Estimation d'efficacité et restructuration automatisées de noyaux de calcul

Christopher Haine

université de BORDEAUX

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Christopher Haine**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Kernel optimization by layout restructuring

Date de soutenance : 3 Juillet 2017

Devant la commission d'examen composée de :

Henri-Pierre CHARLES ..	Directeur de Recherche, CEA	Rapporteur
Allen MALONY	Professeur, Université de l'Oregon	Rapporteur
Emmanuel JEANNOT	Directeur de Recherche, Inria	Examinateur
Pascale ROSSE-LAURENT	Ingénieur, Atos	Examinateur
Olivier AUMAGE	Chargé de Recherche, Inria	Encadrant
Denis BARTHOU	Professeur des Universités, Bordeaux INP	Directeur de Thèse

– 2017 –

Abstract

Careful data layout design is crucial for achieving high performance, as nowadays processors waste a considerable amount of time being stalled by memory transactions, and in particular spacial and temporal locality have to be optimized. However, data layout transformations is an area left largely unexplored by state-of-the-art compilers, due to the difficulty to evaluate the possible performance gains of transformations. Moreover, optimizing data layout is time-consuming, error-prone, and layout transformations are too numerous to be experimented by hand in hope to discover a high performance version.

We propose to guide application programmers through data layout restructuring with an extensive feedback, firstly by providing a comprehensive multidimensional description of the initial layout, built via analysis of memory traces collected from the application binary *in fine* aiming at pinpointing problematic strides at the instruction level, independently of the input language. We choose to focus on layout transformations, translatable to C-formalism to aid user understanding, that we apply and assess on case study composed of two representative multithreaded real-life applications, a cardiac wave simulation and lattice QCD simulation, with different inputs and parameters. The performance prediction of different transformations matches (within 5%) with hand-optimized layout code.

Keywords:
Performance profiling, Layout restructuring, Vectorization, Binary rewriting

Résumé

Bien penser la structuration de données est primordial pour obtenir de hautes performances, alors que les processeurs actuels perdent un temps considérable à attendre la complétion de transactions mémoires. En particulier les localités spatiales et temporelles de données doivent être optimisées. Cependant, les transformations de structures de données ne sont pas proprement explorées par les compilateurs, en raison de la difficulté que pose l'évaluation de performance des transformations potentielles. De plus, l'optimisation des structures de données est chronophage, sujette à erreur et les transformations à considérer sont trop nombreuses pour être implémentées à la main dans l'optique de trouver une version de code efficace.

On propose de guider les programmeurs à travers le processus de restructuration de données grace à un retour utilisateur approfondi, tout d'abord en donnant une description multidimensionnelle de la structure de donnée initiale, faite par une analyse de traces mémoire issues du binaire de l'application de l'utilisateur, dans le but de localiser des problèmes de stride au niveau instruction, indépendamment du langage d'entrée. On choisit de focaliser notre étude sur les transformations de structure de données, traduisibles dans un formalisme proche du C pour favoriser la compréhension de l'utilisateur, que l'on applique et évalue sur deux cas d'étude qui sont des applications réelles, à savoir une simulation d'ondes cardiaques et une simulation de chromodynamique quantique sur réseau, avec différents jeux d'entrées. La prédiction de performance de différentes transformations est conforme à 5% près aux versions réécrites à la main.

Keywords:
Profilage de performance, Restructuration de données, Vectorisation, Réécriture de binaire

Contents

Introduction **11**
Data layout design for high performance . . . 11
Contributions . . . 12
Outline . . . 13

1 Context and Problem **15**
1.1 The memory layout/pattern issue . . . 16
1.1.1 Modern architectures complexity . . . 16
1.1.2 The locality issue . . . 18
1.1.3 Layout abstractions . . . 22
1.2 The complicated relationship between compilers and data layouts . . . 23
1.2.1 Static approaches . . . 24
1.2.2 Dynamic approaches . . . 25
1.2.3 Data restructuring techniques . . . 27
1.2.4 Vectorization . . . 28
1.3 The lack of user feedback . . . 29
1.3.1 Compiler feedback . . . 29
1.3.2 Tools . . . 31
1.3.3 The importance of quantification . . . 32
1.4 Towards a data layout restructuring framework proposal . . . 32
1.4.1 Proposition overview . . . 33
1.4.2 Data restructuring evaluation . . . 35

2 Vectorization **39**
2.1 Layout Transformations To Unleash Compiler Vectorization . . . 39
2.2 Hybrid Static/Dynamic Dependence Graph . . . 40
2.2.1 Static, Register-Based Dependence Graph. . . . 40
2.2.2 Dynamic Dependence Graph. . . . 41
2.3 SIMDization Analysis . . . 42
2.3.1 Vectorizable Dependence Graph. . . . 42
2.3.2 Code Transformation Hints. . . . 43
2.4 Conclusion . . . 44

3 Layout/pattern analysis **45**
3.1 Data layout formalism . . . 45
3.2 Layout detection . . . 48
3.3 Delinearization . . . 50
3.3.1 General Points . . . 52
3.3.2 Properties . . . 54
3.3.3 Characterization . . . 55
3.3.4 Case Study . . . 57
3.4 Conclusion . . . 59

4 Transformations **61**
4.1 Layout Operations . . . 61
4.1.1 Permutation . . . 62
4.1.2 Splitting . . . 63
4.1.3 Compression . . . 64
4.2 Exploration . . . 65
4.2.1 Basic Constraints . . . 65
4.2.2 Locality Constraints . . . 67
4.2.3 Parallelism Constraints . . . 69
4.3 Case Study . . . 71
4.4 Conclusion . . . 75

5 Code Rewriting and User Feedback **77**
5.1 Systematic Code Rewriting . . . 77
5.1.1 On locality . . . 78
5.1.2 Formalism Interpretation . . . 79
5.1.3 Copying . . . 81
5.1.4 Remapping . . . 83
5.2 User Feedback . . . 84
5.2.1 Layout issues pinpointing . . . 85
5.2.2 Hinting the rewriting . . . 85
5.3 Low-level implementation . . . 86
5.3.1 Loop kernel rewriting . . . 86
5.3.2 SIMDization . . . 87
5.4 Conclusion . . . 88

6 Transformations Evaluation **91**
6.1 Evaluation methodology . . . 91
6.1.1 Principle of in-vivo evaluation . . . 91
6.1.2 Automatic mock-up generation and vectorization . . . 92
6.1.3 Current state of implementation . . . 93
6.2 Experimental results . . . 94
6.2.1 TSVC . . . 94
6.2.2 Lattice QCD benchmark without preconditioning . . . 95
6.2.3 Lattice QCD benchmark with even/odd preconditioning . . . 96
6.2.4 Lattice QCD application without preconditioning . . . 97

6.2.5 2D cardiac wave propagation simulation application 98
6.3 Conclusion 98

Conclusion and Future Challenges **101**
6.4 Summary 101
6.5 Perspectives 102

Introduction

Data layout design for high performance

Applications performance is a major concern in many domains, indeed constraints from embedded system for mobile technologies, constraints of graphics rendering for video processing or scientific contraints such as precision and problem sizes for instance in the area of scientific computing – or High Performance Computing –, require significant focus on efficient computing resources utilization. Applications are so demanding that they require supercomputers, that is clusters of computers working together on the same scientific problem, to even address the most computationally challenging tasks. In such context, performance is key to minimize the time to solution, in particular some problems are unsolvable without high performance.

In computer science, there are many vectors of performance to address, one of them is the memory usage, as modern processors waste a considerable amount of time being stalled by memory transactions completion. Intrinsic memory technologies considerations prevent proper synergy between processor and memory, and memory caches has proven necessary to partially circumvent this issue. Therefore, application code performance is highly sensitive to cache usage, however optimal cache usage is difficult to achieve and poorly approximable.

Pushed by specialized domains needs, many architecture developped with distinguishing features such as Graphic Processing Units (GPUs) that use in particular large vectors to tackle imaging problems. Also, tremendous demand on computer resources fuels systematic architectural changes, creating a wide landscape of computer architectures. In high performance computing for instance, it is common to schedule tasks on heterogeneous systems, that is systems composed of different flavours of processors such as CPUs/GPUs couples. Given that performance is intimately tied to each given architecture, it becomes difficult to properly produce efficient code.

In particular, cache memories themselves are different from one architecture to one another. This implies that memory accesses optimizations for one given architecture may impair performances on another given architecture. Memory accesses are typically performed on distinct memory sections – or layouts – with access patterns that have a direct impact on cache usage: regular coalesced accesses tend to improve cache usage while strided patterns tend to cause issues of performance. Moreover, the advent of SIMD units on general purpose processor (CPUs) allow further performance improvement if data layout accesses are contiguous. Layout design should in theory be adapted for any targeted architecture in order to reach high performances.

As a result, hardware is generally underutilized as complicated access patterns on data layouts do not optimally or near-optimally use the cache memories, and vectorization over SIMD units often fails for various reasons such as lack of contiguity in particular. To efficiently take advantage of computing resources, automated tools are required, both to allow programming time

minimization and to rapidly find near-optimal solutions.

A natural solution for automatic code optimizations would be compilers, which are equipped by state-of-the-art techniques to produce quality binaries for a quantity of target architectures. However compilers are facing numerous difficulties, particularly in the context of layout transformations that considerably hinder its capabilities and prevent it from reaching peak performance. Levels of abstractions in the code typically provide fewer semantics specificities useful for compiler optimizations; the order in which given compiler optimizations are applied is important; the evaluation of transformations, in particular for data layout transformations, relies on basic heuristics and theoretical models that are considerably inaccurate; the lack of information at compile time on data layouts in general, in particular pointer expressions and indirections are challenging. Besides, compilers by themselves do not make proper advancement in phase with architectures advancement. As a result, the performance gap between compiled application code and expert programmer code [93] is significant. Finally, compilers also provide no feedback as to why data restructuring is not applied, what are the actual issues and how to resolve them.

Contributions

Now there is another class of programs commonly referred to as tools that are used to specifically pinpoint issues in applications, in order for the programmer to modify the code to optimize for performance. MAQAO [8] and TAU [98] for instance are such performance profiling tools. However, most tools consist in mere problem pinpointing, no optimizations are suggested to the programmer to improve his code, leaving the programmer the burden of inventing and testing layout restructuring strategies by hand and for each targeted architecture. Assessing transformations benefit beforehand would represent valuable information for the programmer, to know which transformation is opportune and how much gain is expected. Tools need more sophistication to address this topic, in particular quality user feedback has to be provided in order to tackle the data layout restructuring issue.

To address compilers limitations relative to the layout issues and the lack of feedback from both compilers and tools, we propose a novel approach to data restructuring; that consists in using the user application binary and collecting memory traces from said binary to analyse, transform, and evaluate new layout possibilities; and we make the following contributions for that matter:

- **Data layout formalism**. Data layouts are defined not only by the layout itself but also by the layout access patterns. Such a complete definition is particularly verbose in the code, especially when the number of access patterns is significant. We propose a concise formalization of layout/patterns, which eases not only discussions throughout this document and discussions with user via elaborate feedback, but also allows systematic expression of layout transformations. This contribution was published in [38].

- **Initial layout normal form**. Layout restructuring implies analysing the initial suboptimal layout in the first place, then to express it using our data layout formalism, in the aim of exploring transformations on it. Analysis consists in delinearization of the initial layout, based on the flat layout definition as given by memory traces of the user application, in order to transform the flat accesses definition into a multidimensional entity with the particular property of constituting a normal form of the layout. This means in particular that the initial layout expression in our formalism is independent of any compiler optimizations

on memory accesses such as unrolling for instance, which creates multiple instructions for a single initial access instruction.

- **In-vivo evaluation**. Given the importance of providing quality user feedback, an evaluation technique is inevitable to quantify each transformation potential benefit. Considering compilers static models and simulation approaches are typically inaccurate and incomplete, we propose to evaluate transformations *in-vivo* which means in the context of the user application run, as to obtain a realistic performance estimation. The main idea is to perform a checkpoint at the application kernel of interest, and then re-executing the kernel with different layout flavours using mock-up codes. This contribution was published in [39]

- **SIMD detection and application**. The recent introduction of SIMD units in general purpose processors implies a significant performance gain to obtain from them. All kernels are not necessarily candidates for vectorization over SIMD units, in particular, dependences issues may hinder vectorization. Therefore, determining intrinsic kernel vectorizability is essential, and our SIMD detection technique was published in [6]. Furthermore, for all vectorizable kernels we implement SIMDization in our evaluation codes, as well as transformations that are designed to take advantage of this particular feature.

These contributions provide the application programmer with elaborate feedback, that is firstly providing a performance issues report pinpointing data layout issues as a whole and also instruction by instruction. Second, it consists in proposing identified transformations strategies, helping copying the data from the initial to the new layout as well as changing the mapping of each individual memory instructions. Third, the evaluation step permits to quantify every single restructuring strategy to allow the user to make an enlightened choice of restructuring. This approach has several benefits. First of all, unlike many optimizations that are architecture-dependent and obfuscate the code, our transformations suggestions simply alter the layout mapping. Second, layout restructuring is intrinsically independent of control flow optimizations, therefore it can be performed alongside any control transformations and has no negative impact on them. Finally, the approach is not architecture-dependent and can be implemented on any system, the evaluation step allows to measure actual layout performance encompassing fluctuations due to specific architectural effects.

Outline

Chapter 1 presents the context and problem that leads to the need of data layout restructuring. Chapter 2 explains how to determine kernel vectorizability from both static and dynamic analysis of the application binary. Chapter 3 shows the steps to reconstruct the application original problematic layout from memory traces into a formal representation. Chapter 4 presents the transformation space exploration heuristic, manipulating the layout formal representation. Chapter 5 explains how to rewrite the code based on new layout mappings selected by the heuristic. It allows to generate evaluation code versions as well as user feedback. Chapter 6 details the data restructuring evaluation methodology and experimental results. Finally, the last chapter concludes this work and presents perspectives and future challenges.

Chapter 1

Context and Problem

1.1 The memory layout/pattern issue . . . 16
1.1.1 Modern architectures complexity . . . 16
1.1.2 The locality issue . . . 18
1.1.3 Layout abstractions . . . 22
1.2 The complicated relationship between compilers and data layouts . . . 23
1.2.1 Static approaches . . . 24
1.2.2 Dynamic approaches . . . 25
1.2.3 Data restructuring techniques . . . 27
1.2.4 Vectorization . . . 28
1.3 The lack of user feedback . . . 29
1.3.1 Compiler feedback . . . 29
1.3.2 Tools . . . 31
1.3.3 The importance of quantification . . . 32
1.4 Towards a data layout restructuring framework proposal . . . 32
1.4.1 Proposition overview . . . 33
1.4.2 Data restructuring evaluation . . . 35

Nowadays computers architectures are complex, numerous, and differ noticeably from each another. Thus, the codes and especially the memory accesses to data layouts must be greatly architecture-dependent in order to achieve high performance. The notion of good layout is vague to programmers, as there is no such thing as performance portability. Plus, different layers of data structures abstractions form a gap between the logical and the physical layout, so writing efficient code has an unacceptable complexity for the user, therefore an automated way is needed. Given compilers limits in the context of data restructuring and the lack of feedback from both compilers and tools, we propose a framework aiming at guiding the programmer through his layout restructuring process. First, we pinpoint the layout issues with a comprehensive report, then we propose layout transformations with both a high-level representation and a precise instruction-level modification report. Finally, we quantify each of the transformations in terms of performance gain, in order to direct the user towards the potentially most profitable layout.

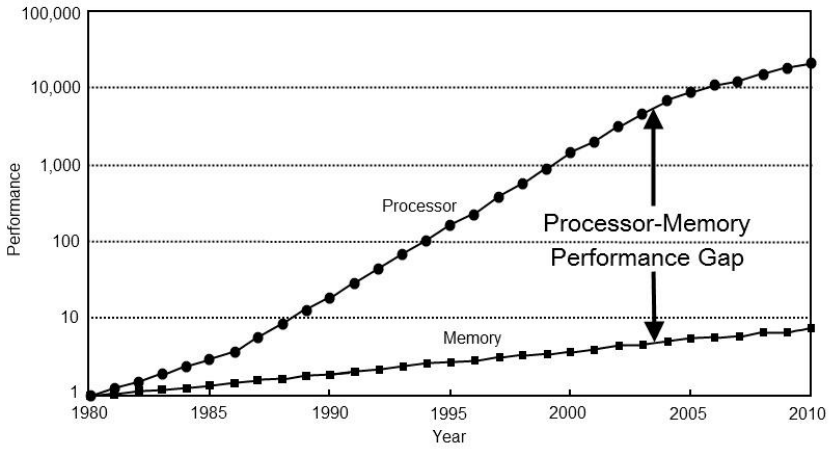


Figure 1.1: Memory performance versus CPU performance [82]

1.1 The memory layout/pattern issue

1.1.1 Modern architectures complexity

In recent years, a lot a researching efforts has been devoted to computer architecture design to address the ever-growing computational need. Semiconductor manufacturers, while watching clock speed reach a plateau, has repeatedly applied transistor miniaturization, one of the main vectors of processor design, increasing single processor complexity to keep allowing to obtain better performance this way. This interest has made processor performance evolution grow linearly following Moore's law [75]. The trend is that the number of transistors on a microprocessor chip doubles every 18 months, making architectural design specificities change on a frequent regular basis. However, memory technology development has not seen quite a growth as shown by Figure 1.1, in fact since the 1980s the performance gap between processors and memory has aggravated steadily. Current memory technology is intrinsically not able to keep up with processor performance as first discussed in [116], urging to find architectural solutions to circumvent the so-called memory wall: memory performance limitations mask the significant performance improvements made on processors, as the codes are slowed down because processors spend too much time stalled by memory requests taking too long to complete.

Memory has been forced off of the microprocessor given its growing size/space requirements, which induces increased latency with respect to processor registers. This physical constraint has been addressed by the introduction of the cache memory to the microarchitecture, creating a memory hierarchy from the registers to the main memory. It allows frequently accessed data to be closer to the processor and thus to improve latencies. When a data element is requested by the processor, the cache is first checked for the presence of said element. If it actually is present, then the latency to fetch the element is lesser than direct main memory access. If it is not, then the element is fetched from the main memory and brought back to the cache, in place of older data, speculating it may be used again (temporal locality) in a close future. Given the success of cache memory, this approach has been further developed to continue optimizing memory access performance. Today,

different levels of cache are commonly found in most microprocessors.

Caches take advantage of the so-called spatial locality of data accesses to improve memory usage. Indeed, given the memory is far from the processor, caches request cache blocks – or cache lines – of memory elements to the ram instead of single elements, under the speculation that consecutive elements may be used in a recent future, as it is the case for instance for large arrays or vector accesses. Cache line size is typically 64 bytes on x86 CPUs architecture like the Sandy Bridge for instance, which corresponds to 16 single precision floating point elements or again 8 double precision elements, that is an arguably small number of element and on typical compute intensive kernels operating on large arrays, one may be tempted to pre-emptively load even larger blocks of memory into the cache to further optimize memory access performance. In turn, both hardware and software prefetch are implemented to anticipate even further the spatial locality property of data accesses, by performing multiple cache line prefetch. Consequently, performing contiguous accesses is essential, as the hardware is designed to take advantage of spatial locality to cover for the inherent limits of the memory technology to sufficiently rapidly provide the processor with data.

Because it has been assigned the difficult task of bridging the gap between respectively processor and memory performances, cache design choices has a direct impact on applications memory access performance. For instance, the respective cache memory level size criterion alone makes performance vary for a single given application, as the application datasets tendancy to fit entirely or almost entirely in the cache can greatly maximize its performance. Although, for any dataset size especially significantly large datasets, blocking can help considering block sizes fitting the cache memory. It consists in principle to arranging the memory accesses such as temporal and spatial locality in particular are maximized. The idea is that the blocks loaded from the main memory into the cache memory has to be filled with elements that all need to be used quickly by the program , so they do not get evicted from the cache before they are used, provoking cache misses and forced to be reloaded. To ensure a good temporal locality property, the element we need to access several times has to be quickly re-accessed so it does not get evicted from the cache. Cache eviction are indeed ruled by a Least Recently Used (LRU) policy in most modern architectures, which states that the least recently used cache line is substituted by a newly requested cache line on a cache miss.

Another cache design choice concerns the respective cache levels associativity. Indeed, within the cache, memory references are segregated in different sets, which prevents too many inopportune evictions. Each of the set contains n blocks where n is the given number in n-associativity, which is commonly varying from 4 to 16 given the architectures and the cache level. For instance the Intel Sandy Bridge L1 and L2 caches are 8-way associative, while the L3 is 16-way associative, whereas the AMD Bulldozer L1 cache is 4-way associative and its L2 cache is 16-way associative. The number of blocks in each set allows a leeway in the context of eviction policies, as an entry least recently used among the n blocks , so intensive memory accesses do no thwart each another. Nevertheless, with a significant number of array with respect to n, which is dependent of the application, evictions can potentially happen more often as well as cache misses, sinking the application performance. As a result, because of varying implementations of cache designs

considering cache size, associativity, energy consumption , latencies and other important parameters, programmers cannot assume their application data layout performance is constant across different architectures: performance is not portable. Also, taking into account these parameters when writing efficient code has most certainly a great impact on application performance.

In the beginning of the 2000s, the first multi-core processors appeared with the IBM Power 4. having several cores on a single chip permits to increase processors performance while avoiding raising the processor frequency and benefiting from thread-level parallelism as many applications are able to take advantage from it. In this configuration, typically, the L1 cache and sometimes the L2 cache are private to each core, as if we considered two separate processors. However, the Last-Level Cache (LLC), typically the L3 cache, and sometimes the L2, is often shared among the cores on the same chip, causing contention issues. Even considering a simple case where threads, each placed on a different core, do not access at all the same pieces of data. First of all, the simple fact of sharing a cache means that a cache miss caused by one given thread may evict a cache line loaded by another thread. Second of all, we observe what is called true sharing effects when two threads on two separate cores share the same element on the same cache line, a synchronization step has to happen between the two concerned cores, to ensure validity of the data. Third of all, false sharing happen when two threads on different cores access different data elements of the same cache line. The first thread that modifies its data in said cache line triggers cache coherency protocols that proceeds to invalidate the cache line from the other threads core respective caches, forcing them to reload the wanted piece of data from the main memory. The second thread, once it modifies the data on the same cache line, invalidates in turn the same cache line on other cores caches, and so on. Even though spatial/temporal locality prevails when it comes to data layout performance, false sharing may in some cases interfere as it is shown in [105].

Nowadays microprocessors feature single Instruction Multiple Data (SIMD) vector units, potentially providing substantial performance improvement by concurrently applying the same instruction to all the elements of a vector. A rich and complex API of SIMD instructions has been developed on multiple architectures (such as AVX for Intel or NEON for ARM). Thus, the performance of a code is highly dependent on the use of the SIMD instructions, assuming the algorithm is vectorizable. Efficient vectorization, or SIMDization, is usually achieved over contiguous arrays, as it involves contiguous small vectors to be loaded and to be applied arithmetic operations without performance overhead. In a sense, the use of SIMD rewards the spatial locality aspect. To load more efficiently packed data, many ISA impose the base pointer to be aligned on cache line size boundary for aligned access, unaligned accesses can also be performed with special instructions at cost of a performance penalty.

1.1.2 The locality issue

Because the layouts spatial locality is determinant for application performance and intrinsically dependent of the underlying architecture, the programmer has an important responsibility in his application code design. Application code algorithmic needs to impose the creation of elaborate data structures, expressed by array and structure types often combined, and complicated patterns expressed by loop nests iterations. A good spatial locality is a consequence of simple patterns that skim through the arrays in a contiguous manner by accessing the arrays element consecutively.

```
1 float a[n][n], b[n][n], c[n][n];
2
3 for (i = 0; i < n; i++) {
4     for (j = 0; j < n; j++) {
5         a[i][j] += b[i][j] * c[j][i];
6 }}
```

Figure 1.2: spatial locality issue caused by a strided pattern on c array, while the layout c is allocated in a simple contiguous fashion

Now, if a given layout is a contiguous array, non-contiguous access patterns alter spatial locality and cause issues of performance. Non-contiguous accesses with fixed stride are also called strided accesses, the stride being the distance between the element accessed at a given iteration step $a[i]$ and the element accessed at the next iteration step $a[i+1]$. In fact, large strides may imply performance issues. If $a[i]$ and $a[i+1]$ are distant from a stride $s > 1$ and they are on the same cache line, there is no issue of performance additional to alignment and SIMD consideration. Most SIMD instructions require memory accessed to be aligned on a 64-byte boundary and the data to be packed, which means there must not be non-unit stride between elements or a penalty occurs. In this cas, multiple loads may need to be performed and additional assembly instructions need to be added in order to form a packed vector, that is a vector full of contiguous elements. If $a[i]$ and $a[i+1]$ are located on different cache lines, other penalty occur as the cache has to find the other cache line, and possibly to fetch it from the main memory. consequently, if the stride is at least one cache line long, an access to the main memory is required for each distinct access on the array. Similarly, the larger the stride, the more TLB misses which are costly in terms of cycles. Translation Look-aside Buffer (TLB) misses occur when the requested data elements is located on a memory page not currently mapped by the TLB. This causes extra performance penalty as the right physical page address need to be mapped on a virtual address and the corresponding record entry has to be added into the TLB. Additionally, these potential issues of performance due to poor spatial locality may have another serious downside if the elements located in-between $a[i]$ and $a[i+1]$ are not consumed quickly by the computations. Indeed, in the meantime, the corresponding cache line or page entry may be evicted from their respective caches and causes systematic cache misses or TLB misses if applicable.

