Data Layout Optimization for Portable Performance

Kamal Sharma[1], Ian Karlin[2], Jeff Keasler[2], James R. McGraw[2], and Vivek Sarkar[1]

[1] Rice University, Houston, TX, USA. {kamal.g.sharma,vsarkar}@rice.edu
[2] Lawrence Livermore National Laboratory, Livermore, CA, USA. {karlin1,keasler1,mcgraw1}@llnl.gov

Abstract. This paper describes a new approach to managing data layouts to optimize performance for array-intensive codes. Prior research has shown that changing data layouts (e.g., interleaving arrays) can improve performance. However, there have been two major reasons why such optimizations are not widely used in practice: (1) the challenge of selecting an optimized layout for a given computing platform, and (2) the cost of re-writing codes to use different layouts for different platforms. We describe a source-to-source code transformation process that enables the generation of different codes with different array interleavings from the same source program, controlled by data layout specifications that are defined separately from the program. Performance results for multicore versions of the benchmarks show significant benefits on four different computing platforms (up to 22.23× for IRSmk, up to 3.68× for SRAD and up to 1.82× for LULESH). We also developed a new optimization algorithm to recommend a good layout for a given source program and specific target machine characteristics. Our results show that the performance obtained using this algorithm achieves 78%-95% performance of the best manual layout on each platform for different benchmarks (IRSmk, SRAD, LULESH).

1 Introduction

As computing platforms increase in diversity, "portable performance" has become one of the most challenging problems for application developers. Achieving good performance on a specific platform often requires coding adjustments to fit a specific set of machine parameters e.g., number of cores, cache size, cache line size, number of registers, memory bandwidth, etc. Unfortunately, adjustments for one platform can impede performance on other platforms. This paper focuses on *data layout optimization*, which has been increasing in importance in recent years. Most programming languages require developers to make array-of-struct (AoS) or struct-of-array (SoA) decisions (or combinations thereof) early in development. For long-lived applications, the following challenge can be encountered repeatedly (and now with increasing frequency): what to do when a new parallel architecture is introduced with a memory and storage subsystem that would benefit from a different data structure layout in the program? This question is taking on a new urgency as proposals for exascale architectures increasingly include major changes in memory and storage structures. With current languages, a near-complete rewrite of an application is usually required, because each data access usually needs to be rewritten when the data layout is changed. Historically, developers of large codes avoid changing data layouts because it involves changing too many lines of code, the expected benefit of a specific change is difficult to predict, and whatever works well on one system may hurt on another. Our approach demonstrates how these obstacles can be overcome.

This paper is organized as follows. Section 2 describes a motivating example (IRSmk) and shows that changing array layouts can significantly impact performance on four different parallel platforms. Section 3 introduces our extensions to TALC, a source-to-source transformation tool. TALC enables the same program to be compiled and executed with different data layouts, without requiring any changes to the source code. Section 4 presents a new automatic optimization algorithm to recommend an optimized layout for a given source program and target machine. Section 5 presents a summary of empirical results obtained for three benchmarks (IRSmk, SRAD, LULESH) on four different multicore platforms: IBM POWER7, AMD APU, Intel Sandy Bridge, and IBM BG/Q. Section 5 also presents results from the automated layout algorithm that are very close to the hand tuned manual layouts. Finally, Section 6 summarizes related work, and Section 7 contains our conclusions and plans for future work.

