x(i,j,k,n,m) +
    ↪  b(i,n)*y(i,j,l,n,m) + c(i,n)*z(i,k,l,n,m)
10              x(i,j,k,n,m) = 0.2*r(i,j,k,l,n,m) - x(i,j,k,n,m)
11              y(i,j,l,n,m) = 0.2*r(i,j,k,l,n,m) - y(i,j,l,n,m)
12              z(i,k,l,n,m) = 0.2*r(i,j,k,l,n,m) - z(i,k,l,n,m)
13              total(j,k,l,m) = total(j,k,l,m) + r(i,j,k,l,n,m)
14            END DO
15          END DO
16        END DO
17      END DO
18    END DO
19  END DO
```

Listing 2: The optimised mega-stream kernel

With these optimisations in place, the mega-stream benchmark is obtaining close to 80% of STREAM bandwidth on Knights Landing MCDRAM, a significant increase over the initial 16.4% we observed. The mega-stream benchmark has one large read data stream and one large write data stream, and therefore we would not expect to reach close to Triad bandwidth which has two read and one write stream. The Knights Landing MCDRAM has separate channels for read and write and therefore we will not maximise the memory bandwidth available with a single read stream [6]. The Scale kernel in the STREAM benchmark is more similar to the read and write balance here, which achieves 400 GB/s on Knights Landing and 100 GB/s on Broadwell. Based on Scale as an achievable peak instead of Triad, mega-stream is now achieving 87.3% of the available memory bandwidth on Knights Landing, a significant improvement over the baseline. On Broadwell it achieves well over 100% of the memory bandwidth according to our model, indicating good cache usage — the model over-estimates the number

of bytes loaded into cache from memory through the assumption that all future reads after the first are not counted. The STREAM kernels are simple and so we not expect to achieve parity with this more complex benchmark.

The solid lines in Fig. 3 show the final estimated memory bandwidth after the optimisations. Remember that this figure explores the variation in achieved memory bandwidth over different problem sizes with the baseline performance for the various input sizes shown with dashed lines. Firstly note that the results are more consistent with each other, generally within 6% (excluding the first points), compared to 145% initially. As such, the effective utilisation of the available memory bandwidth is no longer as dependent on the problem size. With the "middle" iterations set to four (leftmost point of the "middle" dashed green line), the optimised code actually realises an *increased* runtime for this input; this problem size is fully cache resident and hence non-temporal stores moving the memory out of cache to MCDRAM are a hindrance.

Running the optimised mega-stream utilising the hyperthreads on the Knights Landing results in reduced bandwidth. Using 2 and 4 hyperthreads per physical core with `Nm` set to 128 and 256 as before, the bandwidth is estimated as 300 and 245 GB/s respectively; recall running 1 thread per physical core achieves 349 GB/s. This behaviour is in-line with running memory bandwidth bound kernels such as those in the STREAM benchmark.

The Roofline analysis in the Intel Advisor tool shows that for the optimised code the `Nj` loop is limited by L2 cache bandwidth, however again does not show results for the kernel as a whole.

6 Conclusions

A simple benchmark code with sensible, stride 1 memory access patterns and vectorisation is shown which initially does not take full advantage of available memory bandwidth; yet the code should be memory bandwidth bound due to its low computational intensity. The code follows a pattern which may be present in a wide range of important codes: a stencil style access where cell edge values contribute to neighbouring cells. The results we present in this paper could therefore help identify many more cases where performance on Knights Landing could be significantly improved. We have presented a series of three optimisations which improve the runtime of our simple benchmark code by 4X on the Intel Xeon Phi (Knights Landing) processor, and thus allow it to take advantage of the improved memory bandwidth on this architecture. The optimisations also helped on Intel Xeon processors with close to a 1.5X speedup, however due to the large cache sizes on these processors the improvement is much smaller than on Knights Landing.

We are planning on examining the performance of mega-stream on other cache based CPU architectures as well as a GPU port; focusing on uncovering the fundamental reasons why the GPU port of SNAP achieves good performance. We will also apply the techniques and optimisations discussed to SNAP itself.

Acknowledgement

We would like to thank John Pennycook and Andrew Mallinson of Intel Corporation for their assistance with this work. The mega-stream code is made available from the UK Mini-App Consortium on GitHub at https://github.com/UK-MAC/mega-stream. The University of Bristol is an Intel Parallel Computing Center, and the authors would like to thank Intel Corporation for the provision of the Intel Xeon Phi (Knights Landing) Processor. The authors would like to thank Cray Inc. for providing access to the Cray XC40 supercomputer, "Swan".

References

1. Deakin, T., McIntosh-Smith, S., Gaudin, W.: Many-Core Acceleration of a Discrete Ordinates Transport Mini-App at Extreme Scale, pp. 429–448. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-41321-1_22 http://link.springer.com/10.1007/978-3-319-41321-1_22
2. Deakin, T., McIntosh-Smith, S., Martineau, M., Gaudin, W.: An Improved Parallelism Scheme for Deterministic Discrete Ordinates Transport. International Journal of High Performance Computing Applications (sep 2016), http://hpc.sagepub.com/cgi/doi/10.1177/1094342016668978
3. Deakin, T., Price, J., Martineau, M., McIntosh-Smith, S.: GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models. pp. 489–507 (2016), http://link.springer.com/10.1007/978-3-319-46079-6_34
4. Intel: Chapter 10, Programming with Intel Streaming SIMD Extensions, Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1. Intel Corporation (12 2016)
5. Jeffers, J., Reinders, J., Sodani, A.: Chapter 25 - Trinity workloads BT - Intel Xeon Phi Processor High Performance Programming (Second Edition). In: Intel Xeon Phi Processor High Performance Programming, pp. 549–579. Morgan Kaufmann, Boston (2016), http://www.sciencedirect.com/science/article/pii/B9780128091944000259
6. Jeffers, J., Reinders, J., Sodani, A.: Quantum chromodynamics. In: Intel Xeon Phi Processor High Performance Programming, pp. 581–598. Elsevier (2016), http://linkinghub.elsevier.com/retrieve/pii/B9780128091944000260
7. Lamport, L.: The Parallel Execution of DO Loops. CACM - Communications of the ACM 17(2), 83–93 (1974)
8. McCalpin, J.D.: Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (dec 1995)
9. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM 52, 65–76 (2009)
10. Zerr, R.J., Baker, R.S.: SNAP: SN (Discrete Ordinates) Application Proxy - Proxy Description. Tech. rep., LA-UR-13-21070, Los Alamos National Labratory (2013)