As an example, Figure 1.2 exhibits a C source code where stride issues occur. The three arrays a, b, c are correctly allocated in a contiguous manner. The given loop nest expresses patterns on these arrays: arrays a and b are properly accessed, there is no stride issue as each element is separated from the next element by a unit stride which means contiguous accesses. However, the innermost loop happens to navigate through the outer dimension of the array c. Because the memory space is linear, multidimensional structures are linearized on the memory space which means the next elements on the second dimension for instance are distant from a stride n here on the given example, which is the inner dimension size. Consequently, c array has a stride n between each access, implying a poor spatial locality and performance issues accentuated if n is significantly large.

The programmer is also in charge of the data structures choice, which is influenced by the application algorithms he implemented. Among the classical data structures, the struct type is an object constituted of a collection of elements with a different logical meaning, and is typically ac-

cessed by distinct instructions on distinct fields, which are the structure elements. This implies each instruction accessing an array of structures has an intrinsic stride that is the size of the structure, which has several implications on the layout performance. Firstly, if there are unused fields, that is fields that are not accessed by a given fragment, or cold fields, which are fields that are less intensively used than others, then performance issues due to bad spatial locality may arise from the strided accesses provoked by the structure in the case where certain fields are unused, since they may be brought to the cache uselessly.

Secondly, even if all the fields are hot, that is they all are intensively accessed, then the structure achieves good spatial locality, because the element are all quickly used and the structures are accessed in a contiguous manner. However, the nature of structure access prevents efficient vectorization. Indeed, the efficient use of SIMD instructions requires packed load, so the elements has to be consecutive in memory which is not the case for structure fields which are interleaved, therefore on an Array of Structure (AoS) access, the next field access is distant from a stride equal to the structure size. Also, because field accesses are typically performed by distinct instructions, it may be tempting to use different threads to handle them, however that would not be efficient if different fields are on the same cache line which is the case if the structure elements are typically single elements such as single or double precision floating point elements, because false sharing would occur.

For these reasons, Structures of Arrays (SoA) may be preferred. In this scenario, single instructions access distinct contiguous arrays allowing allegedly good spatial locality, and consequently allowing packed loading and storing and efficient vectorization if applicable. Nevertheless, in the case of large arrays inside the SoA, additional TLB misses occur, especially if there is a large number of structure fields, which causes and implies a not so good spatial locality property. In the general case, it is not obvious if the performance gain due to vectorization is enough to mask TLB misses penalties. There is a way of shaping the data structure to benefit theoretically of both good spatial locality and vectorization, that is primarily thought for targeting general CPUs/Manycores with SIMD extensions and referred to as Arrays of Structures of Arrays (AoSoA). AoSoAs construction consists in dividing the higher dimension in the innermost dimension, such as a few consecutive elements belonging to the same field become contiguous, typically it is chosen to represent a size of the vector size so as to enable vectorization. As a result since the vector fields are accessed at once among the instruction accessing several fields, there is a good spatial locality as well as all elements are used quickly enough not to be frequently evicted from the cache memory.

Figure 1.6 shows code examples highlighting the different patterns caused by 3 different structure topology. Any access on the structure is described by a different pattern, therefore the whole application accesses; that means both the individual access instructions and the loops nests; are different from one implementation to the other. Thus, passing from one data structure to the other necessitates a considerable programming effort.

Consequently, the patterns are dependent of the layouts shape or topology; the topology is a property of the layout and not the pattern, and is therefore independent of scheduling considerations. Multidimensional structures intrinsically impose additional patterns issues as shown by Figure 1.7. Here, the array $datarr$ is 4-dimensional, where the two innermost dimensions are actually either explicit structure fields or at least analogous to structure fields, as each instruction access a distinct structure field. There are several spatial locality issues here. First of all, the innermost structure-like dimension has one hot field out of two total fields, producing a useless stride

```
1 struct s {float a, b, c, d;};
2 struct s A[N];
3
4 for(i = 0; i < N; i++)  {
5     A[i].b = ...
6 }
```

Figure 1.3: Array of Structures (AoS)

```
1 struct s {float a[N], b[N], c[N], d[N];};
2 struct s A;
3
4 for(i = 0; i < N; i++)  {
5     A.b[i] = ...
6 }
```

Figure 1.4: Structure of Arrays (SoA)

```
1 struct s {float a[p], b[p], c[p], d[p];};
2 struct s A[N/p]; // assuming N%p==0
3
4 for(i = 0; i < N/p; i++)  {
5   for(j = 0; j < p; j++) {
6     A[i].b[j] = ...
7 }}
```

Figure 1.5: Array of Structures of Arrays (AoSoA)

Figure 1.6: 3 data layouts versions of A implementing classical ways of handling structures. Choosing the best implementation with the best parameters p is not straightforward as their respective performance is highly dependent on the architecture.

```
1  for(Xstep = 1;Xstep<Nx+1;Xstep++)
2  {
3      for (Ystep = 1; Ystep < Ny+1; Ystep++)
4      {
5          ...
6          Vm=datarr[Xstep][Ystep][0][(step-1)%2];
7          dVmdt=datarr[Xstep][Ystep][1][(step-1)%2];
8          IK1=datarr[Xstep][Ystep][2][(step-1)%2];
9          x1=datarr[Xstep][Ystep][4][(step-1)%2];
10         INa=datarr[Xstep][Ystep][5][(step-1)%2];
11         m=datarr[Xstep][Ystep][6][(step-1)%2];
12         h=datarr[Xstep][Ystep][7][(step-1)%2];
13         Is=datarr[Xstep][Ystep][8][(step-1)%2];
14         d=datarr[Xstep][Ystep][9][(step-1)%2];
15         f=datarr[Xstep][Ystep][10][(step-1)%2];
16         Cai=datarr[Xstep][Ystep][11][(step-1)%2];
17         Isum=datarr[Xstep][Ystep][12][(step-1)%2];
18         Diff=datarr[Xstep][Ystep][13][(step%2)];
19         Istim=datarr[Xstep][Ystep][14][(step%2)];
20         ...
```

Figure 1.7: Example of Cardiac wave simulation [112]. The 4-dimensional array `datarr` is actually used as an array of structures and imposes complex patterns with sub-optimal spatial locality.

of 2 elements. Secondly, the second innermost dimension is composed of constant fields, while the two outermost dimensions are processed by the nested loop, implying a stride equal to the product of the two first structures size between consecutive elements which hinders vectorization. While there are intuitively certain ways of optimizing this particular layout, the application context has to be considered as the layout may be used in other intensive nested loops. Also whether to choose between AoS-like or SoA-like or AoSoA-like transformations is not straightforward and depends on the underlying architecture, therefore choosing the best restructuring is difficult. This code excerpt is part of a real life application, namely cardiac wave simulation published in [112]. Its optimization is further discussed in this document.

1.1.3 Layout abstractions

In order to alleviate the programmer the task of explicitly writing optimal code, especially when it comes to maximizing data layouts performance for any targeted architecture, numerous abstraction layers have been proposed. In fact many languages, in particular Object-Oriented languages propose so-called abstract data types, as opposed to data structures, that allow logical representation of the layout by behavior rather than by physical aspect as for data structures. This behavior is defined by the application of high-level functions to iterate, access and manipulate data layouts without explicitly referencing any physical layout. This abstraction layer is also provided by libraries, hiding in particular the complexity of AoSoA layouts with SIMDization to the user, such as ASX [100] or Cyme [32].

GPGPU programming model is unusual for most programmers, one of the most important difference with CPU programming is that memory accesses have to be coalesced in order to get the most of the significantly fine-grained thread-level parallelism, therefore AoS to SoA transformation need to be expressed in order to pass from the host (CPU) to the device (GPU), as SoA layouts tend to naturally take advantage of the GPUs. The Thrust CUDA library [9] allows such an abstraction, providing functions and iterators to transform and process the SoA layout on the device. The programming of systems composed of both CPUs and GPUs is also known as heterogeneous programming and is challenging for many reasons. One of them is that the heterogeneous context may imply transferring data between devices, in particular between CPUs and GPUs that require differently shaped layouts in order to achieve high performance. The Dymaxion++ API [17] proposes the programmer to annotate his code with pragmas in order to optimize memory patterns by remapping the data especially from the CPU main memory onto the GPU main memory and also from the GPU main memory to the GPU scratchpad memory. The remapping is performed on the GPU, the API proposes a few simple transformations such as the transpose and non-unit stride removal. Similarly, both DL [102] and Kokkos [27] proposes layout transformations in heterogeneous context, and addresses the AoS to SoA/AoSoA transformations, with Kokkos addressing manycore architectures as well.

Explicit SIMD programming resorts to low-level function (*eg* Intel Intrinsics) close to the actual SIMD assembly instructions, making them both impractical to use and typically not portable. To mask the difficulty of SIMD programming to the users, many abstractions layers have been proposed to cope with this matter. The OpenMP shared memory multiprocessing API have recently released OpenMP 4.0 SIMD [104]. similarly to the original pragmas commending multithreaded

```
1 for (k = 0; k < N; k++) {
2     for (i = 0; i < N; i++) {
3         for (j = 0; j < N; j++) {
4             ...
5                 a[k] += a[k]*c[j][i]
6                         + b[k];
7 } } }
```

Figure 1.8: Kernel with spatial locality issues. (Synthetic benchmark)

```
1 for (k = 0; k < N; k++) {
2
3     for (j = 0; j < N; j++) {
4         for (i = 0; i < N; i++) {
5             ...
6             a[k] += a[k]*c[j][i]
7                     + b[k];
8 } } }
```

Figure 1.9: Loop interchange

```
1 for (k = 0; k < N; k++) {
2     transpose(t,c);
3     for (j = 0; j < N; j++) {
4         for (i = 0; i < N; i++) {
5             ...
6             a[k] += a[k]*t[i][j]
7                     + b[k];
8 } } }
```

Figure 1.10: Data restructuring

Figure 1.11: Loop interchange or data restructuring? The difficulty of performing proper data restructuring evaluation on the compiler side prevents it from making the most profitable decision, by often choosing control optimizations over both data restructuring opportunities and combined loop and data transformations opportunities.

execution, OpenMP 4.0 uses pragmas clauses to guide loop SIMDization, independently of the underlying architecture. Meanwhile Intel has proposed Intel Cilk Plus SIMD, that feature C/C++ languages extensions to express vectorization. That includes a semicolon notation operator, which much like in Fortran language allows to specify array sections and allows single code line vector operations, that is without using a loop structure just like in Fortran. Another advantage of this notation is that it indicates the compiler potential vectorization opportunities. The Boost.SIMD C++ library [29] allows SIMD instructions abstraction with higher-level functions, independently of the ISA, saving the programmer the effort of rewriting low-level code at each ISA change. The library provides hundreds of already vectorized mathematical functions. Cyme [32] library uses abstract data types to create an AoSoA layout out of AoS to enable efficient vectorization, sparing the user a tedious code rewriting to perform the AoSoA transformation. Additionally, both semiconductor manufacturers Intel and AMD propose math libraries with vectorized functions with respectively the Intel MKL [46] and the AMD ACML [5] . The Intel MKL library for instance disposes of pre-vectorized BLAS, LAPACK and FFT, that are composed of many intensively used functions.

1.2 The complicated relationship between compilers and data layouts

While they offer numerous optimizations and particularly loop transformations, compilers do not sufficiently explore data restructuring, in the aim of maximizing spatial locality and performances. Figure 1.11 illustrates one of the challenges of data restructuring: should only control transformations be performed, should only data transformations be performed, or both?

```
1 for (i1 = 0; i1 < N; i1++) {
2     for (i0 = 0; i0 < N0; i0++) {
3         A[i0][i1] = ...
4 } }
```

Figure 1.12: Transpose case

1.2.1 Static approaches

Numerous loop transformations techniques have been proposed. The idea is to optimize spatial locality by modifying the loop nest and therefore the access pattern, without altering the data structure. The transformation is consequently local to the loop nest, whereas data structure transformations would require complicated global analysis, as the transformation gain has to be assessed for the whole application since the data structure is accessed in more than only one loop most of the time.

Early works like the one done by Carr *et al.* in [15] proposed to integrate such loop transformations in compilers, particularly focused on spatial and temporal locality by minimizing accesses to the main memory. A set of simple transformations such as loop permutation, loop reversal, loop fusion and distribution are studied. Loop permutation for instance is the transformation that allows switching between row-major and column-major in particular. The transformations are chosen by a cost model based on computing each of the loops cost in terms of number of cache lines if they were in the innermost position. As it has been argued that loop transformations are well understood and improve temporal locality as opposed to data layout transformations, few works have been focused exclusively on array restructuring. Nevertheless a first benefit of the approach is that it is intrinsically independent of loop control considerations. Leung *et al.* [61] propose to optimize data locality by array restructuring via index transformation matrices. The proposed cost model has only a local view, global application benefit is not assessed, which is a common difficulty with data transformation.

To the best of our knowledge, Ciernak *et al.* [21] were among the first to propose assessing unifying data layout and control transformations. However, optimizing for both control and data is difficult as transformation space search is expensive, especially for an arbitrary high amount of loops and arrays to assess, the authors thus propose a heuristic to try to cope with the issue, with a pre-selected pool of transformations. The formalism to express loop and data transformations is quite similar. The stride vector, expressing the strides between elements accesses is retrieved statically, the idea is to push a loop with stride 1 on the inner part of the loop nest. Loop nest transformations are formally described by an algebraic representation.

Thus, simple loop transformations such as loop inversion can be expressed this way:

$$\begin{pmatrix} i1' \\ i0' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i0 \\ i1 \end{pmatrix} \tag{1.1}$$

Where the input stride vector $\begin{pmatrix} i0 \\ i1 \end{pmatrix}$ is derived from the original loop nest given in Figure 1.12

In an analogous manner to Equation 1.1, simple layout transformations can also be expressed with the same formalism:

$$T_A = TA \iff \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i1' \\ i0' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i0 \\ i1 \end{pmatrix}$$

Where A is the initial access matrix and T_A is the restructured array after performing the transpose by applying the transformation matrix T.

Kandemir *et al.* [49] are under the same constraint as [21] as detailed before on the difficulty of finding the best unified transformations between control and data layout and developed a different heuristic for that matter. One of the problems is that layout transformation itself has to satisfy multiple nests, the heuristic focuses on optimizing the nest that takes the most time, and transform the other accordingly. They also propose other choices of loop nests to optimize, it can be chosen by profiling or user-specified. In [48] the authors reinforced the control/data transformations study by adding interprocedural locality optimization, using the call graph to propagate the cost model results. [26] proposes a new purely static mathematical model to optimize locality with focus on multithread contention limitation.

The polyhedral model introduced in [33] intended to leverage static loops transformations with a new formalism. Although controversial mostly in reason of the expensive cost on the compiler time, polytope model implementations have found their way to compilers such as GCC [107] and LLVM [36]. Polyhedral model expresses nested loop transformations, where both iteration domain and dependences are represented by Z-polyhedra, allowing convenient code validity verification for instance. Transformations apply on the iteration domain D defined by a Z-polyhedron as loop counters are integer:

$$D = \{x \mid \mathcal{D}x + d \geq 0\}$$

For instance, Loechner *et al.* [65] implemented the polyhedral model to improve the locality of nested loops, with a special focus on avoiding TLB misses, as opposed to most similar works where focus is on regular cache misses only.

Although the polyhedral model is initially intended for control transformations, Lu *et al.* [66] proposed to assess locality via polyhedral analysis identifying potential for data transformation, with a special focus on inter-thread contention to target CMP (Chip Multi-Processors). Two of the studied transformations are the strip-mining and the permutation. Strip-mining consists in reshaping for instance an array dimension with size $n \times p$ into a 2-dimensional array with dimensions n and p respectively, analogously to loop tiling. Permutation transformation consists in permuting array dimensions, similarly to the row-major to column-major transformation. Choice of transformation is driven by the polyhedral analysis. Other works also are based on the polyhedral model power for data layout transformations such as [63].

1.2.2 Dynamic approaches

Compilers are facing challenges difficult to overcome. In particular as far as data restructuring is concerned, it has been shown [85] that optimal data placement on a partitioned cache is NP-hard and poorly approximable. Therefore, building an accurate cache performance model is not simple. Moreover, the optimization scope is the entire program source code, as said before finding the optimal restructuring strategy given an arbitrary number of loops and arrays is also difficult. Nevertheless, as shown in the previous section, a lot of research effort on static analysis has been focused on proposing heuristics to cope with these challenges. However, most layout optimizations are still left unexplored by compilers, as supplementary hurdles limit their power. First of all, the lack of information on memory dependences at compile time is challenging, indeed inferring

```
1  for(iL=0 ; iL < L/2 ; iL+=1) {
2    for (j=0;j<4;j++) {
3      r0 =  U[idn[4*iL]][0]*tmp[j];
4      r0 += U[idn[4*iL]][1]*tmp[n2+j];
5      r0 += U[idn[4*iL]][2]*tmp[2*n2+j];
6      r1 =  U[idn[4*iL]][3]*tmp[j];
7      r1 += U[idn[4*iL]][4]*tmp[n2+j];
8      r1 += U[idn[4*iL]][5]*tmp[2*n2+j];
9      r2 =  U[idn[4*iL]][6]*tmp[j];
10     r2 += U[idn[4*iL]][7]*tmp[n2+j];
11     r2 += U[idn[4*iL]][8]*tmp[2*n2+j];
12     ID2[j] += r0 ;
13     ID2[n2+j] += r1;
14     ID2[2*n2+j] += r2;
15   }
```

Figure 1.13: Example of Lattice QCD simulation. The 2D array U uses an indirection. All elements are complex double values. The space iterated by the outer loop is a 4-D space, linearized, and the indirection is used to walk through the white elements of a 4-D checkerboard.

memory behavior is risky and in general compiler conservativeness prevents it from performing most restructuring strategies to avoid jeopardizing program correctness. Similarly, pointer references in general are difficult to handle. The presence of aliasing for instance limits compilers capabilities. Also all data structures referenced by pointers in general are hard to recognize and optimize, namely arrays of pointers and classic trees, graphs and lists. Indirections happen when arrays indexed by another arrays elements, which is the case in sample code given in Figure 1.13 for the U array that is indexed by the idn array. This is typically ignored by compilers. Libraries loaded by the program are out of the compilers reach, that implies compilers cannot take library routines into consideration when exploring transformations. Compilers also still fail for a lot of cases involving complicated index computation and induction variables.

Nevertheless, it is possible to assist the compiler with profiling. It consists in dynamically assessing certain program behavior, often on a given program portion rather than the whole program because the approach can be costly in terms profiling time. This dynamic approach is often used at complementing compiler analysis, rather than in a standalone way where the optimization scope is limited and lacks of general program semantics. Profiling is useful in several ways, such as information gathering for instance for memory accesses, patterns can be established from running and analysing some hot functions in the program. Many works take advantage of dynamic informations in addition to compiler optimizations. Ding *et al* [24] perform array restructuring at runtime, with two different transformations that are locality grouping and data packing that improve temporal locality in particular. Locality grouping reorder computations by array affinity, while data packing is a data layout restructuring technique that allows to gather elements with close temporal locality into the same cache line. Shen *et al.* [97] propose an affinity analysis technique, which consists in regrouping in memory arrays that are always accessed together in the program. Authors argue symbolic analysis by the compiler suffice and no profiling is really needed. However more complex codes with complicated pointer references may challenge such static analysis. Cho *et al.* [20] use profile information on memory access patterns to effectively use the scratchpad memory.

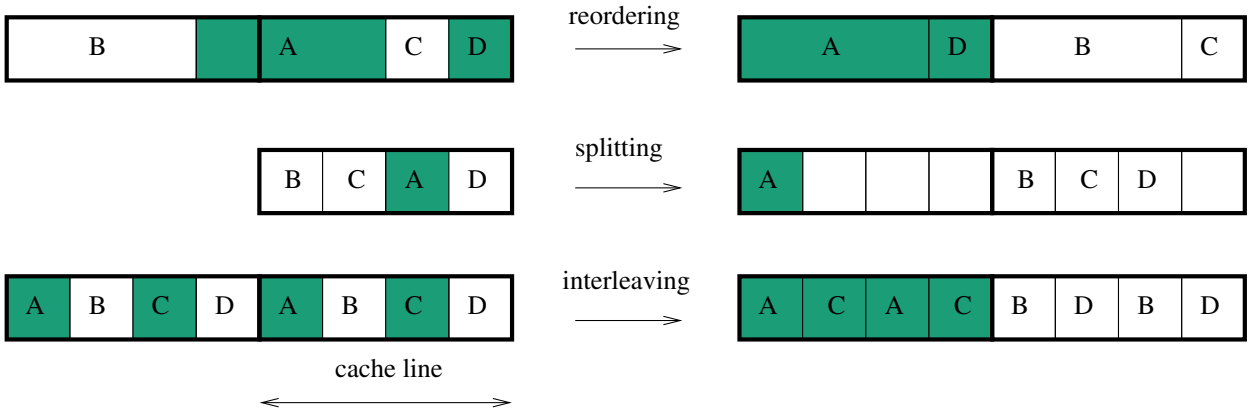


Figure 1.14: Classical cache-conscious restructuring techniques – Common aim is to segregate hot (green) and cold fields

Another approach to improve compiler optimizations is the so-called iterative compilation [11] [53] . It consists in exploring the compiler optimization space by iteratively compiling and measuring the effectiveness of given optimizations. One of the approach major concerns is that it is time consuming, as a result a lot of works focus on one particular aspect to optimize.

1.2.3 Data restructuring techniques

Several well-known strategies have been studied for array restructuring in the aim of cache utilization improvement, or cache-conscious strategies. In this context, profiling is an advantage as it helps characterizing the hot and the cold fields of a structure. Hot fields are typically the most intensively accessed as opposed to the cold fields, which interfere with the hot fields by various cache effects.

Field reordering When data structures are larger than the cache line, reordering structure fields [18] [52] [108] may allow better affinity between fields, that is to segregate hot fields and cold fields on different cache lines if applicable. Indeed, initially in the programmer source code fields are usually ordered logically, therefore profiling information on locality is useful to order them rather by frequency of usage in order to limit cache pollution.

Class Splitting Class splitting [18] is a technique similar to field reordering, the difference lies in the fact the class splitting targets structures that are typically smaller than cache line size. Some padding for instance can separate hot from cold fields, so as to segregate hot fields from cold fields.

Field interleaving Field interleaving [108] is an alternative to class splitting, that proposes filling the cache line with interleaved hot field instances instead of padding, as depicted in Figure 1.14.

Coloring Depending on associativity, an arbitrary small number of concurrent accesses on the same cache set can occur without generating conflicts. However when the number of array refer-

ences is significant, conflicts can greatly affect the code performance. Based on cache associativity knowledge, the idea of coloring [19] is to segregate – by color/temperature – hot and cold fields so as to partition the virtual space in a way that maps hot fields in the same cache set, and the cold fields in another cache set so as to avoid hot field cache lines eviction caused by substitution by a cold field. Padding is employed if necessary to properly partition the data structure on the virtual space.

Clustering Clustering [19] [69] is a technique that consists in the dynamic remapping of recursive data structures such as linked lists for instance, to improve spatial locality by optimizing cache utilization by packing data fields in a cache line. The aim is to gather two temporarily close nodes on the same cache line, that is the case for two consecutive elements via the "next" pointer for instance. Also by properly packing cache line, less cache lines are needed and therefore it minimizes inopportune hot cache line replacement by the cache replacement policy. However, such structures have to be fixed or change very little, because of dynamic restructuring overhead.

Memory segmentation Memory segmentation [95] is a technique which sorts highly referenced objects hopefully on a small set of pages, from short lived objects and not highly references objects. This way the method allows to limit TLB misses. One drawback is the potential fragmentation of the heap.

1.2.4 Vectorization

With the advent of short vector SIMD instructions in modern processors, SIMDizability and automated SIMDization have been the topic of several research efforts. Indeed, making use of these SIMD units has early been recognized as key for performance [59]. Compilers such as from Intel, IBM or PGI, as well as GNU GCC [80, 79, 78, 107] have received much auto-vectorization effort to enable and extend their capabilities [56, 56, 28, 4]. Scout [58] has been designed to vectorize at a higher level, by translating scalar statements to vectorized statements using SIMD intrinsics. However, Malecki *et al.* showed in their SIMDization tests [73] on the TSVC suite that many potentially vectorizable constructs are left unaddressed by state of the art compilers, either because some known theoretical techniques have not yet been implemented, because no known theoretical techniques exist for codes with complex structure, or because the compiler must act conservatively as a consequence to a lack of available information at compile-time [83].

An hybrid compile-time/run-time approach is proposed by Nuzman *et al.* [77] using a two-step vectorization scheme. The compile-time step performs expensive analysis operations. The run-time, specialization step is performed by an embedded just-in-time compiler. This approach enables some degree of adaptiveness to the hardware at run-time. However, the run-time step does not alter the vectorizability status using dynamic dependency information. Adaptiveness is also explored by Park *et al.* [30] by guiding the application of optimizations in a predictive manner through a machine-learning approach, and by Tournavitis *et al.* [106] in a technique associating profile-driven auto-parallelization together with machine-learning.

Multiple approaches have been followed in this attempt to group isomorphic elementary computations together. Loop-level approaches have been explored for several decades. Back in 1992, Hanxleden and Kennedy [110] made proposals for balancing loop nest iterations over multiple lanes of a SIMD machine. MMX instruction sets and alike started to gather interest with works

such as Krall's [57] proposing to apply vectorization techniques to generate code for SIMD extensions. Larsen introduced the concept of Superword-Level Parallelism (SLP) [59] which groups isomorphic statements together, when potentially packable and executable in parallel. More recently Nuzman et al. [81] explored SIMDization at the outer-loop level. Much work has also been devoted into employing polyhedral methods [12]. Several works have been conducted to allow compilers to accept more complex code and data structures, such as alignment mismatch [28], flow-control [99], non-contiguous data accesses [79] or minimizing in-register permutations [90], for examples. SIMDizing in the context of irregular data structures is also being studied [89]. All these works have in common that they accept unmodified source code as input and they attempt to generate the best SIMDized binary code for the target hardware. They do not involve the programmer in their attempt to produce good SIMD code and usually give little information back to the programmer when their attempt fails.