2 Motivating Example

We use the IRSmk benchmark (a 27-point stencil loop kernel in the ASC Sequoia Benchmark Codes [2]) as a motivating example to illustrate the impact of data layouts on performance. IRSmk is an Implicit Radiation Solver for diffusion equations on a block-structured mesh. Figure 1 shows the main loop kernel of IRSmk. For simplicity, we do not consider arrays starting with the letter x as candidates for layout optimization, since they all alias to the same array with different offsets. We also ignore array b since it only occurs in a single write access. This leaves 27 arrays as candidates for layout optimization (dbl to ufr).

```
for ( kk = kmin ; kk < kmax ; kk++ ) {
  for ( jj = jmin ; jj < jmax ; jj++ ) {
    for ( ii = imin ; ii < imax ; ii++ ) {
      i    = ii + jj * jp + kk * kp ;
      b[i] =  dbl[i] * xdbl[i] + dbc[i] * xdbc[i] + dbr[i] * xdbr[i]
            + dcl[i] * xdcl[i] + dcc[i] * xdcc[i] + dcr[i] * xdcr[i]
            + dfl[i] * xdfl[i] + dfc[i] * xdfc[i] + dfr[i] * xdfr[i]
            + cbl[i] * xcbl[i] + cbc[i] * xcbc[i] + cbr[i] * xcbr[i]
            + ccl[i] * xccl[i] + ccc[i] * xccc[i] + ccr[i] * xccr[i]
            + cfl[i] * xcfl[i] + cfc[i] * xcfc[i] + cfr[i] * xcfr[i]
            + ubl[i] * xubl[i] + ubc[i] * xubc[i] + ubr[i] * xubr[i]
            + ucl[i] * xucl[i] + ucc[i] * xucc[i] + ucr[i] * xucr[i]
            + ufl[i] * xufl[i] + ufc[i] * xufc[i] + ufr[i] * xufr[i];
} } }
```

Fig. 1: IRSmk Source Code

As a preview of results to come (with larger number of layouts), we look at four different array layouts here to illustrate the potential for performance gains on different platforms. The default layout is the one observed in Figure 1, where the 27 arrays are stored separately (27×1). A simple rewrite can change the layout

Table 1: Performance improvement of different layouts relative to baseline 27×1 layout, for different platforms

Platform	27×1	9×3	3×9	1×27
IBM POWER7	1.00	4.66	4.66	4.71
AMD APU	1.00	1.26	1.38	1.40
Intel Sandy Bridge	1.00	1.06	1.10	1.10
IBM BG/Q	1.00	1.65	2.14	2.20

by interleaving[3] groups of three arrays, thus producing 9 arrays of structs where each structure contains 3 fields (9×3). Another rewrite can interleave 9 arrays each, producing three arrays (3×9). The final rewrite interleaves all 27 arrays into one array (1×27). We ran these four versions of IRSmk on four different platforms: IBM Power7, AMD APU, Intel Sandy Bridge, and the IBM BG/Q, using a problem size of 100^3 and all cores within a single node on each platform. The results are presented in Table 1. All examples show positive gains for all of the layout options. However, the performance improvement varies dramatically across different layouts and different platforms.

3 TALC Data Layout Framework

This section describes our extensions to the TALC Framework [14] to support user-specified and automatic data layouts, driven by a Meta file specification. The past framework had limited capabilities in terms of error checking, no automatic layout selection, did not take into consideration machine characteristics and profiled information, and explored limited platforms. TALC stands for Topologically-Aware Layout in C. TALC is a source-to-source compiler translation tool and accompanying runtime system that dramatically reduces the effort needed to experiment with different data layouts. Our extended version of TALC has been implemented in the latest version of the ROSE [4] compiler infrastructure. In the process of extending TALC, we have re-implemented its entire code, added new functionality for automated layouts and extended layout transformations [3]. Our new tool explores a wide range of layouts, considers platform characteristics and profile information and performs safety and error checking for different layouts.

Figure 2 shows our extended TALC framework. TALC can be configured to run in two modes: Automated Layout and User Specified Layout. For both these modes, a user needs to provide some input to perform data layout transformations. In the Automated Layout mode, the user provides a *field specification*. A field specification file is a simple schema file, which specifies arrays that should be considered for transformation. The field specification file is necessary because it enables our tool to only transform the specified arrays (like the 27 arrays in the IRSmk example discussed in Section 2). Figure 3 shows a sample field specification file. The View keyword is used to specify a data layout. The field keyword specifies arrays considered for layout transformation. Each field has a type associated with it, specified by the : separator. In this example, d stands for the double data type. Specifying the data type helps with type checking array subscripts during layout transformations. Further details about our TALC tool

[3] In this paper, we use array regrouping and interleaving interchangeably.