1.3 The lack of user feedback

Compilers have a lot of limitations in the context of data layout optimizations, and progress in compiler advances is slow with respect to architectural advancesTherefore code stay sub-optimized as far as data structures are concerned. At this point, the user is left with little to no clue on how to address his performance issues. The use of automatic tools is needed to guide the programmer through performance optimizations, first to locate performance issues and second to perform solutions if applicable or at least hint them. Numerous tools have been developed over the years to help the programmer locating performance issues in the application code, but very few give programming recommendations and strategies for actual performance problem fixing

1.3.1 Compiler feedback

Compiler shortcomings in the context of data restructuring are difficult to address, as little to no feedback is given to the programmer as to understand compiler choices relative to data restructuring. Nevertheless, efforts has been done in this direction in the context of vectorization. Compilers often fail to vectorize a number of candidate kernels that are actually intrinsically vectorizable, and compiler provide a compilation option (eg `-ftree-vectorizer-verbose=[N]` with GCC, `-vec-report[N]` with ICC) to produce a vectorization report, usually a file containing information on the failure reasons.

```
1  Analyzing loop at tsc.c:838
2
3  tsc.c:838: note: misalign = 0 bytes of ref c[i_42]
4  tsc.c:838: note: misalign = 0 bytes of ref d[i_42]
5  tsc.c:838: note: misalign = 0 bytes of ref b[i_42]
6  tsc.c:838: note: misalign = 0 bytes of ref a[_9]
7  tsc.c:838: note: not consecutive access a[_9] = _25;
8
9  tsc.c:838: note: not vectorized: complicated access
       pattern.
10 tsc.c:838: note: bad data access.
11 }
```

```
1  for (int i = 0; i < LEN/2; i++) {
2      a[2*i] = c[i] * b[i] + d[i] * b[i] +
           c[i] * c[i] + d[i] * b[i] + d[i]
           * c[i];
3  }
```

Figure 1.15: GCC 4.8.1 vectorization report excerpt on TSVC s1111 benchmark

```
1  LOOP BEGIN at userroutine.c(77,2)
2    remark #15344: loop was not vectorized: vector dependence prevents
3  vectorization
4    remark #15346: vector dependence: assumed OUTPUT dependence between ID2 line
5  95 and ID2 line 95
6    remark #15346: vector dependence: assumed OUTPUT dependence between ID2 line
7  95 and ID2 line 95
8
9    LOOP BEGIN at userroutine.c(82,3)
10     remark #15516: loop was not vectorized: cost model has chosen vectorlength
11 of 1 -- maybe possible to override via pragma/directive with vectorlength
12 clause
13     remark #15387: vectorization support: scalar type occupies entire vector
14     remark #15475: --- begin vector loop cost summary ---
15     remark #15476: scalar loop cost: 499
16     remark #15477: vector loop cost: 213.000
17     remark #15478: estimated potential speedup: 2.180
18     remark #15488: --- end vector loop cost summary ---
19   LOOP END
20 LOOP END
```

Figure 1.16: ICC vectorization report excerpt on Lattice QCD simulation code (see Figure 1.13)

Typically, non-unit strided patterns is a common cause of vectorization failure as reported by GCC pictured in Figure 1.15. On this example, there is only one loop with mostly perfectly contiguous array accesses, except for one access on array a that is labelled a complicated access pattern. Two issues are at stake here. First of all, the lack of contiguity prevents efficient vectorization. Second, assuming the array base address is aligned on a vector boundary, and given the stride 2, half of the elements are unaligned with respect to vector boundaries, which among others prevents packed loads.

In principle, it is possible to vectorize using partial load assembly instructions, gathering scalars in a single vector by using additional assembly instructions to reshuffle the SIMD vectors. Another solution is to restructure the array a, to compress unused elements and gather hot elements in order to achieve contiguous accesses and enable efficient vectorization. However the potential vectorization gain remains uncertain for both solutions, as other factors such as the application context, for instance the whole loop nest localities has to be taken into account for performance assessment.

Lack of dynamic information imposes compilers conservative dependence management, indeed without profile information handling data dependences is risky. Unfortunately, this causes a lot of potentially advantageous vectorization failures as well, as shown in Figure 1.16. The innermost loop has a reduction-like dependence on `ID2` accesses, preventing compiler vectorization.

It is unfortunate because the outer loop is vectorizable and intrinsically parallel. The outer loop has not been vectorized because the compiler preferred to ignore vectorization opportunities because the complex double element fills the entire vector. However, outer loop parallelization can be achieved by splitting real and imaginary parts of complex numbers. Whether to choose between segregating real and imaginary parts or not is not contemplated here and would make suitable candidates for performance assessment.

1.3.2 Tools

Profiling is a method to analyze programs dynamically, which is particularly opportune for identifying performance bottlenecks and runtime behaviors the conventional compilation techniques cannot grasp. It consists generally in instrumenting the code prior to execution, in order to insert probes into the code to prepare runtime measurements, and then running the probed code to collect runtime information. It allows among others to get time measurements, particular hardware counters values, memory traces, and the number of function calls. Memory traces for instance are program memory accesses records, useful to notice sub-optimal memory access patterns responsible for bad performance. Different approaches to profiling exist.

GNU gprof [34] is a famous performance profiling tool, it involves program interruptions to allow sampling, which is a light form of instrumentation that uses approximations to deduce code profiles. Gprof requires the user to recompile and relink his application. Many vendor-specific profiling tools are available, such as Intel VTune AMD CodeAnalyst, ARM Streamline and Nvidia visual profiler. Valgrind [76], PIN [67] and VTune [88] are profiling tools that perform instrumentations via Just-In-Time (JIT) recompilation with different motivations. Valgrind JIT recompilation does not directly target the physical architecture, rather, it uses simulation on-the-fly with a simplified ISA similar to RISC. On the other hand, PIN replaces on-the-fly original binary instructions with novel instructions such as user probes for instance, code portion by code portion, and performs analysis on-the-fly as well. TAU [98] is another profiling tool that proposes different solutions for instrumentation, one of them is an API for source code instrumentation. In each case the binary is rewritten, either during runtime or offline. TAU has the advantage of supporting heterogeneous architectures. Vampir [54] is a profiling tool that also target CPU/GPU, the instrumentation is performed via compiler wrapper and the user code needs recompilation and relinking much like gprof for instance. Overhead due to instrumentation have typically an order of magnitude of 10 or 100 times, which can be significant on real applications. A technique implemented in many tools is the sampling, a statistic method that extrapolate performance results from a smaller subset of probing instances. Most of profiling tools offer performance analysis to help, however the user is left with the task of improving, implementing and evaluating potential optimizations.

Some works have followed the path to act on the source code side. Frameworks such as the Scout [58] source-to-source compiler enable the programmer to annotate the source code with `pragma` directives that are subsequently translated to SIMDized code. A recent work of Evans *et al.* [31] presents a method to analyze vector potential, based on source annotated code and an on-the-fly dependence graph analysis using the PIN framework. Other works aim at specialized fields and will produce efficient code for a selected class of applications, e.g. stencil computations [47, 42] for instance, or allow to auto-tune specific kernels [111]. Profiling tools such as Intel VTune [45] or the suite Valgrind, for instance, can diagnose code efficiency and pinpoint issues such as memory access patterns with bad locality. Intel VTune may even suggest that the programmer resorts to SIMD intrinsics. However, such an action is not always desirable for code readability and maintainability. More elaborate works focus on specific code optimization [111][56], other works [42] suggest data restructuring, but are limited to very specific cases such as stencils here. VP^3[114] is a tool for pinpointing SIMDization related performance bottlenecks. It tries to predict performance gains by changing the memory access pattern or instructions. However, it does not

```
1 for (i = 0; i < N; i++) {
2         for (j = 0; j < N; j++) {
3                 a[i][j] = a[i][j]*c[j][i]
4                         + b[i][j];
5         }
6 }
```

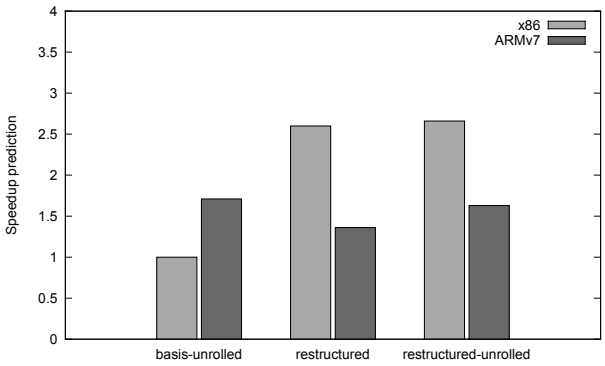


Figure 1.17: Kernel `s1115` from TSVC suite, showing the need for quantified optimization strategy assessment.

propose high level restructuring. Similarly, ArrayTool [64] can be used to regroup arrays, to gain locality, but there is no deeper change in data layouts.

1.3.3 The importance of quantification

Although user hints are important in the sense that they may provide valuable feedback to the programmer, there is no guarantee they are not incorrect. The listing on Figure 1.17 shows a kernel extracted from the TSVC benchmark suite [14, 73], a suite of codes for the evaluation of SIMDization capabilities of compilers. This kernel is part of the function `s1115`. Array `c` is stored row-major, but accessed column-major, which hinders SIMDization. A possible strategy is to transpose the `c` array. Another classical strategy is to perform partial loop unrolling. A third strategy is to combine both data transpose and loop unrolling. The barplot on Figure 1.17 shows the relative speed-ups of these strategies compared to the original version. Comparisons are made both for an x86 architecture, the Intel Sandy Bridge E5-2650 @2GHz using icc 13.0.1, and an ARM architecture, the ST-Ericsson Snowball (ARMv7 Cortex-A9 @800MHz) using gcc 4.6.3. Both architectures clearly show very dissimilar behaviors on this example, to the point that the two strategies showing good results on x86 perform poorly on the ARM architecture. Consequently, quantification is useful and would be particularly relevant for user feedback.

1.4 Towards a data layout restructuring framework proposal

The plethora of computer architectures and their systematic replacements make programming more complicated, despite the numerous abstractions proposed. Using these abstractions means less control over performance on the programmer side, often it is not possible to obtain decent performance without obfuscating the code too much. The non-portability of performance impose the user layout design choices of an unacceptable complexity. Moreover it is generally difficult to estimate the potential gain of a given restructuring strategy. Given the compiler intrinsic limits in the domain of array restructuring and the insufficient feedback reported to the programmers in general, we propose an original framework to overcome these difficulties.

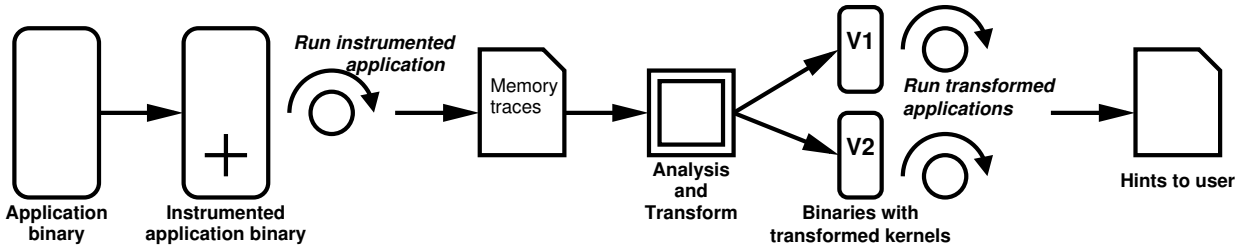


Figure 1.18: Big picture

1.4.1 Proposition overview

We propose a tool for elaborate user feedback on data layout restructuring, the aim is to guide the user through his steps of performance optimization process. First by pinpointing initial layout/pattern issues via a clear report with high-level vision of the issues, as well as a precise report to figure which individual instructions are responsible of the performance issues. Then, the tool suggests a reasonable set of relevant transformations although possibly ambitious, by both a high-level representation and the individual memory accesses to modify in order to implement the new suggested layout mapping. Finally, each of the suggested transformations are provided with proper quantification of benefits, allowing the user to have a good sense of the gain to expect first, and then which transformation to choose. The tool principle is illustrated in Figure 1.18, it works based on the original user application binary. An instrumentation is applied in order to probe all the memory accesses performed by the code, generating a set of memory traces. Then an analysis step is applied using the input profile, to characterize the given memory patterns and the layout structures. Given the potential suboptimality of the data structures, a heuristic chooses a restricted set of appropriate transformation candidates for optimization, which are finally automatically generated by rewriting the application binary, constructing this way the so-called code mock-ups. These mock-ups are binary codes built with the sole purpose of mimicking actual user code behavior, would he apply the transformation considered. Mock-ups are timed in order to deduce the potential performance gain or lack of gain, and permit to finally report an elaborate user feedback: first, a proper description of each transformation is given, in particular all the code modifications needed are pinpointed to the programmer; second, each of the said transformations are quantified, meaning that a performance gain estimation is provided to the user, so as to guide his implementation choice to improve application layout performance.

The benefits of this approach are manifold. The user feedback we provide is exploitable, in the sense that it is a reliable guide to performance optimization. Unlike most tools that offer performance issues reports, our approach accompanies the user through the next steps of performance optimization, that is providing clear restructuring strategies options, and then giving a way of weighing the mentioned options by reporting performance estimations of each option. Besides, inherent code complexity especially with high-level abstraction may even conceal otherwise obvious optimizations to the user. An automatic tool allows among other benefits to traverse a reasonably large space of transformation possibilities, which includes a significant number of transformations the user potentially would not consider. As the user is let the choice of array restructuring, some transformations application may require deeper modifications than others. Indeed, the tool works on a smaller scope than the whole application, therefore its visibility is limited. The decision of

restructuring is left to the user who knows his application, and can decide how to apply the modifications, for instance if the array should be restructured for the totality of the application rather than on a given particular kernel. Finally the user could prefer to choose transformations easier to apply even if they do not bring near-optimal performance, as opposed to users that need the greatest performance improvement they can obtain. While proposal helps portability, user can also utilize feedback to write highly optimized code depending on his requirements. As a matter of fact, were the transformations all not beneficial, the information would still be useful to the programmer as he would know array restructuring is not a viable option and therefore not waste time trying to optimize code this way.

Plus, layout mapping modifications are not obfuscating the code. This approach allows to push to bridge the important gap [94] between typical compiler output and expert programmer hand-written output. Moreover, data layout restructuring is intrinsically independent to control flow optimization. This means any control optimizations can be applied in any order without interfering with any data layout optimization. The proposed method is also non architecture-dependent, as the layout remapping does not assume any particular architectural feature. The proposed framework operates on a small application code portion, usually the hottest kernel, making its visibility somewhat restricted. Besides, the use of memory traces implies the method applies on a given run parameters, and can be seen as tailored to the given run for that matter. Therefore code validity can not be ensured, as number of parameters such as arrays sizes are indeterminate. However, we allow ourselves to explore aggressively more transformations than compilers consider, some may not be applicable for a given application but the user is notified and can make an appropriate choice. On the other side, compiler products need to be operational for all parameters due to semantics preservation and lack of information because it is not specific enough. For that matter, the interaction Tool/User we propose is useful, as it allows the user to rewrite his code, perhaps in a simpler way than it was and potentially enable more compiler optimizations.

This work encompasses a certain number of contributions. First of all, a new formalism is proposed in order to express data layout and data patterns. Its concise form allows to simplify discussions around layouts and patterns, and to apply transformations in an efficient systematic way. Apart from the formalism presented earlier in this sections, other original formalism have been introduced recently to express data structures in a certain level of abstraction. ArrayOL [13] is a language specific for signal processing, where arrays and patterns in particular are represented as vectors, and transformations as matrices. Other works like Torch7 [22] and TensorFlow [2] rather choose to represent data arrays by tensors.

Second, we use the formalism to express the initial application troublesome layout in a normal form, independent of compiler optimizations, which correspond to the delinearized version of the flat accesses as given by the memory traces. This is particularly useful in our binary approach as the input binary is the product of prior compiler optimizations and memory instructions can be expressed in different ways, as it is the case for unrolling transformations for instance. Delinearization is the first analysis to perform on the compiler side, in order to be able to restructure the layout. Parametric delinearization, for some particular codes, has been proposed by Grosser *et al.* [35]. Specifically for stencil codes, using the polyhedral model, Henretty *et al.*[41] propose a complete restructuring of layout for SIMDization. This would not apply to the Lattice QCD code given in Figure 1.13 because of the indirection. Berdine *et al.*[10] propose to analyze the shape of

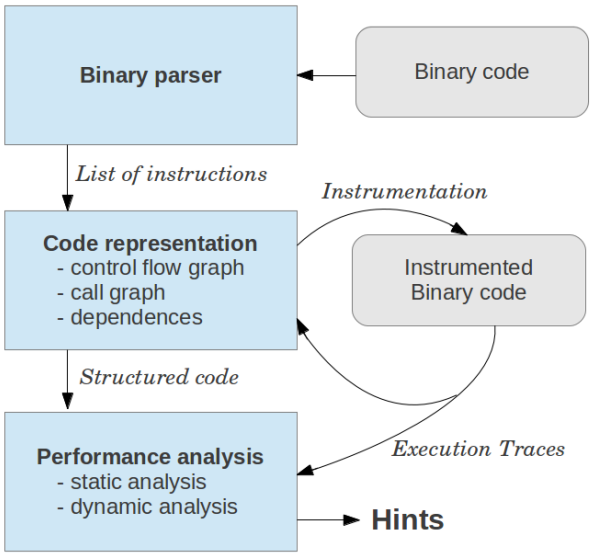


Figure 1.19: MAQAO architecture

more complex data structures, such as recursive data structures with linked lists in particular.

1.4.2 Data restructuring evaluation

There are basically three main techniques for data restructuring evaluation, necessary to choose a near-optimal solution. First of all, there are the theoretical performance models, that can be incorporated in compiler static analysis and assume a simplified view of the architecture. Then, there is the simulation approach, that consists in tracing memory patterns and using a simplified thus lighter architecture environment, a cache simulator typically, to perform measurements on the considered transformations, and is usually used to assess one particular parameter such as the cache miss rate for instance. However, to assess memory accesses performance, several parameters have to be taken into account. The cache miss rate alone, while relevant, cannot properly model all caches considerations. Our approach consists in measuring the impact of restructuring transformations *in-vivo*, which means code is executed on the actual user architecture. Therefore the measurements are more accurate than cache models or simulations, as they represent real measurements with all architectural considerations included. Both simulation and *in-vivo* imply running only a portion of the program, usually the hotspot kernel of the program, because they both rely on memory traces which are costly in terms of time to obtain and thus instrumentation is rarely performed on the whole application. Several previous works have proposed techniques to extract pieces of code for evaluation [60, 84], and shown that such an operating mode is viable for performance measurements [3].

We use the MAQAO profiling tool [8] to instrument and to patch the application binary. As shown in Figure 1.19, MAQAO is a tool the operates at binary level. It extracts the code representation from the user application binary, from which it performs static analysis, and which it can instrument in order to perform dynamic analysis on collected memory traces. Compared

to PIN [68], a tool with similar functionalities, MAQAO performs static rewriting-based instrumentation from binary to binary and further analyzes the collected information in a post-mortem fashion. PIN, on the contrary, dynamically rewrites binary codes while they execute, and performs analysis on the fly. As most of MAQAO work is done offline, the overall cost for analyzing a binary with MAQAO is much smaller than with PIN. With MAQAO, it is possible to captures memory streams by instrumenting all instructions that access memory, on a specified code fragment (such as a function or loop). For each instruction instrumented, the flow of addresses captured is compressed on-the-fly using a lossless algorithm, NLR, designed by Ketterlin and Clauss [51]. This compressed trace represents the accessed regions with a union of polyhedra, captures access strides and multidimensional array indexing (when possible). The purpose of tools based on dynamic dependency graph analysis [71] such as our MAQAO approach is therefore to explore SIMDizability from an entirely different point of view. As such, MAQAO is a complementary analyser with respect to vectorizing compilers, for application programmers to investigate where and how their code could be restructured to enable or improve vectorization by the compiler. Holewinski *et al.* propose a similar approach [43], which is the closest from our own work, to the best of our knowledge so far. However, their implementation does not preserve structural information about the application such as iterated memory references inside loop nests. This limits the amount of details that can be reported to the programmer, and this also limits the richness of the information that could be injected back inside a vectorizing compiler to improve its output. Our approach instead preserves such key information, which distinguishes from prior efforts.

A lot of works have addressed the problem of data layout restructuring. Rubin *et al.* [92] regrouped many existing arrays restructuring techniques we detailed earlier in this document, and assessed their impact using simulation allowing combinations between the different transformations. They research a good layout solution by iteratively applying the transformation and simulating them. Hundt *et al.* [44] proposed both a compiler method for data layout transformation and an advisory tool. The profile-based restructuring heuristic they present for array splitting is based on the hot field/cold field separation. They provide both non-profile and profile profitability analysis. However, the number of transformations they investigate is low because in a lot of their test cases transformations are invalid. In the most difficult cases, the advisory tool is used instead to suggest more effective structure designs. A later paper [87] expands their work to multithread consideration, with a special focus on the false sharing issue. Mannarswamy *et al.* [74] implemented a compiler method using profile information to perform data transformations. Given the difficulty of performing whole program transformations, they proposed to apply the layout transformation on a smaller region, performing selective data copies for local region optimization. Wu *et al.* [115] proposed algorithms to reorganize data on GPU in the aim of improving data coalescing, crucial for GPU performance. Kofler *et al.* [55] focused on AoS to SoA transformations on GPU as well, the idea is to separate AoS by fields using hardware parameters in a first time, then to select a good transformation, which can be a SoA or something else. Annotations and specific data layout optimizations with compiler support has been proposed by Sharma *et al.* [96]. The source-to-source transformation requires to describe in a separate file the desired array interleaving. Similarly, the array unification described by Kandemir [50] and Inter-Array Data regrouping [25] propose to merge different arrays at compile-time in order to gain locality. The POLCA semantics-aware transformation toolchain is an Haskell framework offering numerous transformation operators using programmer inserted pragma annotations [103]. Neither of these approaches provide an assessment of the performance gains to guide the user restructuring or hint

generation, and these compile-time approaches cannot handle indirections. Li *et al.* [62] used data layout transformation on the compiler side to enable vectorization in the context of loops with data structure accesses. The StructSlim profiler [91] helps programmers into data restructuring through structure splitting. For GPU, copy is performed at transfer time and data layout change is also performed at this step [109, 101]. Code analysis is performed statically, on OpenCL for instance. The same approach has been explored for heterogeneous architectures [70], assessing affinity between fields and clustering fields, and devising multi-phase AoS vs SoA data layouts and transforms. Finally, Gupta *et al.* [37] proposed a new formal way of quantifying both spatial and temporal locality, with an approach based on memory traces.

Chapter 2

Vectorization

2.1 Layout Transformations To Unleash Compiler Vectorization 39
2.2 Hybrid Static/Dynamic Dependence Graph 40
2.2.1 Static, Register-Based Dependence Graph. 40
2.2.2 Dynamic Dependence Graph. 41
2.3 SIMDization Analysis 42
2.3.1 Vectorizable Dependence Graph. 42
2.3.2 Code Transformation Hints. 43
2.4 Conclusion 44

Using SIMD instructions is essential in modern processor architecture for high performance computing. Compilers automatic vectorization shows limited efficiency in general, due to conservative dependence analysis, complex control flow or indexing. Typically, not only poorly designed layouts or strided patterns constitute performance issues by themselves without SIMD considerations, they also may hinder full SIMD optimizations, causing even more performance lack of gain in a SIMD context. This chapter details a technique to determine if a given kernel is intrinsically vectorizable. In particular, we notice that data layout issues is a common cause of vectorization failure, which can be solved by performing data layout transformations

2.1 Layout Transformations To Unleash Compiler Vectorization

For modern multi-core architectures, the Single Instruction, Multiple Data (SIMD) instructions are essential in order to reach high levels of performance. With the increase of vector width – up to 16 floats for Intel Xeon Phi SIMD vectors for instance – SIMD instructions are real performance multipliers. Several options are given to the application developer in order to vectorize a code: explicit vectorization through assembly instructions, intrinsics, GCC vector extensions or other language extensions (such as Intel SPMD Program compiler for instance [86]) or implicit vectorization through the automatic vectorizer of the compiler.

Explicit vectorization, however, is complex and time consuming, assembly and intrinsics-based approaches also are not portable and GCC extensions only offer a limited subset of arithmetic operations. Consequently, the vectorization effort for most applications is delegated to the compiler, which may not entirely succeed or even completely fail to meet the programmer expectations, depending on the code structure and complexity. Indeed, an over-conservative dependence analysis, an incomplete static information concerning the control-flow or a strided data layout are among the main reasons why the compiler may not generate vector code, even though the code actually is vectorizable. Therefore, determining whether the code is vectorizable, independently of any compiler limitations, as well as pinpointing the issues that may hinder vectorization and the transformations required to enable it are critical capabilities for the developer.

Our approach is based on a twofold static and dynamic dependence analysis of the binary code, and generates vectorization hints for the user. By combining a static and dynamic dependence analysis, our method identifies the limiting factors for vectorization, such as misaligned data, non contiguous data together with opportunities to vectorize, obtained through rescheduling, loop transformations, reduction rewriting and vectorizable idiom rewriting. The runtime analysis is essential in capturing dependences in presence of complex control flow or structure indexing. This approach is complementary to what compiler optimization reports can provide, bringing a more detailed analysis of vectorization opportunities, in particular of the opportunities missed by the compiler.

2.2 Hybrid Static/Dynamic Dependence Graph

The dependence analysis we propose is a combination of a static dependence analysis, for registers, and dynamic dependence analysis for memory dependences. The static dependence analysis on registers is already implemented in MAQAO and corresponds to an SSA analysis. Memory dependences are obtained by tracing with MAQAO all memory accesses within the innermost loops, and then computing dependence distances.

2.2.1 Static, Register-Based Dependence Graph.

The dependence graph on registers is resulting from an SSA (static single assignment form[23]) analysis, proposed by MAQAO. Besides, MAQAO handles special cases for dependences on x86:

- `xor` instructions, applied twice to the same register, set the value of this register to 0. While the operation is reading a register, this is not considered as a read access.
- SIMD instructions can operate only on the lower or higher part of a SIMD register. Operations that operate on different parts of a register are not considered in dependence.

In addition to the existing analysis, we tag all dependences where a register is read for an address computation. The graph is then partitioned according to these edges (cutting the graph through these edges), usually in two parts: Instructions preceding these edges are address computation instructions (such as index computation, update of address registers), while instructions after these edges are actual computation (memory accesses, floating point operations, ...) for which SIMDization may be applied. When an indirection occurs, the dependence graph has a path with two tagged edges and can therefore be partitioned into three or more subgraphs. The partition of instructions following all tagged edges is said to be the *computational part of the graph*, while the

other instructions are part of the *address computation part* of the graph. In the rare cases where it is not possible to cut the graph following tagged edges, we assume there is no computational part.

2.2.2 Dynamic Dependence Graph.

Dynamic dependences are essential to capture what the compiler may have missed, concerning the control flow or the way data structures are indexed. The dynamic dependence graph is built from the memory trace for each read and write instructions in innermost loops.

Algorithm 1 describes how dependence distances are computed. $w.trace$ denotes the trace captured for an instruction w. For each couple of read and write accesses in a loop, we first perform an interval test, based on their trace (line 2), and then compute a *dependence distance*. The dependence distance is defined as the number of loop iterations between two instructions accessing the same memory location. When two traces have the same loop structure, the subtraction between the traces (line 4) subtracts the expressions that are at the same position in the trace. If the result is not the same constant value for all subtractions, then $*$ is returned, otherwise the constant value is returned. The special $*$ dependence distance notation between two instructions denotes the fact that their dependence distance is not constant during the execution of the program. Note that only uniform dependences are captured this way but as far as SIMDization is concerned, this captures all vectorization opportunities that do not require non-local code transformation.

```
for(i=1; i<100; i++)
  C[i] = C[i - 1] + B[2 * i];
```

(a) Code example

```
for i0 = 0 to 98
   write 0x2ba1a3bd442c + 4 * i0
```

(b) Compressed trace for `C[i]`

```
for i0 = 0 to 98
   read 0x2ba1a3bd4428 + 4 * i0
```

(c) Compressed trace for `C[i-1]`

```
for i0 = 0 to 98
   read 0x2ba1a4000000 + 8 * i0
```

(d) Compressed trace for `B[2 * i]`

Figure 2.1: Example of trace compression using NLR. For the code in (a), one compressed trace per memory access is produced, (b), (c) and (d).

Algorithm 1: Dynamic Dependence computation for an innermost loop L

```
1  for w, a write and r, a read in L do
2      if w.trace ∩ r.trace ≠ ∅ then
3          if loops of w.trace = loops of r.trace then
4              return r.trace − w.trace;
5          else
6              return ∗;
7
8      else
9          return 0 ;
10
11
```

For the example in Figure 2.1.b and 2.1.c, both traces have the same structure, the same strides, and the difference between the read and the write addresses is an offset of -4. Then we evaluate how this difference can be compensated by a variation in the loop indices (here $i0$). We find a unique solution within the loop bounds, 1 and this shows that the dependence distance between the write and the read is 1. In general, finding the vector of iteration counters that compensate for the offset between the read and the write leads to a multi-dimensional dependence vector. Only read after write dependences are evaluated, and the sequential order of the assembly code is used to compare relative positions for reads and writes. Note that all distances for register dependences

```
for (int i = 1; i < LEN2; i++) {
  for (int j = 1; j < LEN2; j++) {
   bb[j][i] = bb[j][i-1] + cc[j][i];
}
```

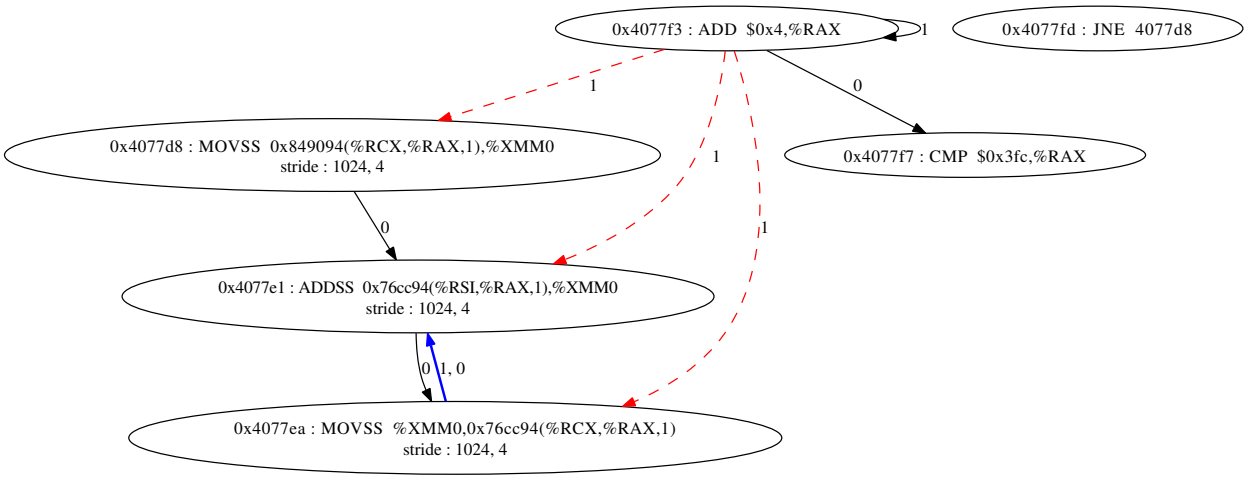


Figure 2.2: Code and dependence graph for one loop of function s2233 in TSVC benchmark.

correspond in this case to innermost loop carried dependences. Figure 2.2 presents the s2233 function from TSVC and its dependence graph, combining both the static and dynamic graphs. The nodes each represent an assembly instruction, along with its strides when it is a memory access. The dashed edges in red represent dependences for registers used in address computation. Cutting the graph along these edges separates the computational part (left nodes) from the address computation and control part. The bold blue edge, labelled with $1, 0$ represents the memory dependence corresponding to `bb`, directly computed from the trace. The strides denoted on the edges have two values: 1024 and 4. The first one corresponds to the stride for the innermost loop, j, and the second to the i loop. This shows that here, none of the accesses have good spatial locality.

2.3 SIMDization Analysis

The SIMDization analysis is in two steps: First, we determine whether the code has a parallelism compatible with a SIMDization, independently of any data layout or control limitations (such as large stride). Then, we refine the analysis to detect special cases and accordingly guide the programmer towards enabling and improving the vectorizability of the code.

2.3.1 Vectorizable Dependence Graph.

The dependence graph (static and dynamic) is first partitioned according to address computation edges as described previously. The graph is said vectorizable if one of the three conditions applies to the computational part of the graph:

- There is no cycle.

- There is a cycle, with a cumulative weight greater than the width of SIMD vectors.
- There is a cycle, with a cumulative weight smaller than the width of SIMD vectors, and the instructions of the cycle all are of one of the following types: `add`, `mul`, `max`, `min`. The cycle corresponds to a reduction.

A code with a vectorizable dependence graph may require transformations in order to be SIMDizable. This is detailed in the following section.

2.3.2 Code Transformation Hints.

We propose to identify a number of transformations required for the SIMDization of the code, depending on the dependence graph, on the stride expressions, and on the control flow graph.

Data alignment: When the graph is vectorizable without cycle, misaligned data is detected by comparing the starting address of all memory streams with the width of SIMD vectors. In the simpler case, the user can either change memory allocation of heap-allocated data structures, or use pragmas for aligning stack-allocated data. When for instance $A[i]$ and $A[i+1]$ occur in the same code, one of the two accesses is misaligned. This would require shuffle instructions, or unaligned loads/stores whenever they exist.

Rescheduling: When the graph is vectorizable without cycle, there may still exist some dependences with non-null distance. Due to the fact that the analysis is performed on assembly code, this may require some modifications at the source code such as some rescheduling of loads/stores and computations, splitting some larger instructions. A template of the vector code is generated by our tool (an example is given in the following section), giving a correct instruction schedule after SIMDization.

Loop transformations: Loop interchange is proposed when all accesses within the loop have a large innermost loop stride, and another loop counter in the expression corresponding to the same outer loop has a stride 4 (for floats and ints). Interchanging these two loops would result in better locality and enable SIMDization.

Loop reversal: Traces with negative stride expressions lead to this hint. Note that if other reads for instance have a positive stride, the reversal is not beneficial any more.

Data reshaping required: This is a fallback hint for large innermost loop strides, and for codes with indirection (detected on the static dependence graph). On the Sandybridge and Xeon Phi architectures, instructions for loading or storing non-consecutive elements into/from SIMD vector have been added to the ISA (GATHER on Xeon Phi and Sandybridge and SCATTER on Xeon Phi). Use of these instructions, through assembly code or intrinsics, is an alternative to data reshaping.

Versioning required: The static analysis on the code may lead to a conclusion different from the trace analysis. For instance, the trace may find a regular stride for a memory stream whereas statically, this stream results from an indirection, or depends on a parameter. Similarly, the dynamic control flow (the real path taken) may be a subset of a more complex static control flow. In these cases, the trace may have captured only one behavior of the code, for a particular input. The vectorization may only be possible in this case through the versioning of the loop, depending on the values of the array, of a parameter.

Idiom Recognition: When the code is vectorizable, with 0 dependence distances or with reductions, the dependence graph can match a predefined dependence graph representing a well known computation. The shape of the dependence graph and the instructions themselves are matched with the predefined graphs. In this case, the user can call a library function instead of trying to

vectorize the actual code. The predefined functions considered are so far: dot product, daxpy, copy, sparse copy (copy with an indirection either in the load or in the store), but more complex functions can be added with ease.

All these hints help the user rewriting his code to enable SIMDization, and in particular, this document explores in depth the problem of *Data reshaping* by proposing quality SIMDizing layout transformations.

2.4 Conclusion

We showed in this chapter a novel technique to highlight vectorization issues, providing the programmer with detailed feedback on vectorization hurdles in his application Some issues correspond to control optimizations, that are simple and clear enough for the user to apply, some can be handled by state-of-the-art techniques as well. While some issues can be overcome via control optimizations , data layout issues remain a major problem that compiler analysis is too limited to resolve and deep code modifications are required, and user is still clueless on how the modifications should be made, and how much is he expected to gain by applying them. In the following chapters, we propose to study extensively data layout transformations, in particular the hint of data reshaping is quantified, and deep code modifications are pinpointed to the user.

Chapter 3

Layout/pattern analysis

3.1 Data layout formalism 45
3.2 Layout detection 48
3.3 Delinearization 50
3.3.1 General Points 52
3.3.2 Properties 54
3.3.3 Characterization 55
3.3.4 Case Study 57
3.4 Conclusion 59

In order to apply opportune data layout transformations, knowledge of the initial memory structure is required beforehand. This chapter details how memory access patterns are formalized, regardless of their respective method of acquisition, be it from compiler analysis or memory traces analysis for instance. We then explain and detail our choice of analysing memory traces for that matter, that is how we sort out the initial structure, and how we delinearize it to obtain an intermediate representation that has two purposes. Firstly, it gives a consistent approach to properly apply transformations. Secondly, handiness of the IR allows to provide quality feedback to the programmer. We also detail how intrinsic kernel SIMDizability can be determined from said memory traces, combining both static and dynamic analysis to obtain valuable information in the aim of high performance achievement.

3.1 Data layout formalism

Discussing data layouts transformation is not as straightforward as it may sound. Not only the structure design must be abstracted but also the patterns that access it, because more than the layout itself it is the data access patterns that dictate the code performance. Indeed, on the one hand, layout expression lies in the programming language definition, which can be verbose especially if the structure is complex: the structure itself is composed of different fields that can be arrays or structures themselves and so on. On the other hand, the data access patterns are described in the source code by loop nests, potentially addressed by multiple complex expressions, with the

possible presence of induction variables, pointers, or index tables. A new level of abstraction is required to allow reasoning on data layouts and access patterns in a systematic manner.

We consider data layouts as any combination of arrays and structures, of any length. A layout is the description of this combination and the elements that are accessed in it. This description is done for each function, for each different structure. Implicitly, all layout are considered for a given program fragment. A syntactic memory access expression in the code defines a set of memory address values. This can be denoted as:

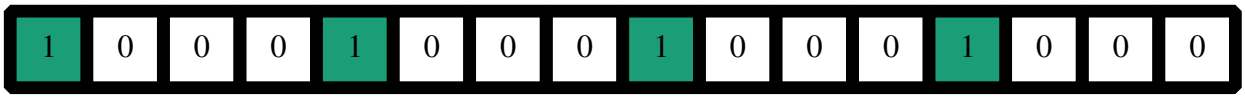
$$base + J$$

where $base$ is the base address and J is a set of positive integers including 0 representing memory offsets. All addresses are within a range $[base, base + d - 1]$ where d is the diameter of J. The set of offsets J can be represented as a function:

$$\begin{array}{rrcl} S_{J,d}: & [0, d-1] & \rightarrow & \{0,1\} \\ & x & \rightarrow & 1 \text{ if } x \in J, 0 \text{ otherwise} \end{array}$$

This function characterizes the set J, since $S^{-1}_{J,d}(1) = J$. Follows an example to illustrate this point:

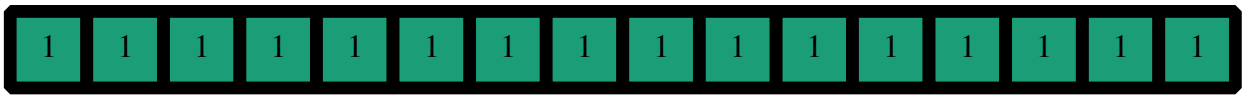
$S_{\{0,4,8,12\},16}$:



Which represents a structure of 16 elements, only 4 of them are actually accessed. Intuitively, this especially allows to express strides, which are factors indicating offsets between two consecutive memory accesses. In the particular case where J is an interval, that is the structure elements are accessed consecutively, structure is denoted A_d, or $A_{J,d}$ if $d > \#J$ meaning there is an offset before the array, and possibly padding after it as well. This kind of layouts will be referred to as array layouts in the following.

Example:

A_{16} :



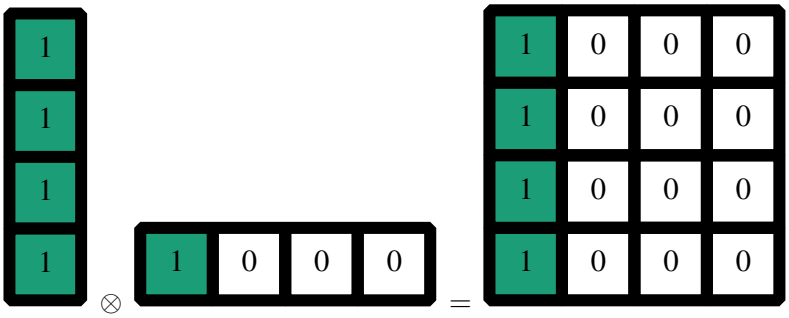
By default, if $S_{J,d}$ is not an array layout, we will call it a structure layout. Now, this flat definition of memory accesses does not account for the data structure itself, and array of structures (AoS), structures of arrays (SoA) or any other combination can be written in this manner. To build a multidimensional data structure, we define the product operator $\otimes$ on such functions:

$$\begin{array}{rrcl} S_{J,d} \otimes S_{J',d'}: & [0, d-1] \times [0, d'-1] & \rightarrow & \{0,1\} \\ & x, y & \rightarrow & S_{J,d}(x) * S_{J',d'}(y) \end{array}$$

The product $\otimes$ defines a multidimensional operator, of domain $[0, d-1] \times [0, d'-1]$.

Example:

$A_4 \otimes S_{\{0\},4}$:



This represents the product of two flat structures, one array layout and one structure layout, and the obtained result is a two-dimensional array of structures. Note that the formal description corresponds to the intuitive representation of the data, as an AoS corresponds to the combination of the two types of layouts, described by $A_{d'} \otimes S_{J,d}$ for some values of d, d' and J.

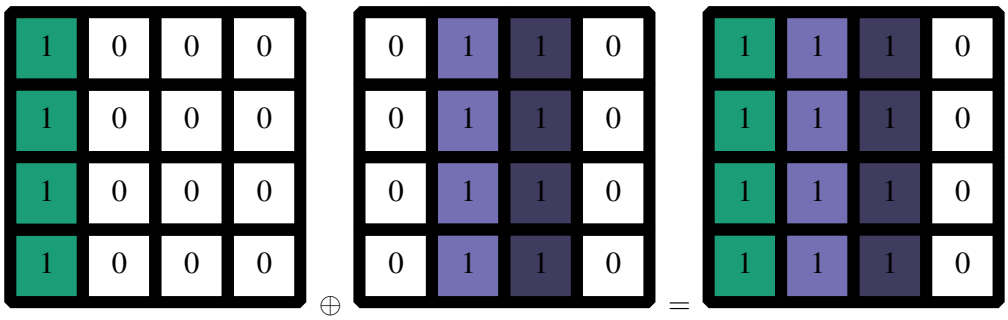
For any two layouts expressed as any product of unidimensional layouts, we define a sum operator, corresponding to the union between the elements accessed by each operand: if L_1 and L_2 are layouts with the same domain J:

$$\begin{array}{rccl} L_1 \oplus L_2 : & J & \rightarrow & \{0,1\} \\ & x & \rightarrow & L_1(x) + L_2(x) \end{array}$$

where the addition on integers is a saturated addition.

Example:

$(A_4 \otimes S_{\{0\},4}) \ \oplus \ (A_4 \otimes S_{\{1,2\},4})$:



This operation is useful to merge different patterns that access the same structure into a single expression, because the same factorization identities exist with $\oplus$ and $\otimes$ as with integers, thus some simplifications are possible between expressions involving both operators:

$$(S_{I,d} \otimes L) \oplus (S_{J,d} \otimes L) = S_{I \cup J,d} \otimes L \tag{3.1}$$

$$(L \otimes S_{I,d}) \oplus (L \otimes S_{J,d}) = L \otimes S_{I \cup J,d} \tag{3.2}$$

Intuitively, union between structures is applicable only if they have the same coordinates: generic layout L must be equal in both operands, otherwise it would mean we are trying to unify two structures that are far apart in memory, which makes no sense and erases useful information.

Also, by definition, if a given structure L is repeated in a contiguous manner, then we can write it as an array of structure.

$$\bigoplus_{k=1}^{n} L = A_n \otimes L \tag{3.3}$$

3.2 Layout detection

Performing data layout transformations requires analysis of initial data structures themselves in the first place. It is important because each array has specific patterns, thus specific transformations. Therefore, identifying each pattern array affiliation is a necessary step. On the source code, finding whether two memory accesses correspond to the same array region is performed by an alias analysis. Delinearization can be used in some simple cases to retrieve the multidimensional structure associated to the addresses. Because indirections or complex operations can be involved in the address computations, we propose to resort to memory traces. The code fragment is executed and all memory accesses generate a trace. This trace is compacted on-the-fly with the NLR method [51] in order to find possible regularities in strides. Memory addresses accessed for each load/store of the code fragment are compacted as shown below:

```
1 for (i = 0; i<1000; i++)
2   for(j=0; j<50000; j++)
3     A[j] = ...
```

Figure 3.1: C snippet

```
1 for i0 = 0 to 999
2   for i1 = 0 to 49999
3     val 0x6c4560 + 4*i1
4   endfor
5 endfor
```

Figure 3.2: Trace snippet, given for an array A of 4-Bytes elements

Translation from memory traces to formalism is immediate, the memory access pattern described in Figure 3.1 by instruction $A[j]$ gives : $S_{\{[0,49999]\},50000}$, which is a structure whose elements are accessed consecutively which can also be written as A_{50000}. The stride of 4 in the trace in Figure 3.2 corresponds to the size of the element in bytes, a single precision float here.

Data layouts are said to be distinct if their memory accesses are in intervals that do not intersect. The distinct data layouts must be detected. Then, for each data structure, restructure it into a multidimensional layout. This has been described in a paper [39]: The idea is to merge the different memory accesses according to the intersection of their interval of addresses. We can determine a base address common to the data layouts found (the minimum address), and a maximum diameter for the layout (the maximum address value minus the base). This analysis consists in determining the layout of data accessed by each assembly instruction. The data layout detection focuses on finding out: How many arrays are accessed, with how many dimensions, with which element structure (that is, how many fields per array element). This assumes that each load/store assembly instruction of the studied function accesses exactly one single array.

Memory addresses accessed for each load/store of the code fragment are compacted as shown in Figure 3.4. Traces are here represented by loops iterating over the successive address values taken during the execution. The first step of the analysis consists in identifying the loads/stores that access the same array. To this end, each region accessed by a load/store is converted to a

```
1 for (nl = 0; nl < ntimes; nl++) {
2         for (i = 0; i < N; i+=2) {
3                 a[i] = a[i-1] + b[i];
4         }
5 }
```

Figure 3.3: C source code of function `s111` from TSVC benchmarks [72].

```
1  # Trace for access a[i-1]        Strided interval for a[i-1]
2  for i0 = 0 to 999
3    for i1 = 0 to 1535
4      val 0x1525d40 + 8*i1      => [ 0x1525d40 ; 0x1525d40 + 8*999; 8 ]
5    endfor
6  endfor
7  # Trace for access b[i]         Strided interval for b[i]
8  for i0 = 0 to 999
9    for i1 = 0 to 1535
10     val 0x1528d84 + 8*i1      => [ 0x1528d84 ; 0x1528d84 + 8*999; 8 ]
11   endfor
12 endfor
13 # Trace for access a[i]         Strided interval for a[i]
14 for i0 = 0 to 999
15   for i1 = 0 to 1535
16     val 0x1525d44 + 8*i1      => [ 0x1525d44 ; 0x1525d44 + 8*999; 8 ]
17   endfor
18 endfor
```

Figure 3.4: Trace example on function `s111` from TSVC benchmarks [72]. Each trace is compacted with NLR algorithm into loops in this simple example, iterating over successive addresses. A simplified representation with strided intervals is used.

simpler representation, using a strided interval (lower address, upper address, access stride). For clarity, in the trace the name of the array in Figure 3.4 is shown in comments.

Formally, let I denote the set of assembly instructions accessing memory. We define a relation $\equiv_{array}$ between instructions $i_1, i_2 \in I$ as: $i_1 \equiv_{array} i_2$ iff i_1 and i_2 access to the same array. $\equiv_{array}$ is an equivalence relation, the classes representing the different arrays. The idea is that two instructions with overlapping accessed memory regions are equivalent. Algorithm 2 finds the different arrays by merging overlapping regions. Its complexity is $O(N \log N)$, due to the sorts.

Lemma 1 *Algorithm 2 finds the sets of instructions that access the same arrays. It computes* $I/\equiv_{array}$.

For example, the analysis of the three regions from Figure 3.4 builds the sets $L = \{0\text{x}1525\text{d}40, 0\text{x}1525\text{d}44, 0\text{x}1528\text{d}84\}$ and $U = \{0\text{x}1525\text{d}40 + 8 * 999, 0\text{x}1525\text{d}44 + 8 * 999, 0\text{x}1528\text{d}84 + 8 * 999\}$. The two first regions are found as being accesses to the same array, while the third one is not since $0\text{x}1526\text{d}84 > 0\text{x}1525\text{d}44 + 8 * 999$.

Now, the second step of the analysis consists in finding load/store instructions accessing the same element field within an array of structures (e.g. `t[0].x, t[1].x, .., t[n].x`). Among the instructions accessing to the same array of structures, we define a relation $\equiv_{field}$ between each pair of instructions accessing the same field in the same array. Thus formally, for each two instructions i_1, i_2 accessing the same array ($i_1 \equiv_{array} i_2$), the relation $i_1 \equiv_{field} i_2$ is verified iff $i_1 \equiv_{array} i_2$ and:

$$lower_1 \equiv lower_2 \pmod{\gcd_{i \in [i_1]_{\equiv_{array}}} (stride_i)}.$$

Algorithm 2: Identifying distinct arrays from access traces.

Data: I = list of load/store triplets $[lower_i, upper_i, stride_i]$, $i = 1..N$
Result: OUT = I/R_{array}, the set of instructions grouped by array they access.

```
L = {lower_i, i = 1..N} ;
U = {upper_i, i = 1..N} ;
sort L by increasing addresses;
sort U by increasing addresses;
CLASS = {I_1} ;
for k = 2..N do
    if L_k > U_{k-1} then
        OUT = OUT ∪ {CLASS};
        CLASS = ∅ ;
    CLASS = CLASS ∪ {I_k} ;
OUT = OUT ∪ {CLASS};
```

The gcd of the strides of all accesses on the array corresponds to the size in bytes of the structure. The values of $lower_1$ and $lower_2$ modulo this size correspond to the offset of the field within a structure. Then, fields for each partition I/R_{array} can be sorted according to their $lower$ value modulo the gcd of the strides. Determining the field layout of an array of structures can be done with a $O(N \log N)$ complexity.