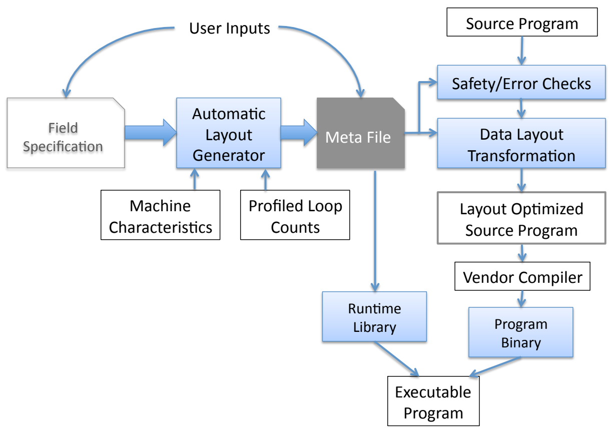


Fig. 2: Extended TALC Framework

```
View node
{
   Field {x:d}
   Field {y:d}
   Field {z:d}
...
}
```

Fig. 3: Sample Field Specification file

```
View node
{
  Field {x:d,y:d,z:d}
  Field {xd:d,yd:d,zd:d}
  Field {xdd:d,ydd:d,zdd:d}
  Field {fx:d,fy:d,fz:d}
}
```

Fig. 4: Sample TALC Meta file

along with a working example can be found in our technical report [18]. More information on the Automatic Data Layout Selection is provided in the next section.

4 Automatic Data Layout Selection

In this section, we describe the automatic data layout selection algorithm. The algorithm takes in a user-written field specification file which specifies arrays that should be considered for transformation, and uses a greedy optimization algorithm to automatically construct a data layout based on the input program and target architecture.

4.1 Automatic Data Layout Algorithm

Our automated data layout algorithm uses the cache-use factors and platform characteristics to produce a meta file that contains the recommended data layout. Algorithm 1 shows the automated data layout algorithm. More details can be found in the related technical report [18]. To begin with, each array in the field specification is placed in its own ArrayGroup. As a heuristic, we disallow arrays that occur in vectorizable loops as candidates for data layout transformation. This heuristic is used to avoid performance degradations that may result from data layout transformations breaking vectorization. We expect this heuristic to be relaxed in the future when processors have more flexible vector capabilities with respect to memory accesses, compared to today's processors. The algorithm compares all pairs of ArrayGroups to determine the profitability of merging each pair. We use Cache-Use Factor (CUF) as a cost metric to capture the possible cache impact of merging two or more array groups. This factor denotes cache

Algorithm 1 Automated Data Layout Algorithm

```
1: procedure AutoDataLayout(ArrayGroupList)
2:     for loop L in the program do
3:         if loop L is vectorizable (based on vector pragma or compiler analysis)
4:             Remove arrays in loop from ArrayGroupList
5:         end if
6:     end for
7:     IsMerge ← true
8:     while IsMerge is true do
9:         IsMerge ← false
10:        for pairs ∈ ArrayGroupList do
11:            if (pair writes) > 2*(pair reads+pair read/writes)
12:                Ignore pair
13:            end if
14:            best pair ← pair with highest cache use factor
15:        end for
16:        if CUF resulting from merging best pair > threshold
17:            merge pair
18:            IsMerge ← true
19:        end if
20:    end while
21:    sortGroups(ArrayGroupList)
22:    splitCacheLine(ArrayGroupList)
23:    return ArrayGroupList
24: end procedure
```

usage efficiency across all the candidate loops in the program. The CUF metric helps limit the amount limit of merging performed by our greedy algorithm. The pair with the highest cache-use factor is merged to form a new group. This process is repeated until the best candidate pair for merging falls below the acceptable merge threshold. After the final grouping is determined, each group's arrays are sorted based on data type(largest data size to smallest data size), to better pack them. The final step performs cache line splitting i.e. split array groups based on cache line boundaries of an architecture, to further improve cache line utilization within a group.

The evaluation of the profitability of merging two candidate ArrayGroups considers two factors. The first consideration examines reads versus writes to an ArrayGroup. Our experimental results (Section 5) showed that grouping arrays written to frequently with arrays that are only read can decrease performance significantly. Our current heuristic prohibits creating a new merged ArrayGroup, if the number of write-only arrays is more than $2\times$ the number of read and read-write arrays The second consideration for merging ArrayGroups computes the cache use factor for the proposed combination. If the cache use factor is greater than our established thresholds, the ArrayGroups are viable for merging. From our empirical results, we have chosen $\textit{Cache Use threshold} = 0.57$ for our algorithm. A detailed analysis to study the effects of varying this threshold across architectures and benchmarks is a subject of future work.