In the previous example, the two strided intervals $[0\text{x}1525\text{d}40; 0\text{x}1525\text{d}40 + 8 * 999; 8]$ and $[0\text{x}1525\text{d}44; 0\text{x}1525\text{d}44{+}8{*}999; 8]$ are not found equivalent, since $0\text{x}1525\text{d}40 \not\equiv 0\text{x}1525\text{d}44 \pmod 8$. Therefore the two instructions access different fields in an array of structures. Note that in the initial C code, there is no structure. However, the stride 2 on the loop counter entails that all loads on `a` are on even indices while the stores are on odd indices, a behavior similar to an access to a 2-field structure.

To sum up, Figure 3.8 shows how we can recover array affiliation of each pattern. Finally, the set of traces on each array has the form:

$$\bigoplus_{k=1}^{n} S_{I_k,d} \tag{3.4}$$

Which is simply the union of all the patterns affiliated to a given initial array.

3.3 Delinearization

Raw memory accesses have a complicated expression an potential a large number of terms. It is necessary to reduce it to a simple compact expression to allow systematic operations. Along with mathematical simplifications, delinearization is a solution to simplify the flat access definition, by exposing the multidimensional aspect of the layout. We explain that such a multidimensional representation allows us to conveniently express the layout specificities through our formalism.

```
1  for (nl = 0; nl < ntimes; nl
       ++) {
2    for (i = 0; i < N; i+=2) {
3      a[i] = a[i-1] + b[i];
4  //    A3       A1       A2
5          }
6  }
```

Figure 3.5: Source code

```
1  for i0 = 0 to 999
2    for i1 = 0 to 1535
3      val 0x1525d40 + 8*i1      # A1
4    endfor
5  endfor
6  for i0 = 0 to 999
7    for i1 = 0 to 1535
8      val 0x1528d84 + 8*i1      # A2
9    endfor
10 endfor
11 for i0 = 0 to 999
12   for i1 = 0 to 1535
13     val 0x1525d44 + 8*i       # A3
14   endfor
15 endfor
```

Figure 3.6: Associated memory traces considering 4-Bytes elements, therefore the factor 8 represented corresponds to a 2-element stride.

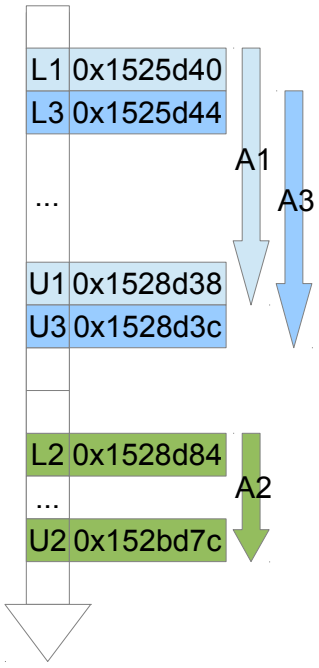


Figure 3.7: Memory Space

Figure 3.8: Layout Detection: we have here three distinct access patterns in the source code, captured by the memory traces. Figure 3.7 highlights how they are separated or aggregated to the same array, by computing their intersections, as detailed in Algorithm 2

3.3.1 General Points

Initial data layout specificity lies in its patterns – or strides –, when it comes to the regular case (ie regular strides). Essentially, strides are factors indicating offsets between two consecutive memory accesses, as said before. Factorization allows to exhibit this specificity in the formalism we defined, while simplifying the initial expression and making it easy to manipulate and transform. As we defined it, this factorization constitutes a delinearization of the initial structure, as $\otimes$ is multidimensional, and can be applied on any strided unidimensional layout. In the general case, the memory accesses are given as addresses, linearized and with no multidimensional address. The objective is to discover the different multidimensional layouts used in the code fragment considered. The following describes how to delinearize a single given layout out of the series of layouts obtained by array detection detailed before. The operation is the same for each of them.

The general formulation of the layout, after the array detection, is a sum of n structure layouts, each corresponding to the different memory accesses:

$$base + \oplus_{k=1}^{n} S_{I_k,d} \tag{3.5}$$

where d is the diameter of the layout, and n the number of patterns , potentially significant. However, factorization may enable some simplifications as defined previously, reducing the number of terms of the sum by performing the union operation.

The transformations of this layout into a multidimensional layout resorts to several rewriting rules, creating additional dimensions. These rules are the following:

$$S_{I\times n,p\times n} \rightarrow S_{I,p} \otimes (A_n, i_k) \quad \text{if } I = \{j \times p + k, j \in I, k \in [0,n] \,\big|\, p > n\} \tag{3.6}$$

$$S_{I,p\times n} \rightarrow S_{J,p} \otimes (A_{[\omega+k[,n}, i_k) \quad \text{if } I = \{j \times n + k + \omega, \omega < k, k \in [0,n[\} \tag{3.7}$$

$$S_{I,p\times n} \rightarrow (A_n, i_k) \otimes S_{J,p} \quad \text{if } I = \{k \times p + j, j \in I, k \in [0,n] \,\big|\, p > \#j - 1\} \tag{3.8}$$

$$S_{p\times I+j,p\times n} \rightarrow S_{I,n} \otimes S_{J,p} \tag{3.9}$$

where p is a scalar. These rules represents different ways of factorizing by p. Tuples (A, i) represents arrays with their associated iterator, as retrieved from the trace, that is of importance for the next Chapters.

Intuitively, Rule 3.8 consists in finding a AoS in the initial layout, that is a consecutively repeated stride p, with possibly a smallish offset j. The offset determines the position of the element – its field number – in the internal structure. We give a full example to get a better grasp on rule 3.6 in particular, and at the same time all the rules as well.

Example:

```
1 struct T {  float a,b,c,d; };
2 struct T t[4];
3   for(j=0; j<4; j++) {
4     t[j].a = ...
5     t[j].c = ...
6 }
```

Figure 3.9: Code snippet

```
1   for i1 = 0 to 3
2     val 0x6c4560 + 16*i1
3   endfor
4 // stride 16 must be divided
    by 4 (sizeof(float))
```

Figure 3.10: Trace snippet

Here we are dealing with a structure of 4 fields, resulting in a access pattern with a stride of 4 within the single loop expressed. Therefore layout can be expressed as $base + [0,3] \times 4 + 2$, or again $S_{\{[0,3]*4+2\},16}$. Figure 3.11 below shows how delinearization operates on this example, focusing on access on the c field as depicted in Figure 3.9:

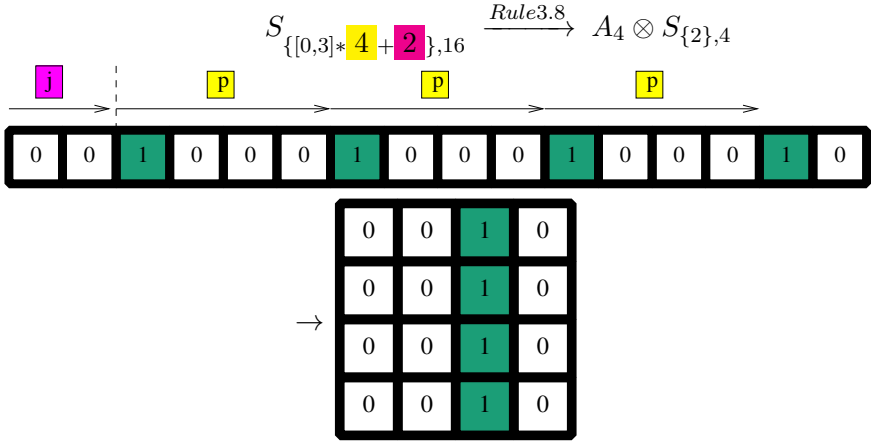


Figure 3.11: Formalism Graphic Representation Example of Rule 3.8

On this case, the stride p indicates the presence of a structure, while the offset j indicates the position of the field inside the structure: we recognize here the AoS, that can be written as a 2-dimensional layout.

More generally, note that after delinearization, S functions are never arrays A. Even after union simplifications and potentially newly found full structures – in the sense that all the fields are accessed– are discovered , an S is never viewed as an A, despite mathematical equivalence. The rewriting rules basically show without ambiguity they are reconstructed structures, this still implies strided patterns, not contiguous access patterns. Another benefit of the factorization is the isolation of arrays, *id est* contiguous memory chunks, that we may want to reorder to our convenience at the time of transformation choices, as we will discuss in the next chapters.

Now this set of rules is used to systematically transform the flat initial layout into an equivalent multidimensional layout. Algorithm 3 explains how to repeatedly apply the rewriting rules in order to find the simplest multidimensional layout.

Algorithm 3: Delinearization

Data: Initial unidimensional layout: $base + \oplus_{k=1}^{n} S_{I_k,d}$
Result: A multidimensional layout, if possible

```
repeat
    foreach rule among (1), (2), (3) do
        If the rule can apply, apply it to all terms of the sum ⊕, at the same position of the ⊗.
until no rule applies ;
```

There is no constraint on the order in which to apply the rules, as long as the rule applies it is applied right away on the input layout formal expression. This property of the rewriting system is

called confluence: there is no preferred order to get to the unique solution. Because the topology is the same regardless of the pattern, when a rule is applied on a given term of a given operand of $\oplus$, it must be applied as well on all the terms. Moreover, because all the rules imply diminishing layout sizes, algorithm is guaranteed to terminate.

As a final word on that topic, if a given layout is intrinsically unidimensional, no rule would be applicable and its structure would stay in its original form.

3.3.2 Properties

We want to show that the product of rewriting rules is a normal form.

Termination Proof Rewriting rules can only produce A or S functions from a given S function. Consequently, as no rule applies on A functions, they cannot be derivated. Structures in the general sense on the other hand are defined such as their diameters is always strictly greater than 1, as a structure of size one is the neutral element of $\otimes$. We want to show that the rewriting rules always produce simpler objects from a structure $S_{I,d}$; producing structures $S_{I',d'}$ equipped with a set I' such as $\#I' < I$, or structures that have a strictly smaller dimension, that is $d' < d$; which implies that the system terminates.

The first 3 rules (3.6, 3.7, 3.8) all similarly divide a structure of size $p \times n$ into two separate structures of respective size p and n. As stated before, both p and n are necessarily strictly greater than 1, therefore both produced structures are smaller than the initial one. The 4th rule (3.9) produces two structures. One of them has a size of $p < p \times n$ which implies it cannot be infinitely derivated. The other has a set I necessarily smaller than the initial one $p \times I + j$, which implies it cannot be infinitely derivated either.

Finally, we have shown that each rules cannot be derivated infinitely, thus the rewriting system terminates.

Confluence Proof We want to show that the rules commute two by two.

Rules 3.6 and 3.8:

- If $J = \emptyset \; and \; p \neq 1$, rule 3.6 is unapplicable.
- If $J = \emptyset \; and \; p = 1$, then I is an interval $[0, n[$ and thus $S_{I,d}$ is actually written as A_n
- If $J \neq \emptyset$, In rule 3.6, $\forall j \in J, \forall k, j \times p > k$, while in rule 3.8, $\forall j' \in J', \forall k, k \times p > j'$. Therefore $J \times p$ and J' are distinct, meaning rules are have distinct domains of application and can be thus be applied in arbitrary order.

Rules 3.8 and 3.9:

- Rule 3.9 is a generalization of Rule 3.8 that applies on structures in the general sense rather than uniquely on arrays. The key is that both rules extract a factor p out of a structure/array, therefore
  - if Rule 3.9 is applied after Rule 3.8, p has already been taken out as it would have using Rule 3.9

– if Rule 3.8 is applied after Rule 3.9, either $S_{I,n} = A_n$ or the array represented by an interval is left untouched by Rule 3.9 and still can be properly transformed by Rule 3.8

Rules 3.6 and 3.7 are mutually exclusive as a scalar ω as defined in 3.7 differs from the two rules.

Therefore, by transitivity, the rewriting system is confluent.

Finally, termination and confluence properties together imply that the rewriting system entails a normal form.

3.3.3 Characterization

We want to shed light on some particular and well-known access patterns that are correctly handled by our delinearization technique.

Structure fields with different sizes Proposed formalism does not take into account element size information – or element type. Structures should not be made of differently sized elements, first because of the induced stride, second because of the potential alignment issue, depending on the ordering of the different field in the structure. Implicitly, here, all elements have the same type that is determined beforehand, at the array detection step. We distinguish instructions by their respective types, this way we segregate memory traces and therefore, layouts. This way, transformed layout will be of only one data type.

Now, structure fields can still be arrays of different sizes. If the structure is not divisible by the array size, then it may be because we are dealing with a structure with arrays of different sizes. First of all, if there is a single lonely element in the structure, it must be extracted just like described before for the heterogeneous data type structures case. Second, the presence of such structure is explicitly handled by Rule 3.7, as shown in the following example.

Example:

```
1 struct s {float a[2], b[m0], c[m1];};
2 struct s A[N×M];
3 ...
4 for (j=0; j<N; j++) {
5   for (i1 = 0; i1 < n1; i1++) {
6     for (i0 = 0; i0 < n0; i0++) {
7       A[j].b[i0] ...
8       A[j].c[i1] ...
9 } } }
```

Figure 3.12: Synthetic example of a case of structure of arrays with different sizes

Which gives, after tracing the two patterns:

$$S_{\{M\times[0,N[+[0,n_0[+2\},NM} \oplus S_{\{M\times[0,N[+[0,n_1[+2+m_0\},NM}$$

Rule 3.8 applies:

$$A_N \otimes S_{\{[0,n_0[+2\},M} \oplus A_N \otimes S_{\{[0,n_1[+2+m_0\},M}$$

Here, the classical SoA Rule 3.6 is inapplicable, due to the offsets within the structure. These are actually structure of different arrays, treated by Rule 3.7:

$$A_N \otimes A_{[2,n_0+2[,M} \\ \oplus A_N \otimes A_{[2+m_0,n_0+2+m_0[,M}$$

This depicts how the formalism expresses structures of arrays of different sizes, they are represented as incomplete arrays, as here $M = 2 + m_0 + m_1$. The structure fields b and c in Figure 3.12 are materialized here by the sum of two layouts.

Scattered structure Rewriting system also handles array of pointers, a class of layouts that can be poorly allocated arrays of arrays. They can also correspond to fixed trees or graphs; were they still in the building phase, our topological approach of layout transformations would not be useful. Array of pointers correspond to the case where the trace of a given instruction is separated in several equivalent chunks. That means multidimensional elements can be separated by any offsets ω_k, it does not affect delinearization. A corresponding given access pattern has the form:

$$base + (\omega_0 = 0) + S_{I,d} + \omega_1 + S_{I,d} + \cdots + \omega_{n-1} + S_{I,d}$$

Which is also

$$base + \bigoplus_{k=0}^{n-1} (\omega_k + S_{I,d})$$

Or again

$$base + \bigoplus_{k=0}^{n-1} (S_{\emptyset,\omega_k} + S_{I,d}) \tag{3.10}$$

Now, NLR algorithm by itself is not able to detect such pattern in the trace. However, it happens that this problem is close to the longest repeated substring problem, the key being to match Equation 3.10 pattern, where the longest repeated substring corresponds to the term $S_{I,d}$. The initial string s is formed by the distance between each access in the trace elements (or addresses) a such as:

$$(s_0, s_1, \ \ldots\ , s_{n-1}) = (a_1, a_2, \ \ldots\ , a_n) - (a_0, a_1, \ \ldots\ , a_{n-1})$$

From s we build the suffix tree [113], that allows efficient resolution of the longest repeated substring problem. Finally, we have actually found an array of pointers pattern if the distance between each substring is exactly the substring size plus 1, which corresponds to the offset ω_k whose value is irrelevant. Note that the need for finding an offset value between the substrings automatically excludes any eventual substring overlapping.

Formalized array of pointers described by Equation 3.10 does not fall under the definition of an array of structures as we defined it, because all the elements here are different due to their respective offsets. However, as we will see in the next section, offsets – chunks of memory where no element is accessed – are removed by compression, which allows us to retrieve the natural array definition. This gives:

$$base + \bigoplus_{k=0}^{n-1} S_{I,d}$$

Which is, using property 3.3:

$$base + A_n \otimes S_{I,d}$$

Non distinct patterns Rule 3.9 allows to extract unnecessary strides out of the arrays expression (=factorization). However, incompressible strides complicate the array expression and may limit transformations opportunities.

Example:

```
1 for (i0 = 0; i0 < n0; i0++) {
2   a[4*i0] = a[28*i0];
3   ...
4 }
```

Figure 3.13: Synthetic example of imperfectly overlapping patterns

Which gives, after delinearization:

$$S_{\{4\times[0,n_0[\},28\times n_0} \\ \oplus S_{\{28\times[0,n_0[\},28\times n_0}$$

Rule 3.9 applies:

$$S_{\{[0,n_0[\},7\times n_0} \otimes S_{\{0\},4} \\ \oplus S_{\{7\times[0,n_0[\},7\times n_0} \otimes S_{\{0\},4}$$

Rule 3.6 and Rule 3.8 apply:

$$S_{\{0\},7} \otimes A_{n_0} \otimes S_{\{0\},4} \\ \oplus A_{n_0} \otimes S_{\{0\},7} \otimes S_{\{0\},4}$$

Stride 7 here is incompressible, the expression can not be simplified any more. This kind of supplementary stride will limit our transformation power – as it should.

3.3.4 Case Study

Let us consider the example code in Figure 3.14, that belongs to a Lattice QCD application.

We want to apply Algorithm 3 to restructure the example trace below:

```
for(iL=0 ; iL < L/2 ; iL+=1) {
  for (j=0;j<4;j++) {
    r0 =  U[idn[4*iL]][0]*tmp[j];
    r0 += U[idn[4*iL]][1]*tmp[n2+j];
    r0 += U[idn[4*iL]][2]*tmp[2*n2+j];
    r1 =  U[idn[4*iL]][3]*tmp[j];
    r1 += U[idn[4*iL]][4]*tmp[n2+j];
    r1 += U[idn[4*iL]][5]*tmp[2*n2+j];
    r2 =  U[idn[4*iL]][6]*tmp[j];
    r2 += U[idn[4*iL]][7]*tmp[n2+j];
    r2 += U[idn[4*iL]][8]*tmp[2*n2+j];
    ID2[j] += r0 ;
    ID2[n2+j] += r1;
    ID2[2*n2+j] += r2;
 } }
```

Figure 3.14: Example of Lattice QCD simulation. The 2D array U uses an indirection. All elements are complex double values. The space iterated by the outer loop is a 4-D space, linearized, and the indirection is used to process the white elements of a 4-D checkerboard.

```
for i0 = 0 to 255
  for i1 = 0 to 255
    val 0x00001000 + 16384*i0 + 32*i1
    val 0x00001008 + 16384*i0 + 32*i1
  endfor
  for i1 = 0 to 255
    val 0x00003010 + 16384*i0 + 32*i1
    val 0x00003018 + 16384*i0 + 32*i1
  endfor
endfor
```

The trace is given as a `for..loop` enumerating addresses and out of simplification, only corresponds to the trace generated by the memory access of matrix U in the first statement, and removing the outer dimension. Removing the scheduling information, we can build the following initial structure corresponding to the values of the trace. Essentially, the iterated domains correspond to the domains of the structures:

$$\begin{aligned} U \quad + \quad & S_{\{2048*[0,255]+4*[0,255]\},d} \\ \oplus \quad & S_{\{2048*[0,255]+4*[0,255]+1\},d} \\ \oplus \quad & S_{\{2048*[0,255]+4*[0,255]+1024+2\},d} \\ \oplus \quad & S_{\{2048*[0,255]+4*[0,255]+1024+3\},d} \end{aligned}$$

Applying Rule 3.9 removes the stride 4:

$$\begin{aligned} U \quad + \quad & S_{\{512*[0,255]+[0,255]\},d} \otimes S_{\{0\},4} \\ \oplus \quad & S_{\{512*[0,255]+[0,255]\},d} \otimes S_{\{1\},4} \\ \oplus \quad & S_{\{512*[0,255]+256+[0,255]\},d} \otimes S_{\{2\},4} \\ \oplus \quad & S_{\{512*[0,255]+256+[0,255]\},d} \otimes S_{\{3\},4} \end{aligned}$$

This can be simplified, by merging the first two lines and the last two ones:

$$\begin{aligned} U \quad &+ \quad S_{\{512*[0,255]+[0,255]\},d} \otimes S_{\{0,1\},4} \\ &\oplus \quad S_{\{512*[0,255]+256+[0,255]\},d} \otimes S_{\{2,3\},4} \end{aligned}$$

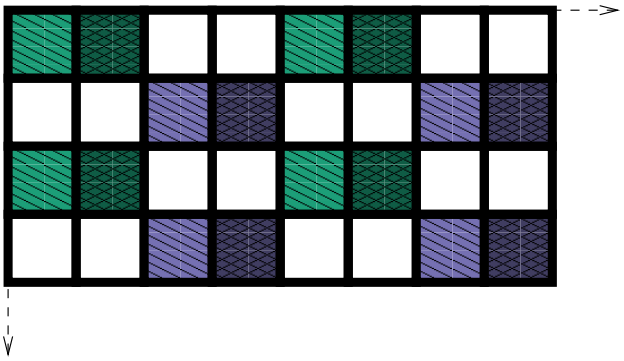
Applying Rule 3.8 transforms the structure into an array of structures:

$$\begin{aligned} U \quad &+ \quad A_{256} \otimes S_{\{[0,255]\},512} \otimes S_{\{0,1\},4} \\ &\oplus \quad A_{256} \otimes S_{\{256+[0,255]\},512} \otimes S_{\{2,3\},4} \end{aligned}$$

Finally, applying Rule 3.6 splits the first structure into a structure of arrays:

$$\begin{aligned} U \quad &+ \quad A_{256} \otimes S_{\{0\},2} \otimes A_{256} \otimes S_{\{0,1\},4} \\ &\oplus \quad A_{256} \otimes S_{\{1\},2} \otimes A_{256} \otimes S_{\{2,3\},4} \end{aligned}$$

This corresponds to an AoSoAoS: This is an array of 2 lines, even lines and odd lines. Even lines have 256 elements that are structures of 4 doubles, using only the first 2. Odd lines have 256 elements having 4 doubles, using only the last 2.



3.4 Conclusion

This chapter focus was on data layout expression in a novel formalism. We proposed an algorithm to detect distinct initial arrays based on memory traces, and we proposed a rewriting system that allows to express any given layout on a normal form, which is a delinearized expression of the initial layout. We also shown on case study that regular structures such as checkerboard patterns can be expressed using our formalism and are in fact AoS of AoS. The next step after array delinearization is the layout transformations exploration.

Chapter 4

Transformations

4.1 Layout Operations 61
4.1.1 Permutation 62
4.1.2 Splitting 63
4.1.3 Compression 64
4.2 Exploration 65
4.2.1 Basic Constraints 65
4.2.2 Locality Constraints 67
4.2.3 Parallelism Constraints 69
4.3 Case Study 71
4.4 Conclusion 75

Finding the best layout transformation is difficult, runtime randomness and everchanging computer architectures along with the unpredictability factor of data structures make it hard to determine *a priori* such a layout. However, it is possible to propose a narrow selection of educated guesses as to which layout transformation is thought to potentially improve performance. This chapter presents the transformations space exploration, that means exploring the most promising layout in terms of locality in the first place, which is the one performance judge we can rely on and can be computed through our proposed formalism. The exploration is also shaped to preferably address layout designs that take advantage of different levels of parallelism.

4.1 Layout Operations

We define in this section basis operations on data layouts. Formally, all correct transformations L' of L are such:

$$|L'^{-1}(1)| = |L^{-1}(1)| \tag{4.1}$$

Where L' and L are layouts as we defined in Section 3.1. This states the first constraint on data layout transformations: the initial layout elements must be preserved. Indeed, the memory traces

approach ensures that all the layout elements are actually used for the computations, since the layout representation is based exclusively on data actually processed. Whereas on a compiler-side approach, possibly unused – ever – layout elements cannot be spotted and evicted. However, it is more likely that the possible gaps in a layout are artifacts of the memory traces, in the sense that they are meant to be used, but just not for the particular run the memory traces capture. This entails uncertainty on the appropriate transformation to apply: should the gaps be preserved? Should they be erased by default? One more note on the formulation of correct transformations (Equation 4.1), redundant layouts – *ie* layouts composed of some duplicated elements – are not considered for the moment.

Now, actual transformation operations need to be properly characterized. Indeed, it is necessary to define systematic operations allowing a wide set of new layouts to be constructed. Moreover, it is the sequence of transformations that defines the data mapping over the new layout, which is essential for transformations code generation. For this matter, we introduce in Table 4.1 a rewriting system describing given layout operations we want to perform.

Permutation	$L \otimes L' \rightarrow L' \otimes L$
Splitting	$A_{n \times m} \rightarrow A_n \otimes A_m$
Compression	$S_{I,d} \rightarrow S_{I',d'},\ if \#I = \#I' and\ d' < d$

Table 4.1: Layout transformations rewriting system

This rewriting system reflects that our approach focus is on layout topology changes, consistently with the delinearization system proposed in Section 3.3, where the layout topology is defined by the number and the size of the layout respective dimensions. It particularly means that transformations which swap individual array elements are not considered, as we assume all the structure fields are hot so the swapping have no impact on actual layout performance. Let us now expand more thoroughly on the mentioned operations.

4.1.1 Permutation

High performance can only be achieved through locality in the data access patterns. Therefore we need to explicit a way of properly altering locality, meaning reordering memory blocks, in hope to optimize memory accesses. To do so, we define the permutation operation as the following:

$$L \otimes L' \rightarrow L' \otimes L \tag{4.2}$$

A lot of locality optimizations can be expressed as layout permutations. An intuitive example is the matrix transposition. Let M be a 2-dimensional matrix of size $n \times m$, it can be expressed as:

$$M = A_n \otimes A_m$$

The transposed matrix is the permutation of the two terms, which gives a new matrix M^T:

$$M^T = A_m \otimes A_n$$

This is also the operation that expresses formally the common AoS (Array of Structures) to SoA (Structures of Array) transformation, crucial on nowadays architectures and thus subject of numerous research efforts (see Chapter 1). Suppose the following excerpt:

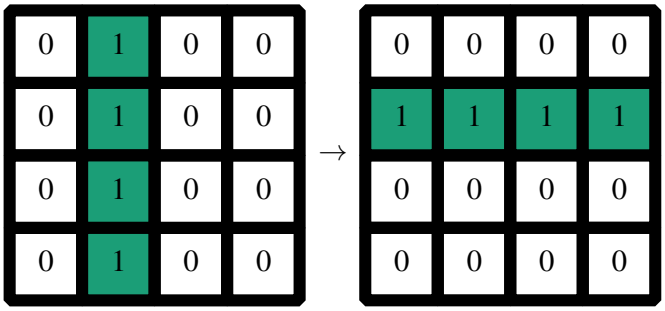
```
struct s {float a, b, c, d;};
struct s A[4];
...
for(i = 0; i < 4; i++)  {
    A[i].b = ...
}
```

Then the layout obtained *via* delinearization is an AoS expressed as:

$$A_4 \otimes S_{\{1\},4}$$

Where the outer dimension is the explicitly defined array of structure of 4 elements A. The inner dimension is a structure s that has only one element accessed, it is the field nicknamed b. The transformation to SoA is a permutation that optimizes locality as graphically shown below

$$A_4 \otimes S_{\{1\},4} \xrightarrow{permutation} S_{\{1\},4} \otimes A_4$$



Where the horizontal layers represent contiguous chunks of memory. The layout expression after permutation is the following:

$$S_{\{1\},4} \otimes A_4$$

And the resulting transformed code can be written as:

```
struct s {float a[4], b[4], c[4], d[4];};
struct s A;
...
for(i = 0; i < 4; i++)  {
    A.b[i] = ...
}
```

4.1.2 Splitting

Another way of optimizing locality is to carefully change the different array dimensions sizes, in order to take advantage of different levels of parallelism such as SIMD-level or thread-level parallelism for instance. Indeed, if applicable, one may be wanting to reshape the initial array to have a small contiguous innermost dimension to allow efficient SIMDization, and similarly have an outer dimension large enough to be conveniently parallelized with threads for instance. Moreover, reshaping layout dimensions according to the knowledge of architecure such as cache sizes etc. also can have massive impact on performance.

The splitting operation allows us to divide a given dimension into smaller ones:

$$A_{n \times m} \rightarrow A_n \otimes A_m$$

Note that this is a parameterized transformation, in the sense that n and m have to be chosen, among the values that divide $n \times m$ obviously as the array size must be an integer greater than 1. A concrete example is the transformation from AoS to AoSoA (Array of Structure of Arrays). The idea behind AoSoAs is to take profit of recent architectures such as Intel MIC [1]or more generally any architecture with SIMD units.

$$\begin{aligned} & A_{256} \otimes S_{[0,3],4} \\ \xrightarrow{splitting} \quad & A_{64} \otimes A_4 \otimes S_{[0,3],4} \\ \xrightarrow{permutation} \quad & A_{64} \otimes S_{[0,3],4} \otimes A_4 \end{aligned}$$

The resulting layout now have an inner dimension suitable to SIMD parallelization.

We define the inverse operation to splitting that is the merger operation, that, even though it doesn't affect physically the topology, allows mathematical simplification of the layout formulation:

$$S_{I,n} \otimes S_{J,m} \rightarrow S_{I',n \times m},\ if \#I' = \#I \times \#J$$

4.1.3 Compression

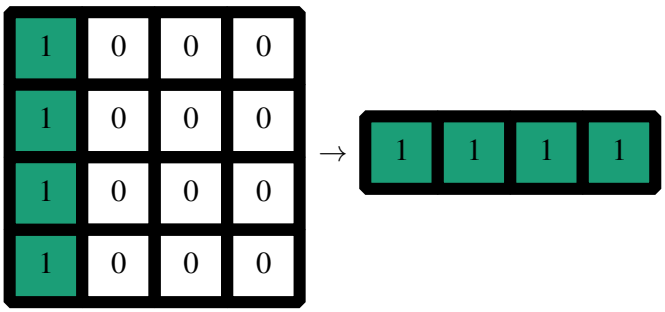
Initial layout detected may be composed of unused elements, that we may want to erase in order to improve locality. Structure compression is defined as the following:

$$S_{I,d} \rightarrow S_{I',d'}, \#I = \#I', d > d'$$

Where the elements of I' are remapped in order to not leave unused space in the structure.

Example:

$$A_4 \otimes S_{\{0\},4} \xrightarrow{compression} A_4$$



As an AoS with only one structure field used is inherently simply just one array. We define the inverse operation to compression which is the expansion, aiming at performing padding, which is basically filling structures with blanks, in order to make them fit better in the cache:

$$S_{I,d} \rightarrow S_{I',d'}, \#I = \#I', d < d'$$

[1] https://software.intel.com/en-us/articles/memory-layout-transformations

4.2 Exploration

Transformations rewriting system 4.1 in itself is not terminating. This is due to the fact that infinite sequences such as the following exist:

$$L \xrightarrow{permutation} L' \xrightarrow{permutation} L \xrightarrow{permutation} L' \dots$$

This highlights the need for an algorithm to constrain transformation sequences. Moreover, while the number of different transformed layouts $\{L''\}$ of L is finite, we can notice that:

- Number of permutations of a finite set of element is finite
- Number of divisors of an integer is finite
- Number of zeros in a structure is finite

It still remains too large for evaluation purposes, as the number of permutations may be large, the number of splitting as well and numerous equivalent transformations or even useless transformations need to be pruned. In practice, rewriting system 4.1 is already constrained by the original layout topology, because its topology may not allow certain given transformations. As an example, $S_{\{0,2,1,3\},4}$ does not allow any transformation at all. Let us assume they are all applicable. This is the worst case scenario for the exploration, and the best from the user point of view because this means there are a lot of restructuring possibilities.

4.2.1 Basic Constraints

It is necessary to narrow down the exploration space, therefore evicting useless transformations and equivalent transformations is important. Indeed, the consecutive application of given transformations of 4.1 constitutes a path. In fact, there are potentially several ways to pass from a layout A to a layout B, as depicted in Figure 4.1. Different paths induce different code transformations and different element distribution. Algorithm 4 details how we reduce exploration space to a reasonable set.

We allow only one Split operation for each structure/dimension of the layout, with fixed parameters. The split operation is not meant to be performed by itself, because both the initial and resulting topologies are equivalent as shown in example below:

$$A_N \otimes A_M \rightarrow A_{NM}$$

Indeed, there is no topological difference between a 2-dimensional array layout and a unidimensional array layout, what makes the topology is the structure/array alternation. It is the permutations that allow actual topological reshaping. Now, it is necessary to limit the number of split because having a large number of array/dimension fragment is not beneficial and a lot of combinations entail equivalent layout, by merger simplification. As an example, let us consider the following transformation:

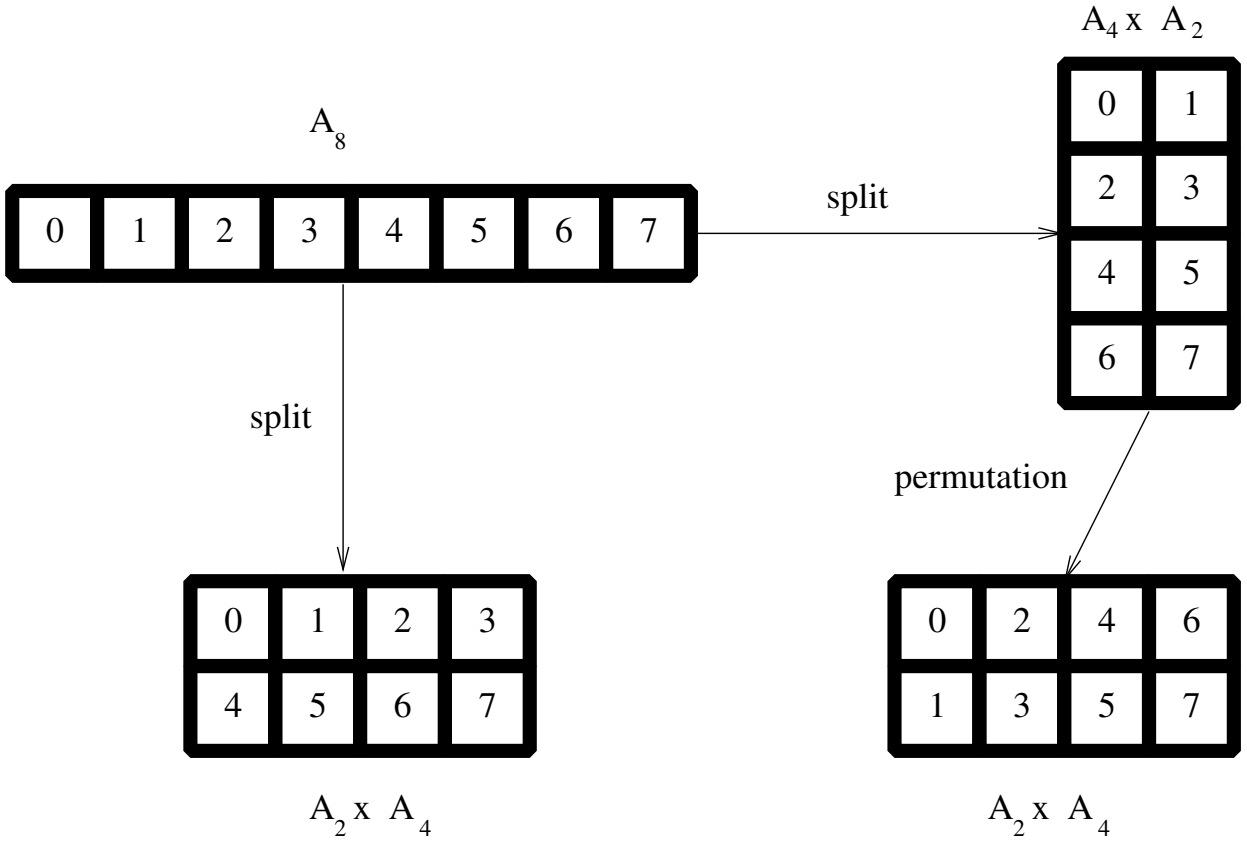


Figure 4.1: Several paths lead to the same topology, with a different element ordering — Elements distribution is irrelevant in the context of our study as we assume all the fields are hot fields (therefore swapping consecutive elements does not influence performance) — longer paths mean more complex code transformation

$$
\begin{aligned}
& A_{LNM} \\
\xrightarrow{split} & A_L \otimes A_{NM} \\
\xrightarrow{split} & A_L \otimes A_N \otimes A_M \\
\xrightarrow{permutation} & A_L \otimes A_M \otimes A_N \\
\xrightarrow{merger} & A_L \otimes A_{MN} \\
\xrightarrow{merger} & A_{LMN}
\end{aligned}
$$

This shows an useless path induced by too many splits. Here, at each step layouts are equivalent to one another in a topological sense. Moreover, split creates new dimensions, which implies increased combinatorials, Split operation is mainly thought to enhance parallelism, by constructing an adequately sized array for SIMD or thread-level paralellism for instance. Thus we judge it not necessary to try an extensive set of possibilities.

Compression is performed once per dimension, and we only consider the case where all zeros are removed, as we assume for now more zeros mean less performance. We do not deal with the padding case. This decreases the number of paths and useless transformations significantly as well.

Finally, that leaves the combinatorial analysis on permutations only, that depends on the number of dimensions of the layout.

Algorithm 4: Basic Layout Transformation Space Exploration

Data: $L = S_n \otimes \cdots \otimes S_1$: Initial multidimensional layout,
S_λ: structures to split with fixed parameters
Result: $\{L'\}$: Set of transformed layouts from L

```
foreach term S_k in L do
    compression(S_k)

foreach term S_λ in L do
    split(S_k)

L' ← {permutation(L)}
```

4.2.2 Locality Constraints

We have bounded the exploration with basic constraints, however the set of potential transformed layouts remains too large, mostly because of the combinatorials, and needs to be shortened. To do so, we introduce supplementary constraints to help the making of a reduced relevant set of solutions. Locality is the key notion when it comes to data layout optimization. Although the perfect layout is not known in advance due to various noises as discussed earlier, we present here the intuition that can help us make educated guesses on which layouts may improve performance. Let us now elaborate on the three locality-related aspects at stake here:

- The notion of access order $\vec{s_0}$
- The inter-iteration stride $\vec{s_h}$
- The intra-iteration stride $\vec{s_v}$

The objective is to minimize the triplet $\sigma = (\vec{s_0}, \vec{s_h}, \vec{s_v})$, *ie* a given transformed layout has a near-optimal locality property if $\sigma \to 1$. Aforementioned triplet allows us to rate any given layout in terms of locality, then the idea is to select an arbitrary restricted number of the best rated layouts to evaluate for performance.

Access Order

More than the topology itself, the order elements are accessed must be taken into account when designing an optimized layout. Indeed, consecutively accessed elements need to be in close proximity to ensure high performance. This order is explicitely given by the memory traces. As an example, let us consider the following trace:

```
1 for i1 = 0 to 999
2   for i0 = 0 to 49999
3     val 0x6c4560 + 4*i1 + 200000*i0
4   endfor
5 endfor
```

Which, after delinearization, entails:

$$(A_{1000}, i_0) \otimes (A_{50000}, i_1)$$

Here the performance issue lies in the fact that the outer dimension is accessed first, given i_0 is the deepest iterator, which causes each element to be accessed with a penalty. An intuitive transformation is the matrix transpose:

$$(A_{50000}, i_1) \otimes (A_{1000}, i_0)$$

where the deepest iterator i_0 is affected to the deepest dimension, meaning elements are now accessed in order. Consequently, our aim is to transform layouts in a way that preserves natural access order, that is to sort the dimension by iterator number.

Finally, let us explicit $\vec{s_0}$:

$$s_{0,j} = \begin{cases} 1, if\ i_k \leq j \\ \#i_{j-1} \times \cdots \times \#i_1, else \end{cases}$$

Note that loop bounds can be parametric. Let us now consider a couple examples.

Example 1:

$$(A_{N_3}, i_3) \otimes (A_{N_2}, i_2) \otimes (A_{N_1}, i_1) \otimes (A_{N_0}, i_0)$$

That makes:

$$\vec{s_0} = (1, 1, 1, 1)$$

Which represents a perfect access order.

Example 2:

$$(A_{N_0}, i_0) \otimes (A_{N_1}, i_1) \otimes (A_{N_3}, i_3) \otimes (A_{N_2}, i_2)$$

That makes:

$$\vec{s_0} = (1, 1, N_3 \times N_2, N_3 \times N_2 \times N_1)$$

Which implies the considered layout has a costly alteration of the access order, induced by large strides.

Intra-iteration Stride

It is the distance between the elements accessed within the same iteration. To optimize locality, this distance must be small.

$$s_{v,j} = \begin{cases} 1 & , if\ j = 1\ or\ A_{j-1}\ !exists \\ \#A_{j-1} \times 2/P(j-1) & ,\ else\ if\ j-1 = 2\ and\ innermost = S_c \\ \#A_{j-1}/P(j-1) & , otherwise \end{cases}$$

Where S_c represents the complex number structure and where $P(dimension)$ expresses the level of parallelism on the given dimension: it is the scalar equivalent to the number of elements processed simultaneously. For instance, if on a given architecture SIMD vectors size is v, then $P(1) = v$, where 1 represent the innermost dimension, where the SIMD instructions apply. Intuitively, if arrays inside structures are too large to be entirely processed in parallel, they will degrade locality especially if they are significantly large. Such a non-optimal locality property is typically the case on CPU where vectors are small in front of the arrays of a SoA, and where consequently the intra-interation strides are significant.

Inter-iteration Stride

This is the distance between two memory addresses accessed by a given instruction at iteration n and $n+1$. If all addresses between the two are accessed in the meantime, there is no locality issue. Gaps in the layout tend to increase dramatically this number, therefore it is important to have an indicator of such effect in the context of our transformation space exploration.

$$s_{h,j} = d_{j-1} - \#I_{j-1}$$

$\vec{s_h}$ allows to measure locality breaches over a given layout.

4.2.3 Parallelism Constraints

Because of architectures complexity, more contraints are highly relevant to our space exploration, in particular in regards to parallelism. We detail basic performance requirements for data layout transformations in multithread and SIMD contexts.

Multithread

Locality constraints also permit to avoid low performing multithread work share. Indeed, it is crucial not to accidently make an inappropriate layout for thread-level parallelism, as performance variation may be significant.

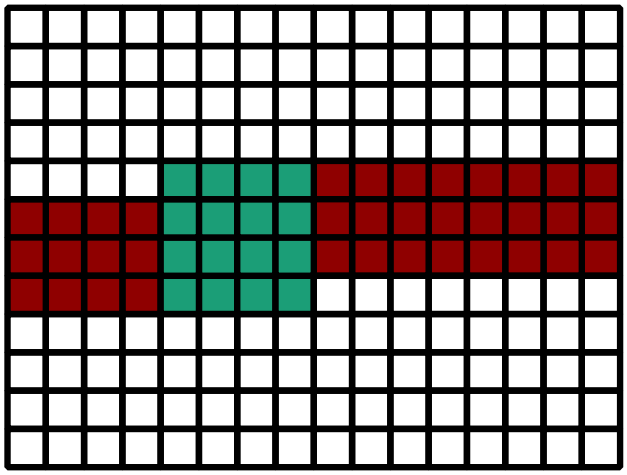


Figure 4.2: A given thread t accessing only a block of the layout (green elements). There is a penalty for passing from a line to the other as locality is breached (red elements are unused)

Careless loop parallelisation may induce thread blocking effects, for instance when the parallelisation is not applied on the highest-level dimension. Thread blocking constitutes a locality breach, as it is shown in Figure 4.2 on layout L. Each memory access thread affiliation information is unambiguously made available by the memory traces. Thread t layout chunk L_T of initial layout L is given as:

$$L_T = S_{\{1,Y\}} \otimes A_y \otimes S_{\{1,X\}} \otimes A_x$$

Where the dimension of L are $X \times Y$ and the dimensions of L_T are $x \times y$. While it is conceivable to perform customized transformations tailored for each thread and a given thread pool, we choose not to deal with thread-level layout parameterizability and to also offer less painful transformations for the user, by viewing the initial layout as one chunk which means disregarding multithread affiliation. Therefore, while we are able to report the user thread blocking effects and suggest him to revise his loop parallelisation, we assume it is not an inherent layout issue and do not make a special case. However, for the sake of parallel distribution, we will prioritize layouts that exhibit a large outer dimension, that is layouts on the form $A_X \otimes L$ where X is arbitrarily large.

SIMD

SIMD itself brings another constraint to our space exploration, because to be efficient it requires the transformed layout to be on the form $L \otimes A_v$ where v is the vector size. However, different layouts accessed within the kernel of interest may be treated separately in the context of vectorization. Indeed, to benefit the most of vectorization, layouts may need to be transformed accordingly

to avoid using additional assembly in order to reorder the values in the right registers and the right memory places. In fact, it is difficult to determine beforehand if it would be advantageous to also transform the other layouts accordingly. Therefore, it may be interesting to evaluate both approaches.

Let us now expand on how we transform other layouts L_o accordingly to the layout of interest L transformation. First of all, if the layout L_o is read-only, then it is not useful to modify the layout, a couple of assembly instructions such as unpacks will deal with it conveniently. Otherwise, we have a few more restructuring options. Outer loop vectorization can be a powerful ally in particular when the innermost loop iterations can be executed in parallel. However operations on vectors accessed exclusively by the innermost loop may be tricky to perform with SIMD instructions, it may be smart to restructure them in order to effectively vectorize the kernel.

Consider the following example:

$$\begin{cases} L = (A_L, i_1) \otimes (S_{dmc}, i_0) \\ L_o = (A_{12}, i_0) \end{cases}$$

As it stands, L_o cannot be directly manipulated with packed instructions. A possible transformation would be:

$$\begin{cases} L = (A_{L/v}, i_1) \otimes (S_{dmc}, i_0) \otimes (A_v, i'_1) \\ L_o = (A_{12}, i_0) \otimes (A_v, i'_1) \end{cases}$$

Where the second array has been allocated v more space. This is the classic computations versus memory compromise, which we propose to address by our profiling methodology. Disposing of more memory can be determinant for performance especially if the kernel is significantly large, as such a secondary array can namely serve as a temporary buffer.

4.3 Case Study

We propose to explicit transformations application on real-life codes layouts. Their respective performance improvements are addressed in Chapter 6.

Lattice QCD Layout Transformation

In quantum physics, Lattice QCD is an approach to compute the strong interaction as modeled by quantum chromodynamics (QCD) in the standard model of physics, where quarks and gluons interact via a color charge, analoguous to the electromagnetic charge. The lattice is the discretization of spacetime into a 4-dimensional grid. We study here codes expressed through Qiral [7], a Domain Specific Language (DSL) that allows to ease the expression of parallelism on Lattice QCD simulations.

We focus here on the main array U (see code in Figure 4.3) – the lattice – in the sense that it is large enough, intensely accessed and a significant amount of large stride issues are reported, therefore it constitutes an interesting candidate for data restructuring. However, since we study transformations on U and only U, the two other arrays this kernel operates on (*CRNEAp*, *CRNEp*) and worth mentioning due to their size — 1/8th of U both — are left untouched. As a consequence, full kernel parallelism cannot be achieved. Nevertheless, significant performance gain opportunities remain, as shown by the results presented in Section 6.2. U is read-only, therefore a

layout transformation limited to the scope of the function would require a copy-in, but no copy-out. The copy-in is the operation that consists in copying the data from the old layout to the new one, while the copy-out is the inverted operation. In this case, copy cost impact is minimized since copy-in can be performed at the actual array declaration. We study two different numerical schemes, generating different memory access patterns.

```
1  // double complex* U[L][d][m];
2
3  for(iL = 0; iL < L; iL ++)
4  {
5    xgemm(0,1,3,4,4,1,&CRNEp[sup(iL, dx)*12],li8,0,tmp);
6    xgemmfast(0,0,3,4,3,1,U[uup(iL, dx)],tmp,0,ID2);
7    ...
8    xcopy(12, ID1, ID2) ;
9    xscal(12, - kappa, ID1) ;
10   xgemm(0,1,3,4,4,1,&CRNEp[sdn(iL, dx)*12],li2,0,tmp);
11   xgemmfast(0,0,3,4,3,1,U[udn(iL, dx)],tmp,0,ID2);
12   ...
13   xcopy(12, ID15, ID2) ;
14   xscal(12, - kappa, ID15) ;
15   xgemm(0,1,3,4,4,1,&CRNEp[iL*12],li5,0,ID29);
16   xcopy(12, &CRNEAp[iL*12], ID29) ;
17
18   xaxpy(12,1,ID15,&CRNEAp[iL*12]);
19   xaxpy(12,1,ID1,&CRNEAp[iL*12]);
20 }
```

Figure 4.3: Code except from Lattice QCD simulation. We study layout transformation on main array U

Qiral benchmark without preconditioning According to the memory traces analysis, the accesses to U are given by

$$A_L \otimes S_{\{1\},d} \otimes S_m \otimes S_c$$

where $L = 131072, d = 8, m = 9, c = 2$. The innermost structure corresponds to complex, double numbers.

A quick glimpse on U general structure shape highlights two main performance issues. First of all, the size of the 3-dimensional inner multi-level structure imply significant (= $d \times m \times c$) strides, thus we can assume spacial locality may be improved by restructuring the layout. Second, AoS-fashion general structure prevents efficient SIMDization, working on contiguity might enable automatic SIMDization, or at least allow efficient SIMDization. We can also notice that only 1 out of 8 elements is accessed in the superstructure.

For reference, let us explicit σ to characterize the non-optimality of locality of U.

- $\vec{s_0} = 1$, as there is only one array layout dimension and the loop is perfectly nested.
- $\vec{s_v} = 1$, as the only array layout is on the outer dimension
- $\vec{s_h} = (7, 1, 1)$, because of the gap on the last structure layout dimension.

This shows that locality, apart from the significant gap provoked by the not fully utilized structure layout $S_{\{1\},d}$, can be viewed as satisfying. However, the inner multi-level structure that inhibits vectorization is an issue the transformation should definitely address.

Each transformation we propose removes memory gaps formed by unused elements — in the context of the instance evaluated — therefore the resulting layouts are 1/8th the size of U. Consequently, $\vec{s_h} = 1$. We study four specific layout transformations, implementing different flavours of SoAs and AoSoAs. *AoSoA-cplx* and *SoA-cplx* which preserve complex number structure, *id est* they keep real and imaginary parts of a given complex number contiguous, whereas *AoSoA-dbl* and *SoA-dbl* are layouts oblivious to complex numbers. We make sure they all guarantee perfect access order, *ie* $\vec{s_0} = 1$ by enforcing iterator lexicographic order. Thus the only judge of locality left to decide between them is $\vec{s_v}$, that we will explicit for each transformation.