5 Experimental Results

We ran a series of tests to evaluate the productivity and performance gains obtained by using TALC to perform layout transformations. In Section 5.1, we first describe our experimental methodology. We then provide a detailed discussion of performance results for user-specified layouts, obtained by evaluating a range of manual layouts on different architectures (Section 5.2). Finally, we we present performance results obtained by using our automatic layout algorithm (Section 5.3).

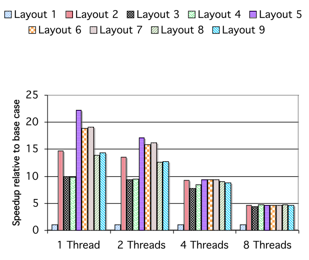
5.1 Experimental Methodology

To show the impact of data layouts on performance we ran experiments using our three benchmark programs on four different platforms: *IBM Power7*, *AMD APU*, *Intel Sandy Bridge* and *IBM BG/Q*. *IBM Power7* represents a 32-core IBM Power 7 processor system (four eight-core 3.55 GHz processor, 32KB L1 D-Cache per core, 256KB L2 Cache, 32MB L3 Cache) used with compiler options `xlc-v11.1 -O3 -qsmp=omp -qthreaded -qhot -qtune=pwr7 -qarch=pwr7`. *AMD APU* represents a 4-core AMD A10-5800K APU processor (quad-core 3.8 GHz processor, 16KB L1 D-Cache per core, 4MB L2 Cache) used with compiler options `gcc-v4.7.2 -O3 -fopenmp`. *Intel Sandy Bridge* represents a 16-core Intel E5-2670 Sandy Bridge CPU system (eight-core 2.6 GHz processor, 32KB L1 D-Cache per core, 256KB L2 Cache per core, 20MB L3 Cache) used with compiler options `icc-v12.1.5 -O3 -fast -parallel -openmp`. *IBM BG/Q* represents a 16-core IBM PowerPC A2 system (1.6 GHz processor, 32 MB eDRAM L2 cache) used with compiler options `gcc-v4.4.6 -O3 -fopenmp`. For the *AMD APU*, we focused on the CPU and ignored the GPU. IRSmk and LULESH were both run in double precision, while SRAD was run in single precision. Specifically, we ran IRSmk on a problem based on a 100^3 mesh for 500 iterations. LULESH was run with a problem size = 90 (i.e. 90^3 elements and 91^3 nodes). SRAD was run for 200 iterations on a 4096^2 grid with the $x1$ and $y1$ speckle values set to 0, the $x2$ and $y2$ values set to 127 and lambda set to 0.5. We made minor changes to the original source program to conform to extended TALC framework specifications. These changes related to renaming the program variables and did not affect program execution in any way.

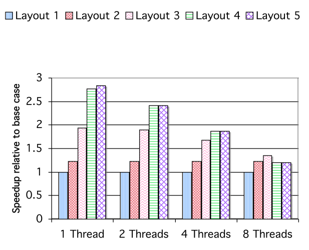
All of these benchmarks use OpenMP for parallelism. We use the default memory allocation scheme provided in these benchmarks and limited our experiments to one socket. Studying the Non Uniform Memory Access (NUMA) effects with data layouts will be part of future work. All benchmarks were run with varying thread counts on the four platforms. For this work, we only used the default memory allocation provided on these systems. For all codes, TALC enabled testing a range of layouts, 9 for IRSmk and 11 for LULESH and 5 for SRAD. To perform the layout transformations in IRSmk, between 56 and 272 (82%) lines of the original 330 lines of code were changed. For the LULESH the numbers are 98 to 477 (18%) lines of the original 2640. For SRAD the number are 11 to 39 (16%) lines of the original 239. By using TALC, we not only were able to automate these changes, but also eliminate the possibility of subtle bugs being introduced when these changes are performed manually (and repeated for different architectures).

5.2 User Specified Layout Results

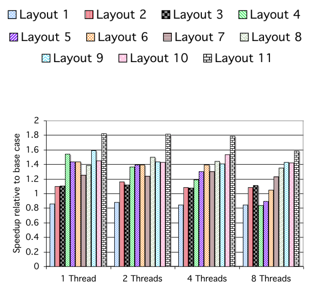
For each benchmark, we conducted extensive experiments across different layouts on four architectures. However, due to space limitations, we limit the number of layouts presented here to the most interesting ones. For each test case, we report the speedup (which can be a slowdown, for *values* < 1) of each layout against the "base case" which is the original code, running with an equivalent number of threads. In some cases, for example IBM BG/Q for 2, 4 and 8 threads, we omit showing results for all thread counts because their results were similar to adjacent thread counts of 1 and 16. However, full details with results for all layouts can be found in [18].