AoSoA-dbl transformation is given by:

$$\begin{aligned} & A_L \otimes S_{\{1\},d} \otimes S_m \otimes S_c \\ \xrightarrow{compression} & A_L \otimes S_m \otimes S_c \\ \xrightarrow{merger} & A_L \otimes S_{m\times c} \\ \xrightarrow{reshaping} & A_{L/v} \otimes A_v \otimes S_{m\times c} \\ \xrightarrow{permutation} & \boxed{A_{L/v} \otimes S_{m\times c} \otimes A_v} \end{aligned}$$

Which gives:

$$\vec{s_h} = (v/P(v)) = (1)$$

AoSoA-cplx transformation is given by:

$$\begin{aligned} & A_L \otimes S_{\{1\},d} \otimes S_m \otimes S_c \\ \xrightarrow{compression} & A_L \otimes S_m \otimes S_c \\ \xrightarrow{reshaping} & A_{L/v} \otimes A_v \otimes S_m \otimes S_c \\ \xrightarrow{permutation} & \boxed{A_{L/v} \otimes S_m \otimes A_v \otimes S_c} \end{aligned}$$

Which gives:

$$\vec{s_v} = (1, v \times c/P(v)) = (1, 2)$$

SoA-dbl transformation is given by:

$$\begin{aligned} & A_L \otimes S_{\{1\},d} \otimes S_m \otimes S_c \\ \xrightarrow{compression} & A_L \otimes S_m \otimes S_c \\ \xrightarrow{merger} & A_L \otimes S_{m\times c} \\ \xrightarrow{permutation} & \boxed{S_{m\times c} \otimes A_L} \end{aligned}$$

Which gives:

$$\vec{s_v} = (L/P(v))$$

SoA-cplx transformation is given by:

$$\begin{aligned} & A_L \otimes S_{\{1\},d} \otimes S_m \otimes S_c \\ \xrightarrow{compression} & A_L \otimes S_m \otimes S_c \\ \xrightarrow{permutation} & \boxed{S_m \otimes A_L \otimes S_c} \end{aligned}$$

Which gives:

$$\vec{s_v} = (L \times 2/P(v))$$

To sum up, for CPU vectorization in particular, the degree of parallelism P(v) is typically small in front of L, which implies $\vec{s_v} >> 1$. Thus AoSoA transformation are expected to perform better than SoA on this kind of architectures.

Qiral benchmark with even/odd preconditioning The preconditioning considered iterates on one out of 2 values of the lattice, but in a checkerboard fashion: only the white positions of a 4D checkerboard are visited. According to memory traces analysis described before and applied to this application, memory accesses to U are given by:

$$\bigoplus_{x,y,z,t \equiv 0[2]} \left(\bigotimes_{k=x,y,z,t} A_l \otimes S_{k,2} \right)$$

Here, what initially appeared to be an 1-dimensional array of L elements on the outermost dimension is in fact a 4-dimensional array, sparsely processed by its patterns as only one element out of two are accessed. Indeed, its basis addresses form a 4-dimensional checkerboard-style pattern, so half of the initial layout is used in this case. Again, because there is only one base on the d-elements dimension, $1/8$th of this half is actually processed. Since $16^4 = L/2$, the access function can also be written as:

$$\begin{aligned} & A_{L/2} \otimes S_{I,2} \otimes S_{\{1\},d} \otimes S_m \otimes S_c \\ \xrightarrow{compression} & \boxed{A_{L/2} \otimes S_{\{1\},d} \otimes S_m \otimes S_c} \end{aligned}$$

Therefore, proposed transformations for this benchmark are the same as the one before.

Qiral application without preconditioning According to memory traces analysis described before and applied to Qiral, access is given by:

$$A_L \otimes S_d \otimes S_m \otimes S_c$$

and encounters similar difficulties as the benchmarks studied before, as the inner superstructure still forms a SIMDization-challenging gap. We study two transformations.

AoSoA-fashion transformation:

$$\begin{aligned} & A_L \otimes S_d \otimes S_m \otimes S_c \\ \xrightarrow{merger} & A_L \otimes S_d \otimes S_{m \times c} \\ \xrightarrow{reshaping} & A_{L/v} \otimes A_v \otimes S_d \otimes S_{m \times c} \\ \xrightarrow{permutations} & \boxed{A_{L/v} \otimes S_d \otimes S_{m \times c} \otimes A_v} \end{aligned}$$

SoA-fashion transformation:

$$\begin{aligned} & A_L \otimes S_d \otimes S_m \otimes S_c \\ \xrightarrow{merger} & A_L \otimes S_d \otimes S_{m \times c} \\ \xrightarrow{permutations} & \boxed{S_{m \times c} \otimes S_d \otimes A_L} \end{aligned}$$

```
1  for(Xstep = 1;Xstep<Nx+1;Xstep++)
2  {
3      for (Ystep = 1; Ystep < Ny+1; Ystep++)
4      {
5          ...
6          Vm=datarr[Xstep][Ystep][0][(step-1)%2];
7          dVmdt=datarr[Xstep][Ystep][1][(step-1)%2];
8          IK1=datarr[Xstep][Ystep][2][(step-1)%2];
9          x1=datarr[Xstep][Ystep][4][(step-1)%2];
10         INa=datarr[Xstep][Ystep][5][(step-1)%2];
11         m=datarr[Xstep][Ystep][6][(step-1)%2];
12         h=datarr[Xstep][Ystep][7][(step-1)%2];
13         Is=datarr[Xstep][Ystep][8][(step-1)%2];
14         d=datarr[Xstep][Ystep][9][(step-1)%2];
15         f=datarr[Xstep][Ystep][10][(step-1)%2];
16         Cai=datarr[Xstep][Ystep][11][(step-1)%2];
17         Isum=datarr[Xstep][Ystep][12][(step-1)%2];
18         Diff=datarr[Xstep][Ystep][13][(step%2)];
19         Istim=datarr[Xstep][Ystep][14][(step%2)];
20         ...
```

Figure 4.4: Cardiac wave simulation [112]

2D cardiac wave propagation simulation application

We focus on an application [112] whose main kernel code is given in Figure 4.4. The kernel we chose to study uses two arrays, input U and input/output O, where most of the accesses are performed in a stencil fashion on U which is the array we will examine here. According to memory traces analysis described before and applied to this benchmark, access is given by:

$$A_X \otimes A_X \otimes S_{\{1\},a} \otimes S_{\{1\},s}$$

where $a = 15, s = 2$ and where the output matrix O is initially encapsulated in original layout — at the 14th field of the a-elements structure, while the rest of the data lies in the 0th field — therefore suffering the same stride issues. We restructure it as well into a distinct similar array. Stencil layout actually hides an inner superstructure of size $a \times s$, which implies a large stride and possibilities of performance improvement. Parameter X vary with the dataset and its value is one of $\{256, 512, 1024\}$ for the datasets we chose to study. Copy-in can be placed as early as the array initialization.

We study one transformation:

$$\begin{aligned} & A_X \otimes A_X \otimes S_{\{1\},a} \otimes S_{\{1\},s} \\ \xrightarrow{compression} & A_X \otimes A_X \otimes S_{\{1\},a} \\ \xrightarrow{compression} & \boxed{A_X \otimes A_X} \end{aligned}$$

4.4 Conclusion

We covered the different existing layout transformations; some of them are popular ones and already acknowledged for their benefits; explaining their uses and how they are expressed in our data layout formalism. Transformations space exploration is driven by two fundamental constraints. The first, locality, is meant to ensure all data is properly used in an efficient way. The

second is parallelism. We guide the exploration in such way to exhibit SIMD-friendly layouts in order to obtain the most of the computers performance. We also ensure that multithread distribution is appropriately done so the locality property is not degraded and the layout is processed efficiently over multiple cores. For better understanding, we also detailed examples of transformations on case study, using two real-life application codes, and we characterized their relevance in the context of transformation space exploration. By nature, said formalism can be manipulated conveniently to express transformation paths, which encode the information of how the related user source code transformations should be written, this is the topic of the following chapter.

Chapter 5

Code Rewriting and User Feedback

5.1 Systematic Code Rewriting . . . 77
5.1.1 On locality . . . 78
5.1.2 Formalism Interpretation . . . 79
5.1.3 Copying . . . 81
5.1.4 Remapping . . . 83
5.2 User Feedback . . . 84
5.2.1 Layout issues pinpointing . . . 85
5.2.2 Hinting the rewriting . . . 85
5.3 Low-level implementation . . . 86
5.3.1 Loop kernel rewriting . . . 86
5.3.2 SIMDization . . . 87
5.4 Conclusion . . . 88

Formal abstraction introduced in the previous chapters allows us to rewrite initial application code and seek high performance improved versions. Considering a given formal transformation, two generations are applied separately. First, we propose pseudo-C language code snippets of proposed transformations, providing valuable feedback. Second, we create mock-up codes serving evaluation purposes via binary rewriting. Evaluation results are complementing user feedback by quantifying each given transformation, providing a reliable gain estimation for the programmer to make an appropriate decision in terms of application code rewriting.

5.1 Systematic Code Rewriting

This chapter discusses the implementation of the restructuring strategies detailed in the previous chapters. Each of the selected layout transformations has to be generated for evaluation of its potential, but also as it is automatically generated, it is possible to notify the programmer about the aspects of the transformation, meaning not only transmitting the main idea behind but also precise individual memory instruction rewriting.

The formalism we defined in the previous chapters, that abstracts data structures, gives a complete overview of the stride issues, and can be translated to pseudo-C in a straightforward manner. As for evaluation code versions themselves, they are implemented through mock-up codes, which are binary versions rewritten from the original user application binary, which are not totally semantically equivalent to the initial code. The sole purpose of their existence is to predict the impact of given layout transformations on kernel performance. Preservation of semantics can not be guaranteed as the approach is based on memory traces, which reflect only the memory accesses performed on a given run, or more precisely, a given instance of a given function. However, our goal is to tend to actual user rewritten kernel behavior and performances, so we try to tend close to the initial version semantics.

5.1.1 On locality

The code transformation, and the evaluation that comes with it, is local. Indeed, memory traces collection is costly in terms of elapsed time. Generally, it entails $100\times$ of slowdown factor for MAQAO instrumentation for instance, which operates directly on the binary file and it is the least intrusive method for probing memory accesses. That is why instrumentation is usually performed on a restricted scope, namely a given function of interest. Consequently, our visibility is constrained to the function scope only, as opposed to compiler visibility that is the entire program, minus the runtime information the memory traces contains. Another difference with the traditional compiler transformation approach is that we perform code rewriting, as opposed to compiler code generation. The difference lies in the fact that the compiler disposes of the immaculate source code and a wide set of optimizations to apply, while in our case, the input of our framework is a binary file already featuring numerous compiler optimizations. Therefore, the degree of freedom is lesser, as we keep the initial binary code skeleton and rewrite inside. Besides, aforementioned compiler optimizations can constitute noise to our approach, as it renders the binary less readable, in the sense that it may be harder to analyse through top-notch optimizations, an example is the unrolling that multiply instructions within the loop nest considered and complicate both the analysis and generation that prefer factorized code. Another example is the SIMDization, compilers try to vectorize as they can suboptimized layout accesses, by sometimes performing partial loads and such, requiring analysis to first unvectorize the code, before performing actual layout transformations, and possibly revectorizing afterwards if applicable. As said before, our binary-based approach implies using code already optimized by a compiler. On the other hand layout transformation is independent to control optimizations, and as we operate well after a mixture of compiler control optimization, we are not concerned by the pass order issue, namely the fact that compiler passes best order of application is unknown in the general case. Therefore proposed codes are better or worse than initial code, depending only on the layout proposed.

We distinguish three major aspects of code modifications, as depicted in Figure 5.1. First of all, we need to retrieve the data from the original layout and map it onto the new layout, as our scope is as wide as the function of interest only, we can not assume we have control on the original layout allocation, often it will be allocated earlier in the program. One way to cope with that is to copy the data into a newly allocated structure. This operation is named *copy-in*, while the reverse operation, which consists in storing the new data back into the original layout, is called *copy-out*. The later permits to get back to the normal application functioning as we leave the function and consequently our scope of optimization. Second, the loop nest structure has to be

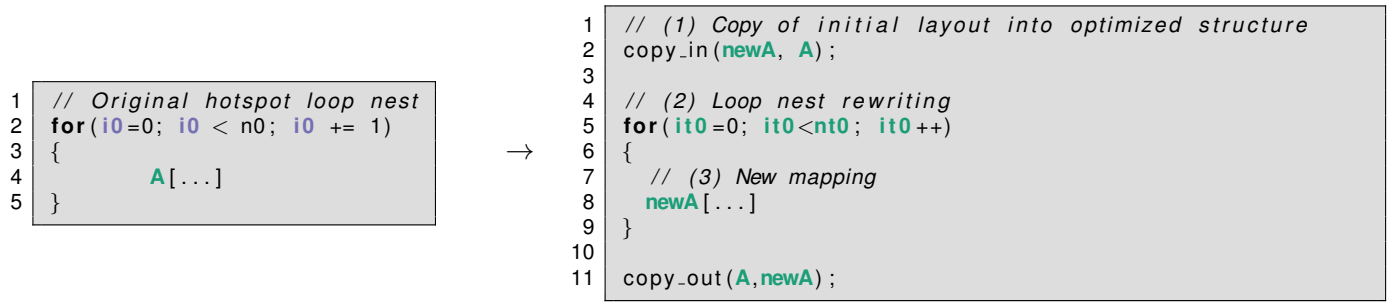


Figure 5.1: The three main steps of loop nest rewriting — copy of the initial layout, new loop nest itself, individual memory instruction new mapping

rearranged, accordingly to the respective restructuring strategies contemplated, that may induce nest-breaking transformations such as loop permutations or loop fusions for instance. Finally, each of the individual memory accesses instructions have to be modified in order to take account of the new memory mapping.

5.1.2 Formalism Interpretation

According to Chapter 3, we can write a generic n-dimensional layout as:

$$\begin{Bmatrix} A_{d_{n-1}} \\ S_{I_{n-1},d_{n-1}} \end{Bmatrix} \otimes \cdots \otimes \begin{Bmatrix} A_{d_0} \\ S_{I_0,d_0} \end{Bmatrix}$$

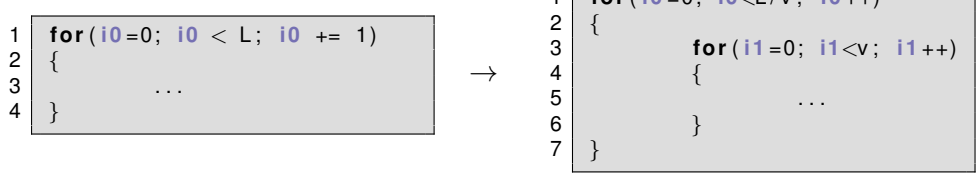
Even though as formally defined $S_d = A_d$, there is a difference between the two notations that is important to mention for code generation purposes. As given by delinearization described in chapter 3, A functions actually represent memory chunks accessed wholly by any assembly instruction probed for memory traces – much like the array represented by an A in the traditional notation AoS – , while S functions usually represent small memory chunks where only one element is accessed per instruction, much like structure fields in C programming language. This distinction allows us to make deductions from the formal description of the layout. The A elements are consecutive and need to be processed by loop structures, thus each dimension written as an A represents a loop level. Said otherwise, there are as many loops as As needed to process the entire layout.

Example – Lattice QCD:

Consider the following layout transformation:

$$A_L \otimes S_{\{2\},d} \otimes S_m \otimes S_c \xrightarrow{AoSoA-dbl} A_{L/v} \otimes S_{m\times c} \otimes A_v$$

We can deduce the corresponding loop nests:



Here, the initial layout only has one A which implies one loop in the code. However, the second layout as an additional A, consequence of an AoS to AoSoA transformation. In terms of code, it means another loop need to be written.

Formalism also hints the loop bounds. Loop bounds are — outermost to innermost — the $\{d_{p-1}, \ldots, d_1\}$ as given in Equation5.1.2, where p is the number of arrays A, and the set elements their respective diameters. Indeed, there is no loop on a given structure S, since only one element is accessed by the pattern, rather it becomes part of the strides. The loop increment on dimension j is given by:

$$s_j = \prod_{k=0}^{j} d_k$$

which is the linearized stride on the dimension $j \in [0, p-1]$

Example – Lattice QCD:

Consider the following layout transformation:

$$A_L \otimes S_{\{2\},d} \otimes S_m \otimes S_c \xrightarrow{AoSoA-dbl} A_{L/v} \otimes S_{m \times c} \otimes A_v$$

Therefore the memory accesses have the form given in Figure 5.2.

```
1  for(iL=0 ; iL < L ; iL+=1) {
2    for (j=0;j<n2;j++) {
3      ...
4      r0 += U[8*iL+2][1]*tmp[n2+j];
5  // real part eq. to
6  //   U[8*iL+2][1][0]
7      ...
8  }}
```

```
1  // double newU;
2  for(i0=0 ; i0 < L/v ; i0++){
3    for (i1=0;i1<v;i1++)  {
4      ...
5      r0 += newU[i0*(m*c*v)
6                 + 2*(v)
7                 + i1]
8            *tmp[n2+j];
9      ...
10   }
11 }
```

Figure 5.2: Memory access example on Lattice QCD layout transformation

Here given in C, 3-dimensional $newU$ has therefore 2 non-unit inter-dimensional strides $(m \times c \times v, v)$ that are explicit through the formalism.

Now that the intuition is given, the problem is the following: Loop nest as given by the formalism is different than the one in the initial code. There are several reasons behind this matter.

First of all, loop nest in the trace is different than corresponding loop nest in the code. Compilers may transform the loop nest as initially expressed in the source code, a lot of optimizations (loop fusion, strip mining, useless loops removal, etc.) themselves provoke important modifications impacting the binary, and thus the memory traces. There are also indirections that may conceal some information to the compiler, such as a regular strided multi-dimensional access that can be rightfully interpreted as loops by the NLR algorithm that compress that memory traces. Another point is that delinearization may result in different layout structure than the one expressed in the source code. Indeed, it is not meant to match the source code representation, rather it permits to obtain a normal form. Moreover, the formalism does not comprise all the loops in the initial code fragment considered, just the ones the layout of interest depends on, as they are reported in the memory traces.

Second, the formalism we obtain after a lot of different reshaping strategies implies different intuitive loop nest than the one in the trace. An example is the AoS to AoSoA transformation where another loop is injected in the original loop nest as given by the memory traces.

```
1 for(iL=0 ; iL < L ; iL+=1) {
2   for (j=0;j<n2;j++) {
3     ...
4     r0 += U[8*iL+2][1]*tmp[n2+j];
5     ...
6 }}
```

```
1 for i0 = 0 to 131071
2   val 0x100910 + 1152*i0
3 endfor
```

Figure 5.3: Loop nest as it appears in memory traces is different than the loop nest in the source code – Here loop on j has been unrolled by the compiler, and U does not depend on it thus it does not appear in the trace

We explain in this chapter how the code layout transformations are smoothly integrated in the original binary loop nest for code mock-up generation, and how loop nest code restructuring can be hinted to the user. Multiple inner loops may have different expressions in the formalism. We will first assume they have the same expression, that is there is one layout transformation to apply per layout.

5.1.3 Copying

As discussed before, because our scope is local it is likely that the layout of interest allocation is unreachable, meaning that performing a copy to pass from the original suboptimal layout to the new layout is inevitable. The programmer however, as he has a global view on his source code, can decide to apply the copy earlier in the code, or even skip the copy by directly allocating the layout according to the transformations suggestions if applicable. Still, one way to perform the copy is to copy each trace one by one. However, this would result in a lot of instructions, possibly costly patterns and certainly poor performance. Copying time can be critical if the copy is close to the innermost kernel of interest, therefore it is important to have an efficient and fast copy code, and formalism can help us doing so. Moreover, both arrays, the old one and the new one, cannot be accessed in a contiguous manner because otherwise that would mean their topology is identical, which would be useless in our case. The key here is to guarantee that one of them is accessed contiguously for performance

Here again, the notion of iterator is leading. Indeed, we need to be able to perform the mapping between the old and the new layout, and the iterators, as given by the memory traces, guarantee the order of access to the old layout. The splitting transformation in particular, unlike permutation and compression, induce the creation of new iterators out of the original ones, and it is important to be able to track back to the initial iterator to ensure correct mapping between the old structure and the new structure.

Let $\overrightarrow{\times}$ be the non-commutative multiplication, and the associated division operators $\overrightarrow{/}$ and $\overleftarrow{/}$ are defined by:

$$c = a \overrightarrow{\times} b \iff a = c \overrightarrow{/} b \iff b = c \overleftarrow{/} a \tag{5.1}$$

We add the notion of iterators to the transformations rewriting system described in 4.1.

$$\begin{aligned}
(L, i_1) \otimes (L', i_0) &\to (L', i_0) \otimes (L, i_1) \\
(A_{nm}, i'_1 \overrightarrow{\times} i'_0) &\to (A_n, i'_1) \otimes (A_m, i'_0) \\
(S_{I,n}, i_1) \otimes (S_{J,m}, i_0) &\to (S_{I',nm}, i_1 \overrightarrow{\times} i_0),\ if \#I' = \#I \times \#J \\
(S_{I,d}, i) &\to (S_{I',d'}, i),\ if \#I = \#I', d \leq d'
\end{aligned} \tag{5.2}$$

Because the new layout is gapless by default, we propose to perform the copy in a manner that allows contiguous access on this layout. The general expression of the copy can be written such as:

$$new[i'_n]\dots[i'_1] = old[e(i_m)]\dots[e(i_1)]$$

Where the i' iterators correspond to the copy loop nest iterators and where the $e(i)$s are expressions, that depend on at least one of the i's, and that can be infered from the results of the transformations described by 5.2, in order to have only expressions of i's in the copy code. We express e as a rewriting system:

$$\begin{aligned} i_1 = i'_0 \overrightarrow{/} i_0 &\ \rightarrow\ i'_0 \ / \ \#i_0 \\ i_0 = i'_0 \overleftarrow{/} i_1 &\ \rightarrow\ i'_0 \ \% \ \#i_0 \\ i_Y = i'_1 \overrightarrow{\times} i'_0 &\ \rightarrow\ i'_1 \times \#i'_0 + i'_0 \end{aligned} \tag{5.3}$$

If any iterator is not an interval, that is a structure with unused fields, then the copy instruction model is duplicated and we use all the combinations of the scalars instead.

Example: Consider the following transformation:

$$\begin{aligned} & (A_X, i_2) \otimes (S_W, i_1) \otimes (A_{YZ}, i_0) \\ \xrightarrow{split \mid i_0 = i'_1 \overrightarrow{\times} i'_0} & (A_X, i_2) \otimes (S_W, i_1) \otimes (A_Y, i'_1) \otimes (A_Z, i'_0) \\ \xrightarrow{permutation} & (A_X, i_2) \otimes (A_Y, i'_1) \otimes (S_W, i_1) \otimes (A_Z, i'_0) \\ \xrightarrow{merger \mid i'_2 = i_2 \overrightarrow{\times} i'_1} & (A_{XY}, i'_2) \otimes (S_W, i_1) \otimes (A_Z, i'_0) \end{aligned} \tag{5.4}$$

Therefore copy code is on the form:

$$new[i'_2][i_1][i'_0] = old[e(i_2)][i_1][e(i_0)]$$

Let us now express each of the $e(i)$s in terms of i's:

$$\begin{aligned} i'_2 = i_2 \overrightarrow{\times} i'_1 &\overset{5.1}{\iff} i_2 = i'_2 \overrightarrow{/} i'_1 \\ &\overset{5.3}{\iff} e(i_2) = i'_2 / \#i'_1 \\ &\iff \boxed{e(i_2) = i'_2 / Y} \\ i'_1 = i_0 \overrightarrow{/} i'_0 &\overset{5.1}{\iff} i_0 = i'_1 \overrightarrow{\times} i'_0 \\ &\overset{5.1}{\iff} i_0 = (i'_1 = i'_2 \overleftarrow{/} i_2) \overrightarrow{\times} i'_0 \\ &\overset{5.3}{\iff} e(i_0) = (i'_2 \% \#i'_1) \times \#i'_0 + i'_0 \\ &\iff \boxed{e(i_0) = (i'_2 \% Y) \times Z + i'_0} \end{aligned}$$

Let $i_1 = \{0, 1, 3\}$.

Finally, the copy code is the following:

```
1 for (ip2=0; ip2<XY; ip2++)  {
2     for (ip0; ip0<Z; ip0++) {
3       new[ip2][0][ip0] = old[ip2/Y][0][(ip2%Y)*Z + ip0];
4       new[ip2][1][ip0] = old[ip2/Y][1][(ip2%Y)*Z + ip0];
5       new[ip2][3][ip0] = old[ip2/Y][3][(ip2%Y)*Z + ip0];
6 }}
```

Note that such code snippet can be reported to the user as it is, and can be used as it is as the copy is always correct. Only the aliases $\{old, new\}$ must be modified as they are given here arbitrarily. This same code is also used in the mock-up code generation.

5.1.4 Remapping

Each memory access in the original loop nest need to be modified as a new mapping is given after data restructuring. A solution would be to rewrite the entire loop nest in order to access the new layout efficiently, however such deep code modifications would possibly require more static analysis to ensure the mock-up code is close enough to a user rewritten version . We propose to access the new layout elements using a NEXT function to theoretically calculate every iteration the new position to reach, however in practice such function would be in most cases removed by the compiler via the use of induction variables.

We want to express the new layout iterators i's in terms of the original layout iterators is and generate the expressions needed to properly calculate the new accesses

$$new[e(i'_n)] \dots [e(i'_1)]$$

This is done again thanks to 5.2. However, the i iterators are given by the memory traces, and thus different from the source code iterators. We need to inject them into the original loop lest, in order to correctly access the new layout. So, in order not to make the addresses calculation rely on costly / and $\%$, we make a first pass to remove these operators by adding systematically a new iterator:

$$(i'_k/X, i'_k\%X) \rightarrow (i'_k, i''_k) \;\Big|\; \#i''_k = X \tag{5.5}$$

Then, for each of the multi-level structure fields combination, we write a function given in Figure 5.4 to compute the addresses to be accessed by any memory instruction on the layout of interest, independently of the original control, that we will insert prior to any of said memory instructions.