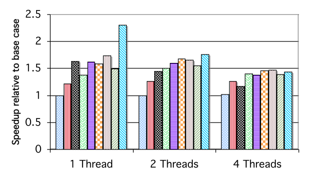
(a) IRSmk - IBM Power7



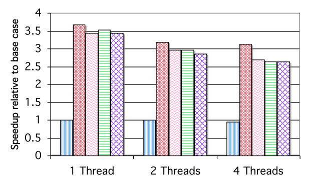
(b) SRAD - IBM Power7



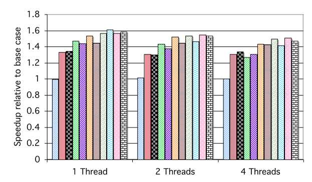
(c) LULESH - IBM Power7



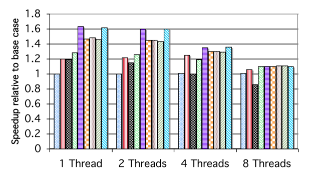
(d) IRSmk - AMD APU



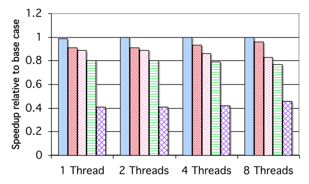
(e) SRAD - AMD APU



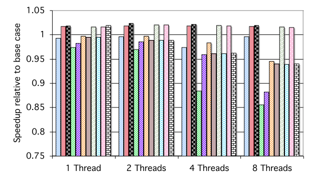
(f) LULESH - AMD APU



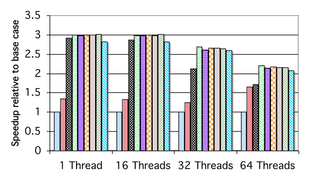
(g) IRSmk - Intel Sandy Bridge



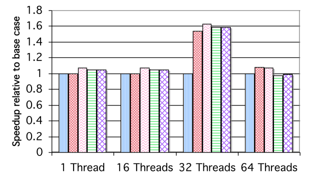
(h) SRAD - Intel Sandy Bridge



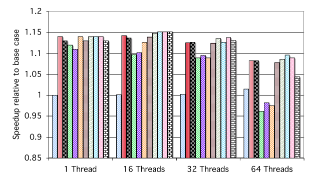
(i) LULESH - Intel Sandy Bridge



(j) IRSmk - IBM BG/Q



(k) SRAD - IBM BG/Q



(l) LULESH - IBM BG/Q

Fig. 5: Benchmark performance results on different platforms with varying threads.

IRSmk The implicit radiation solver (IRS) [2] is a benchmark used as part of the procurement of the Sequoia system at LLNL. Figures 5a, 5d, 5g and 5j show the results obtained by running IRSmk with different thread counts on all nine layouts on each of the four platforms. IRSmk is a memory bound kernel whose performance is limited by memory bandwidth. However, we see that except for Sandy Bridge significant speedups occur at all thread counts due to data layouts.

The results of the best layout for IRS on all machines show performance of at least 70% of optimal and over 95% on Sandy Bridge. For Sandy Bridge, the execution time for the best layout is 3.05 seconds, for the AMD APU it is 10.04 seconds, for BG/Q it is 5.2 seconds and for the Power 7 it is 12.52 seconds. BG/Q performs slightly worse than other architectures due to in-order cores not hiding as much latency as the other processors, while the AMD APU could be hurt by less data in the x array staying in its smaller cache. Finally, all the processors might be limited in their handling of the unequal amount of read and write data in IRSmk.

On the Sandy Bridge, data layouts only sped up the computation by $1.11\times$. Since, the base case was already running at over 85% of peak memory bandwidth. On the other processors, performance is significantly worse for the base case. A related trend is that improvements from data layouts are more significant at lower core counts. This implies two conclusions. First compute-bound codes also benefit from data layout transformations. In the case of Sandy Bridge where there are enough stream prefetchers for the base code and enough bandwidth to feed a few, but not, all, cores merging arrays reduces the number of registers used as pointers by the compiler resulting in fewer instructions and possibly fewer pipeline stalls. Another benefit is that the number of elements accessed in each loop from an array can be matched to cache boundaries, such as layout 9. The second observation is that for processors with an under provisioning of prefetchers when fewer cores are used the computation becomes latency-bound. With fewer cores to issue memory requests, the memory bus becomes idle for a larger percentage of the time. Therefore, bandwidth is used less efficiently, allowing for larger speedups when the core uses it more effectively.

A final observation is that not merging read only arrays in a loop with arrays that are written increases the performance significantly. Modern architectures, such as AMD APU, often implement a write buffer to combine multiple writes to the same cache line to reduce the amount of data sent to main memory, known as Write-Combining [1].

SRAD The SRAD benchmark [7] from the Rodinia suite performs the image processing calculation speckle reducing anisotropic diffusion. The algorithm removes speckles (locally correlated noise) from images without removing image features. We focus on the loop that iterates over the image as it takes almost all of the time in SRAD. Figures 5b, 5e, 5h and 5k show SRAD results for running the five layouts on our four test platforms. Due to space constraints, we have omitted the five layout details. SRAD contains many of the same trends and results as IRSmk, but adds some new features and complexity. SRAD contains multiple loops so there are cases where two arrays are used together in one loop and only one array is used in another loop. Examples of this are the IN, IS loop pair and the JW, JE set of loops. SRAD is run on more compute intense problems where vectorization can increase its performance significantly. Our results show how some of these tensions impact performance. On the Sandy Bridge chip with the Intel compiler SRAD gets a significant performance boost from vector instructions. However, when data layout transformations are performed the compiler no longer vectorizes any instructions due to the use of pointers to the structures. The result is a performance hit from vectorization that is greater than the gain from data layout transformations. To confirm this we ran the base version of SRAD with compiler vectorization turned off and data layout transformations resulting in a 1.66× to 1.84× speedup from data layout transformations. For future work, we plan on investigating how to generate array of struct of array code(AoSoA) that the compiler can still vectorize. In the current tool, we ignore the array references where the loop has a vector pragma associated with it. Overall, performance gains on SRAD ranged from the minor 1.07× on most BG/Q thread counts to 3.68× on a single thread of an AMD APU.

LULESH The largest application we focus on is the Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) mini-application [5]. LULESH solves the Sedov problem on a 3D hexahedral mesh. Different array sizes and the fact that they are used in various combinations throughout the loops in LULESH provide a larger search space for data layouts and tension for data layout transformations not found in the smaller benchmarks. The version of LULESH used in this study has undergone a variety of optimizations from the original published code, including aggressive loop fusion and elimination of temporary arrays [13]. Figures 5c, 5f, 5i and 5l show LULESH results for running the eleven layouts on our four test platforms. Due to space constraints, we have omitted the eleven layout details. We only show selected interesting thread counts (with the full data available in [18]).

Data layout transformations on LULESH were less profitable overall than for IRSmk. This is not surprising since LULESH is a larger application than IRSmk, and some arrays in LULESH are used together in certain places and not together in others. Therefore, combining them together will help and hurt performance simultaneously[4]. For example, layout 4 combines all four triples of x, y , z values together. Many of these triples are used together in many functions, but not all. However, most of the time layout 6 which leaves the triples separate is faster. A notable exception can be seen on Power7 for a single thread, which has the most cache, but the least bandwidth. It also suffers the most from not getting good prefetching as shown by the IRSmk results.

The most interesting result from LULESH is that in most cases it seems the code, not the hardware, is dictating the best data layout. On the AMD APU, Intel Sandy Bridge and BG/Q the list of the best layouts always includes 8 and 10 and usually, includes 2 and 3. However, the Power7 is an outlier with its best layout being 11 for all thread counts by a significant margin for the reasons explained above. For LULESH, as with IRSmk and SRAD, data layouts impacted the Sandy Bridge system the least with the largest speedup seen being only 1.02×. There are a few likely reasons for this. First, as with IRSmk, the Sandy Bridge architecture should be able to prefetch many streams at once. Also,

[4] This phenomenon motivates future work on redistributing data layouts across phases.

Table 2: Speedup of best Manual Layout (ML) and Automated Layout (AL) speedup relative to base layout

Benchmark	Power7-8Threads	AMD APU-4Threads	Sandy Bridge-8Threads	BG/Q-64Threads
IRSmk ML	4.70	1.46	1.11	2.20
IRSmk AL	4.67	1.43	1.10	2.08
LULESH ML	1.43	1.50	1.02	1.10
LULESH AL	1.58	1.46	0.96	1.07
SRAD ML	1.35	3.13	1.00	1.08
SRAD AL	1.20	2.55	0.46[†]	0.98

[†] Assuming vector pragma is not specified. If vector pragma is specified then results is same as ML.

in the case of bundling indirect accesses, the large re-order window of the Sandy Bridge might hide memory latency better than the other chips. Finally, the Intel compiler used on this platform was the best at generating SIMD instructions for some of the compute bound loops of LULESH. Some of the data transformations result in the compiler no longer generating SIMD instructions and, therefore, while data layouts save on data motion in memory-bound portions of the code they can sometimes hurt performance in the compute bound sections.

5.3 Automatic Data Layout Results

Table 2 shows the speedup of the best manual layout and the automated layout relative to the base layout. The results demonstrate that automated layouts can come close to the best manual layout in most cases. In one particular case, 8 Threads on Power7 for LULESH, automated layout improved performance as compared to manual layouts. For SRAD, automated results were close to the best manual results for Power7 and BG/Q. However, the results fell behind manual results for AMD APU and Sandy Bridge. In both cases, we suspect that the data layout transformation inadvertently disabled some compiler optimizations, especially vectorization in the case of Sandy Bridge. (All results in this section were obtained without enabling the vectorization test at the start of the automatic layout algorithm).

6 Related Work

Past research has proposed various data layout optimization techniques [6,8,9]. Here, we present a brief survey of past work, focusing on aspects that are most closely related to our work.

Zhang et al. [20] introduced a data layout framework that targets on-chip cache locality, specifically reducing shared cache conflicts while observing data patterns across threads. Using polyhedral analysis, their framework rearranges data layout tiles to reduce on-chip shared cache conflicts. However, their optimization currently works with single arrays. In contrast, our approach works on merging multiple arrays and operates at the element level rather than tiles. Henretty et al. [11] presented a data layout framework to optimize stencil operations on short-SIMD architectures. Their work specifically targets stream alignment conflicts on vector registers and uses a dimension transposition method (non-linear data layout optimization) to mitigate the conflicts. In comparison, our approach works for more general applications, not just stencil code. Also, our work did not specifically address the impact of data layout on vectorization. Ding and Kennedy [9] introduced a data-regrouping algorithm, which has similarities to our work on automatic selection of data layouts. Their compiler analysis merges multi-dimensional arrays based on a profitability cache analysis. Dynamic regrouping was also provided for layout optimization at runtime. Experimental results show significant improvement in cache and TLB hierarchy. However, their results were all obtained on uniprocessor systems and it is unclear how their approach works in the presence of data aliasing. Raman et al. [17] used data layout transformations to reduce false sharing and improve spatial locality in multi-threaded applications. They use an affinity based graph approach (similar to our approach) to select candidates. Inter-procedural aliasing issues arising due to pointers is not addressed in this work. Our work is intended to explore data layout transformations more broadly, not just for false sharing and spatial locality. Using polyhedral layout optimization, Lu et al. [15] developed a data layout optimization for future NUCA CMP architectures. Their work reduces shared cache conflict on such architectures. Simulation results show significant reductions in remote accesses. Finally, a number of papers, [10,12,16,19] have explored the integration of loop and data layout transformations. To the best of our knowledge, our work is the first to support both user-specified and automatic AoS data layout transformations, while allowing the user to provide a data layout specification file. Our results on the LULESH mini-application demonstrates the importance of data layout transformations on modern multicore processors.

7 Conclusions

This paper establishes the foundation for a new approach to supporting portable performance of scientific codes across HPC platforms. The upgraded TALC source-to-source transformation tool permits application developers to maintain one “neutral” data layout source code and explore architecture specific array layouts. The new automated portion of TALC can analyze the original source code based on platform characteristics and produces a new source code with new array data layouts ready to be compiled and run on that system. The results for the three test codes show that manual layouts improve performance by 1.10× to 22.23× for IRSmk, 1.00× to 3.68× for SRAD and 1.02× to 1.82× for LULESH with results varying with thread count and architecture. The automated algorithm achieves 95-99% of the best layout manual layout performance for IRSmk. For LULESH the automated approach achieves 90% of the best layout performance on all processors. For SRAD, automated results achieves 78% of best manual layout performance for all architectures except for Intel Sandybridge where layouts interfered with vectorization provided by Intel compiler.

Our future direction is to expand the flexibility of constraints on the original source code to include manipulation of multi-dimensional arrays. Finally, we also need to include enriched layouts (such as AoSoA) that reduce interference with vectorization. Another interesting direction to pursue is to develop specialized data layouts for accelerators such as GPU and Xeon Phi. We look forward to pursuing these challenges in the future.

Acknowledgments We thank the anonymous reviewers for their feedback to improve the presentation of the paper. This work was prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-672056).

References

1. AMD64 Architecture Programmer's Manual Volume 2
2. ASC Sequoia Benchmark, `https://asc.llnl.gov/sequoia/benchmarks/`
3. Extended TALC Infrastructure, `https://github.com/rose-compiler/edg4x-rose/tree/master/projects/TALCDataLayout`
4. ROSE Compiler Infrastructure, `http://rosecompiler.org/`
5. Hydrodynamics Challenge Problem. Tech. Rep. LLNL-TR-490254, LLNL (July 2011), `https://computation.llnl.gov/casc/ShockHydro`
6. Calder, B., Krintz, C., John, S., Austin, T.: Cache-conscious data placement. pp. 139–149. ACM ASPLOS VIII (1998)
7. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. IEEE IISWC 2009
8. Chilimbi, T.M., Hill, M.D., Larus, J.R.: Cache-conscious structure layout. pp. 1–12. PLDI '99 (1999)
9. Ding, C., Kennedy, K.: Inter-array Data Regrouping. pp. 149–163. LCPC '99, Springer-Verlag, London, UK, UK (2000)
10. Ding, C., Kennedy, K.: Improving Effective Bandwidth through Compiler Enhancement of Global Cache Reuse. pp. 10 pp.–. IEEE IPDPS '01 (Apr 2001)
11. Henretty, T., Stock, K., Pouchet, L.N., Franchetti, F., Ramanujam, J., Sadayappan, P.: Data layout transformation for stencil computations on short-vector SIMD architectures. pp. 225–245. CC'11/ETAPS'11 (2011)
12. Kandemir, M., Choudhary, A., Ramanujam, J., Banerjee, P.: A Framework for Interprocedural Locality Optimization Using Both Loop and Data Layout Transformations. pp. 95–102. IEEE ICPP '99 (1999)
13. Karlin, I., McGraw, J., Keasler, J., Still, C.: Tuning the LULESH Mini-app for Current and Future Hardware. In: (NECDC12) (December 2012)
14. Keasler, J., Jones, T., Quinlan, D.: TALC: A Simple C Language Extension For Improved Performance and Code Maintainability. In: 9th LCI International Conference on High-Performance Clustered Computing (April 2008)
15. Lu, Q., Alias, C., Bondhugula, U., Henretty, T., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P., Chen, Y., Lin, H., Ngai, T.f.: Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors. pp. 348–357. IEEE PACT '09 (Sept 2009)
16. O'Boyle, M.F.P., Knijnenburg, P.M.W.: Efficient Parallelization using Combined Loop and Data Transformations. pp. 283–. PACT '99 (1999)
17. Raman, E., Hundt, R., Mannarswamy, S.: Structure Layout Optimization for Multithreaded Programs. pp. 271–282. IEEE CGO '07 (2007)
18. Sharma, K., Karlin, I., Keasler, J., McGraw, J.R., Sarkar, V.: User-Specified and Automatic Data Layout Selection for Portable Performance (April 2013), `http://cohesion.rice.edu/engineering/computerscience/tr/TR_Download.cfm?SDID=307`
19. Taylan Kandemir, M.: Improving whole-program locality using intra-procedural and inter-procedural transformations. J. Parallel Distrib. Comput. 65(5), 564–582 (May 2005)
20. Zhang, Y., Ding, W., Liu, J., Kandemir, M.: Optimizing Data Layouts for Parallel Computation on Multicores. pp. 143–154. PACT '11 (Oct 2011)