Example

Based on 5.4, all the memory instructions in the transformed loop nest have the form:

$$new[e(i'_2)][i1][e(i'_0)]$$

```
1  NEXT(r0)
2  {
3      if (j_n     > #i_n)     return; // unreachable
4      if (j_{n-1} > #i_{n-1}) j_{n-1} = 0; j_n++;
5      ...
6      if (j_1     > #i_1)     j_1 = 0; j_2++;
7      r0 = base + e(i_n) * (#i_{n-1} * ... * #i_1)
8                + e(i_{n-1}) * (#i_{n-2} * ... * #i_1)
9                ...
10               + e(i_1)
11     j_1++;
12 }
```

Figure 5.4: NEXT function

We need to express the $e(i')$s in terms of is, since the loop iterators are is here:

$$
\begin{aligned}
i'_2 = i_2 \overrightarrow{\times} i'_1 &\overset{5.1}{\iff} e(i'_2) = i_2 \times \#i'_1 + i'_1 \\
&\iff e(i'_2) = i_2 \times Y + (i'_1 = i_0 \overrightarrow{/} i'_0) \\
&\overset{5.1}{\iff} e(i'_2) = i_2 \times Y + i_0/\#i_0 \\
&\iff \boxed{e(i'_2) = i_2 \times Y + i_0/Z} \\
i'_0 = i_0 \overleftarrow{/} i'_1 &\iff e(i'_0) = i_0 \% \#i'_0 \\
&\iff \boxed{e(i'_0) = i_0 \% Z}
\end{aligned}
$$

Therefore all the new memory accesses are on the form:

$$new[i_2 \times Y + i_0/Z][i_1][i_0 \% Z]$$

Then, after division-removing pass (5.5) it becomes:

$$new[i_2 \times Y + i_0][i_1][j_0]$$

We deduce the following $NEXT$ function that calculates individual memory accesses.

```
1  NEXT(r0, i_1)
2  {
3      if (i_0 > Y) i_0 = 0; i_2++;
4      if (j_0 < Z) j_0 = 0; i_0++;
5      r0 = base + i_2 * Y + i_0 * (Y * Z)
6                + (i_1) * Z
7                + j_0
8      j_0++;
9  }
```

Where i_1 is a structure field, and depends on the memory instruction base address itself.

Then, the compiler can use induction variables to remove unnecessary ifs and duplications, and generate efficient code.

5.2 User Feedback

One of our main contributions is the feedback we can provide, as to which layout transformations to apply and how to proceed. Now that we explained how the layout remapping is done, we

can report to the user. Here we detail how automatic feedback is generated, thanks to both the formalism and the memory traces.

This section exclusively deals with the remapping feedback problem, as the copy feedback subject as been treated previously and the quantified feedback is the next chapters topic.

5.2.1 Layout issues pinpointing

The first feedback we must provide the user with is the actual layout issues happening in his application. Gaps and SIMD-inhibiting structures can be reported to the user in a very concise and convenient way. Because the formalism itself is naturally not to be used raw for feedback, we propose to resort to the NumPy [1] notation to immediately translate the formalism to a more user-friendly version.

Example – Lattice QCD benchmark:

Consider the following layout:

$$A_{131072} \otimes S_{\{2\},8} \otimes S_9 \otimes S_2$$

Let us use a C declaration to express the layout structure:

$$a[131072][8][9][2]$$

Now mixed with NumPy semantic to give a hint of access patterns:

$$a[0:131072,'2:3','0:9','0:2']$$

Where $2:3$ actually means an access to the element 2 only, while 3 represents the unreached upper bound. Here we used the '' notation reserved to data fields access to hint the presence of a structure on the dimension where it appears. This notation is originally meant to refer to a single data field name, but since we do not have any, we assume the names would be the fields position, potentially several – if not all – of them. Finally, we have translated all potential the layout issues held by the formalism representation into more conventional notations for better user understanding.

Furthermore, we can also report issues up to the individual memory instruction level. This is done by translating the memory traces into a C representation of the formalism notation, as shown in Figure 5.5. Here the user can notice that the innermost loop accesses the outermost dimension of the layout, which is bad for performance and SIMDization.

5.2.2 Hinting the rewriting

Having reported the layout issues to the programmer, now we want to propose possible solutions, as to which layout may improve the user application performance.

Example – Lattice QCD benchmark:

Consider the following transformation:

$$A_L \otimes S_{\{2\},d} \otimes S_m \otimes S_c \xrightarrow{AoSoA-dbl} A_{L/v} \otimes S_{m \times c} \otimes A_v$$

Table 5.2.2 reports how the formalism is translated to the user, using C/NumPy notations, where v is a given SIMD vector size. Description of both initial and transformed layout is shown,

```
1  for i1 = 0 to 131071
2    val 0x7f845baa2168 + 1152*
       i1
3  endfor
4  for i1 = 0 to 131071
5    val 0x7f845baa2170 + 1152*
       i1
6  endfor
7  for i1 = 0 to 131071
8    val 0x7f845baa2178 + 1152*
       i1
9  endfor
10 ...
```

```
1  //double A[131072][8][9][2];
2  for(i0=0 ; i0<131072; i0++)
3  {
4      A[i0][2][0][0]
5      A[i0][2][0][1]
6      A[i0][2][1][0]
7      A[i0][2][1][1]
8      A[i0][2][2][0]
9      ...
10     A[i0][2][8][1]
11 }
```

Figure 5.5: Layout issues pinpointed to individual instruction level

Representation	Initial Layout	AoSoA-dbl Transformation
Formalism	$A_{131072} \otimes S_{\{2\},8} \otimes S_9 \otimes S_2$	$A_{131072/v} \otimes S_{9\times 2} \otimes A_v$
C declaration	$A[131072][8][9][2]$	$A[131072/v][9*2][v]$
NumPy	$A[:, '2:3', ':', ':']$	$A[:, ':', :]$

Table 5.1: View of given layout transformation through C/NumPy

aiming at highlighting the global idea behind the proposed transformation, so the user gets the big picture of how his code should be transformed to comply with the given transformation.

It is possible to go further, up to the individual instruction level to suggest corrections in order to rewrite the memory accesses accordingly to the hinted transformation. Thanks to the formalism and the knowledge of the instructions base addresses thanks to the memory traces, it is possible to give a C representation of what the new accesses should resemble for a given layout transformation, in order to give a sense of how the access should be performed. Figure 5.6 shows how the initial accesses can be rewritten to map onto the new structure. The difficulty here lies in the fact that it might not be straightforward to pass from the initial loop nest to the new one, the loop design is still the user responsibility.

5.3 Low-level implementation

We have detailed an approach to redesign memory mapping, in this section we explain how the modifications are applied at lower level in order to actually rewrite the user application binary code.

5.3.1 Loop kernel rewriting

The first step in rewriting the user application binary loop kernel of interest is to retrieve said kernel. We use MAQAO to disassemble the binary and to perform basic loop analysis as to retrieve the control flow graph and the dependence graph. This way we obtain the kernel assembly listing as well as its address in the binary, so it is possible to patch a new modified kernel instead. The patch consist in editing the binary kernel, the idea is to replace the original kernel by NOP instructions – instructions with no effect other than padding the binary – so as not to compromise all hardcoded binary offsets in the code, and to add jump instructions to and from a safe area

```
1  //double A[131072][8][9][2];
2  for(i0=0 ; i0<131072; i0++)
3  {
4      A[i0][2][0][0]
5      A[i0][2][0][1]
6      A[i0][2][1][0]
7      A[i0][2][1][1]
8      A[i0][2][2][0]
9      ...
10     A[i0][2][8][1]
11  }
```

```
1  //double newU[131072/v][18][v
       ];
2  for(i0=0 ; i0 < 131072/v ; i0
       ++)
3  {
4    for(i1=0; i1<v ; i1++)
5    {
6      newU[i0][0][i1]
7      newU[i0][1][i1]
8      newU[i0][2][i1]
9      newU[i0][3][i1]
10     newU[i0][4][i1]
11     ...
12     newU[i0][17][i1]
13   }
14 }
```

Figure 5.6: Loop nest rewriting

at the end of the binary file where the new kernel code is added. We choose to encapsulate this new kernel in a function body, which as a few consequences. For instance, we ought to be careful with stack usage, it is perturbed by the insertion of a new function, in place of the old hot spot. Therefore instructions using offsets relative to the stack pointer become wrong, a simple fix is to retrieve the data pointed prior to the function call itself and store it into newly allocated structures. Likewise, instructions using IP register need to be treated carefully, and we may want to suggest the use of other registers instead.

In order not to manage the registers ourselves, we choose to resort to compiler register reallocation. This means the assembly code we rewrite is written as C inline assembly, and then recompiled before being patched onto a new binary, as a function call.

Every memory instruction that accesses the initial sub-optimized layout has to be changed, so the data is accessed exclusively via the newly generated layout. In the case where the new structure is composed of more dimensions than the initial layout, code generation may or may not use additional integer registers, the decision is made by compiler register allocation. Furthermore, the new layout we allocate is well-aligned which makes possible the use of aligned instructions. Therefore as for alignment, all the initial hardcoded offsets are removed from the memory instructions. Other instructions such as arithmetic operations are left untouched, in a non-SIMD context.

As we assume fixed stride instructions, indirections do not happen anymore. Indeed, in some applications code, patterns mapped by index table are in fact regular patterns, but suffer of a deficit of performance due to indirections. The difficulty here is transparent in this specific case, as the new layout does not require an index table, and the indirection leftovers in the code are evicted through dead code elimination.

5.3.2 SIMDization

The vectorization analysis as described in Chapter 2 allows us to determine if vectorization is actually applicable to a given innermost loop kernel. If it is, then a lot of code modifications is needed, to feature new access patterns alongside the use of SIMD instructions and other SIMD-related adjustments.

We have to substitute the original hot loop with the new one, which is shorter by a factor equal to the architecture vector size. The initial number of iterations is retrieved from the memory traces,

therefore the mock-up loop is bound to the run captured. Increment on the new layout is modified to be vector-sized, and if pointers are updated within the loop before the first use as revealed by static dependence analysis – that is what we name pre-incrementation – the original instruction has to be placed right before the loop to set the pointer correctly.

```
1
2 loop:   ...
3         add       $0x4, %rax
4         movsd     (%rax),%xmm7
5         ...
6         jmp loop
```

```
1         add       $0x4, %rax
2 loop:   ...
3         movapd    (%rax),%xmm7
4         add       $0x10, %rax
5         ...
6         jmp loop
```

Figure 5.7: Pre-incrementation fix – assuming vector size is 0x10 bytes

As shown in Figure 5.7, since the new iterator has a wider stride, pre-incrementation becomes incorrect as the base address is shifted. Most of the instructions mnemonics need to be substituted for their SIMD equivalent. We use packed instructions, and exclusively scalar instructions such as movsd with three parameters, has to be replaced by their equivalent SIMD instructions.

```
1         movsd   0x40(%r10,%r14,1),%xmm7
```

```
1         movapd  (%r10),%xmm7
```

Figure 5.8: Non-SIMD instructions substitution

Figure 5.8 shows an example of instruction that need to be replaced by an equivalent SIMD instruction. Particular instruction shown does not have an equivalent but still can be effectively substituted, as we use exactly one register for the new layout. Then it is up to the register reallocation to reaffect registers if needed. One of difficulty of binary code rewriting is that compiler optimizations may complexify the loop instructions. However, some compiler optimizations can be untangled, that is the case for partial loads that are replaced by a single packed load operation.

```
1         movsd   (%r10),%xmm6
2         movhpd  0x8(%r10),%xmm6
```

```
1         movapd  (%r10),%xmm6
```

Figure 5.9: Partial load eviction

Figure 5.9 shows an example of partial load that can be merged into one instruction, which can be advantageous if the two addresses are far apart.

Reductions are spotted through SIMD analysis, thus we are able to tackle them using horizontal operations. We can also spot read-only arrays or constants and unpack them.

5.4 Conclusion

We presented in this chapter how based on data layout formalism we are able to rewrite the user application binary, and notify the user with comprehensive feedback on how to apply the code modifications needed to implement suggested transformations. Formalism allows us to limit the copy overhead, in order to pass from the initial suboptimal layout to a new given transformed layout, and to remap each of the loop kernel accesses individually. We also gave details of implementation at low-level, where binary and assembly tricks are useful to process the input user

binary and produce an efficient output. Assessment of transformations and transformed binary at the same time is presented in the next chapter.

Chapter 6

Transformations Evaluation

6.1 Evaluation methodology 91
6.1.1 Principle of in-vivo evaluation 91
6.1.2 Automatic mock-up generation and vectorization 92
6.1.3 Current state of implementation 93
6.2 Experimental results 94
6.2.1 TSVC 94
6.2.2 Lattice QCD benchmark without preconditioning 95
6.2.3 Lattice QCD benchmark with even/odd preconditioning 96
6.2.4 Lattice QCD application without preconditioning 97
6.2.5 2D cardiac wave propagation simulation application 98
6.3 Conclusion 98

To assess the performance of proposed layout transformations, we introduce a novel technique that is the in-vivo evaluation. The idea is to directly measure transformed kernel performance in the context of the user application, so the prediction can be reliable as the numbers correspond to real use case. We show how we accelerate such potentially costly evaluation with a methodology based on checkpoint/restart, that only replays the kernels of interest within the user application. Then we present the final results of our framework on sets of benchmarks, as well as on two real-life multithreaded applications, namely a lattice QCD simulation and a cardiac wave propagation simulation. We show how performance prediction are reliable for the user to make data layout restructuring decision opportune in order to reach a significant performance gain.

6.1 Evaluation methodology

6.1.1 Principle of in-vivo evaluation

High accuracy of predictions can only be reached if the mock-ups behave closely like the user-restructured version. To do so, we perform dynamic evaluation in the context of the application, which we call in-vivo evaluation, thus tending to preserve application execution conditions.

Evaluation of our different restructuring strategies resorts to Checkpoint/Restart technique: The original binary is patched with a checkpoint function call, then run until the checkpoint; right before the hot function call, for this matter. This produces a context file, which serves as a base context for instrumentation and for all mock-up evaluations. The binary code is instrumented in order to collect the memory trace and restarted from this context. Several layout transformations are applied on the initial code, generate as many versions and for each of these versions, the application is again restarted from the same context. Note that contrarily to the common use of checkpoint/restart, here the code checkpointed is different from the codes restarted (with modified layout or instrumentation added). The context is also used for a reference timing, using a binary containing the initial kernel of interest, in order to deduce speedup from mock-up timing measurements. Context file also ensures the memory addresses collected by the instrumentation are always valid through re-runs. Indeed, restructuring strategies implementations rely on memory areas knowledge, as we need to retrieve data from original layout to be copied in our optimized layout. The one parameter our approach does not preserve is cache state. Cache warm-up may be a solution, but goes beyond the scope of this document. Re-runs are stopped at the function of interest exit, and function timing is deduced at this point. For checkpoint/restart, we resort to the BLCR library [40].

Restarting a context file with other binaries than the original one is not a proper use case handled by the checkpoint/restart software, we need to adapt. Being a snapshot of the original process, the context file features information such as memory data, binary instructions and library information, among others. When the restart utility is run, it requires the initial binary as well as the libraries used to be exactly the same as they were at the checkpoint time. Now, as said before, code mock-ups are different binary versions that need to be restarted using the context of a real user application run. Therefore the original code section that is subject to modification has to be concealed, and one way to do so is to encapsulate the section in a new library, dynamically loaded after the restart point, in order to be absent from the context file and the restart to be safely executed, even though the initial library is replaced by mock-up libraries, which incorporates many modifications. The aforementioned section is simply the function of interest, and is called right after the restart point, as the initial application was stopped right before its call. This approach can be generalized to evaluate multiple instances, *id est* same callee but different callers and parameter values, and to evaluate multiple functions.

It is important to note that instrumentation is performed only for given set of functions, and never on the whole application as it is overly costly, in terms of both instrumentation time and memory usage. The estimation of the global impact of layout transformations can be obtained by adding the impacts on the different points of interest. One of the consequences of restarting at the functions of interest, and stopping at their respective exits, is the minimization of the whole set of evaluation elapsed time. Indeed, even though the hot kernels represent a large proportion of the overall application time, the rest of the application execution is not negligible, especially when it is re-run multiple times.

6.1.2 Automatic mock-up generation and vectorization

Not all loop kernels are intrinsically SIMDizable, thus we first have to determine if loop kernels of interest are actually SIMDizable before effectively attempting SIMD optimizations. We rely on MAQAO, which is able to determine loop kernel SIMDizability [6], checking dependence in particular. SIMDizability of loops is not an absolute necessity in the context of mock-up generation,

as performance improvement is still achievable through locality optimization. Mock-up libraries are copies of the reference hotspot library, with deeply modified critical area. Modification has two aspects, relative to layout and SIMDization.

Copying of initial library skeleton requires a few binary operations. First, the old kernel must be substituted, and second, a function call to the mock-up kernel is added instead. Also two other function insertions have to be performed to the new binary, corresponding to copy-in and copy-out to/from the new layout.

Performing copy-in is essentially retrieving data from the old array, rearranged in a newly allocated array in order to improve performance through contiguous accesses. The mapping depends on the restructuring strategy considered and shall be reused in the copy-out function. Data retrieval is possible since memory trace analysis provides the array base address, or at least the array lowest address accessed during the function execution, which might be different since memory traces are profile-dependent. New array size is defined by the memory traces analysis, consequently only the data accessed during a given run are collected here, it may not be the actual size of the initial array and may lack elements, although they are not used in this very computation.

For the sake of our evaluation, we need to find suitable spots to perform copy-in and copy-out, proposed to the user. The difficulty is to optimize hot loop performance, *id est* pushing away the copies from the kernel to minimize their impact, while avoiding cache pollution potentially induced by inopportune insertion of copy functions. Optimizing copies placement for the whole function is difficult, rather we focus on kernel optimization on a smaller scope. The idea is to move the copy up to the beginning of the function if applicable, the further away from the innermost kernel, the best it is for its performance.

6.1.3 Current state of implementation

The framework we discussed is currently a prototype, written in LUA. The input user application can originally be written in C, C++, Fortran or any language as long as there is a binary version for the framework to analyse. The framework is currently not fully automatic, there are still some tasks the user has to perform by himself to use it properly. The hotspot function detection has to be done by the user which is feasible with any profiling tool. Also, once said function is spotted it has to be written in an external dynamic library, so it can be swapped with mock-up libraries – the libraries that contain each one different mock-up function – and re-ran without disturbing the checkpoint/restart software functioning.

From then on, all the framework steps are automatic, except for the transformations selection that has to be user specified: the transformation space exploration is currently not implemented. Nevertheless, our framework scripts process automatically the user binary to add the actual checkpoint function call which is added before the hot function call, as well as the `dlopen` system call that permits to open the dynamic mock-up libraries right after the restarts. Then the instrumentation, trace analysis, transformations implementations and the evaluation are automatic. Instrumentation is performed by MAQAO in our implementation, which is also the case for the disassembly step to retrieve the original loop kernel assembly code.

As for vectorization, our method to determine kernel intrinsic vectorizability is implemented in MAQAO, which tells us when to perform vectorization on our restructured kernels. Our kernel vectorization is a naive implementation and uses SSE2 instructions. It basically consists in original instruction substitution by SIMD instructions, and the few other operations as described in Chapter 5.

Finally, the user feedback itself is not currently implemented.

6.2 Experimental results

The objective of the section is to show how relevant the speedup hints are, in the sense that they provide useful advice to the programmer, as to which layout transformation best suits his needs. To do so, we compare our framework speedup predictions with actual user C-level code restructuring performance, following the transformation hints automatically proposed. All the code mock-ups produced and analyzed in this section implement transformations explained in Section 4.3. Each of these are vectorized by default, and are accompanied by another mock-up which is intended to serve as a referent non-SIMD version, in order to observe the sole effect of layout restructuring on performance, for the sake of this discussion.

We used an Intel(R) Xeon(R) CPU E5-2650, 2.00GHz 2*8-core processor, with its SSE2 features, which implies 2-elements vectors as all codes studied use double precision numbers. Also, rewritten C codes following our framework guidelines are compiled with icc 15.0.0 and gcc 5.3.1, both with O3 flag.

6.2.1 TSVC

We assess the accuracy of our mockup-driven predictions on a suite of benchmarks, TSVC [72]. TSVC consists of 151 functions intended to explore the typical difficulties a compiler can meet in the context of vectorization. Out of these 151 benchmarks, 31 matches data layout access issues, our primary focus in this study. The others correspond to control issues, mostly already well handled by compilers. In Figure 6.1, we report speedups obtained by *mockup* kernels and by manually restructured (*correct*) kernels over compiled basis kernel compiled with icc 13.0.1, on Intel Sandy Bridge E5-2650 @2GHz. These kernels are a subset of the data layout issue category.

Non-contiguous stride accesses are an obstacle to vectorization, compilers may see the opportunity of vectorization but consider it not efficient enough to vectorize. This is the case for benchmarks *s111* and *s128*, performing accesses with strides 2. Their respective mockups show significant gain to expect from data restructuring. When 2-dimensional arrays are accessed column-wise (in C) or row-wise (in Fortran), accesses with large strides are performed, and one may resort to data transposition prior to massive computations, in order to allow vectorization. Code mockup here predicts significant gains to expect from restructured kernel, which is effectively perfectly reached. One big challenge for compiler autovectorization is brought by rescheduling issues, that is, codes where compilers see vector dependences it can not resolve, although such dependences could be fixed by permuting instructions or loop peeling. All *s241*, *s243*, *s211*, *s212*, *s1213*, *s244* and *s1244* benchmarks have rescheduling issues and are not vectorized by the compiler. Here, the dynamic dependence graph enables to find a correct schedule for SIMD code. In some cases, a non-contiguous data pattern may not cause performance issues, as they are already well handled by the compiler and/or the architecture. Here, benchmarks showing no speedup over the basis kernel (*s1111*, *s131*, *s121*, *s151*) correspond to alignment issues, which can be solved by unaligned accesses or vector permutations. On this very architecture, unaligned accesses do not produce a significant performance overhead, therefore data restructuring will not bring better performance.

For all these measurements, the time to restructure data (using a copy) is not included. Indeed, the benchmarks are small functions and a copy is, with a few exception, not amortized.

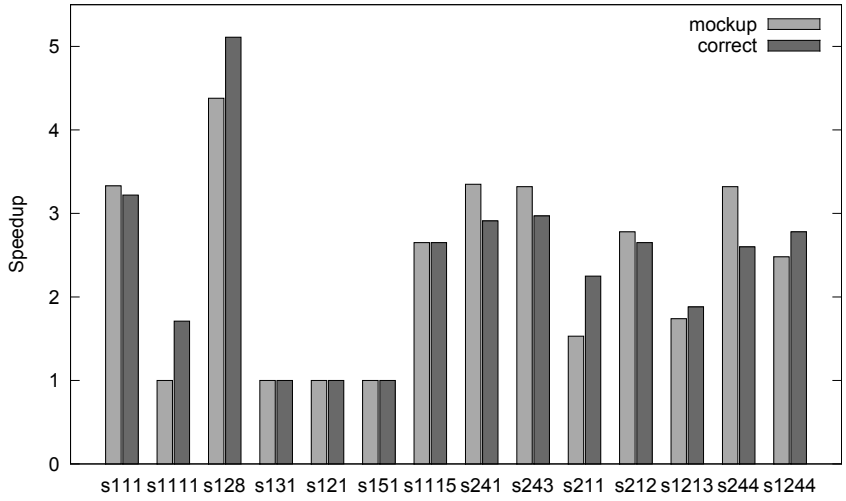


Figure 6.1: TSVC Mockup Prediction on x86

Code	Initial Layout	Transformed Layouts [short name]
Qiral excerpt	$A_L \otimes S_{\{1\},d} \otimes S_m \otimes S_c$	$A_{L/v} \otimes S_{m\times c} \otimes A_v$ [AoSoA-dbl]
Qiral precond. excerpt	$\bigoplus_{x,y,z,t\equiv 0[2]} \left(\bigotimes_{k=x,y,z,t} A_l \otimes S_{k,2} \right) \otimes S_{\{1\},d} \otimes S_m \otimes S_c$	$A_{L/v} \otimes S_m \otimes A_v \otimes S_c$ [AoSoA-cplx] $S_{m\times c} \otimes A_L$ [SoA-dbl] $S_m \otimes A_L \otimes S_c$ [SoA-cplx]
Qiral application	$A_L \otimes S_d \otimes S_m \otimes S_c$	$A_{L/v} \otimes S_d \otimes S_{m\times c} \otimes A_v$ [AoSoA] $S_d \otimes S_m \otimes A_L$ [SoA]
Cardiac Wave	$A_X \otimes A_X \otimes S_{\{0\},a} \otimes S_{\{0\},s}$	$A_X \otimes A_X$

Table 6.1: Studied Transformations, obtained as described in Section 4.3

Performance of the mockup is in most cases close to the real transformed code. When there are differences, this is explained by the fact that the mockup results from a binary transformation, while the correct hand-tuned code results from a source-to-source transformation. Hence, the binary code resulting from hand-tuning the source code may not be exactly the same as the mockup code due to compiler optimizations.

6.2.2 Lattice QCD benchmark without preconditioning

The copy-in can be inserted anywhere prior to the computations, as we have already established in Section 4.3. As for the kernel itself, initial layout and transformations applied are given in Table 6.1. The reference version operating on the initial layout is not automatically vectorized by neither icc nor gcc. Hand-rewritten versions are not autovectorized neither, although complex multiply optimization is applied by icc. *cplx* versions use C complex type, as originally written in the Qiral application, while *dbl* versions bypass complex multiply optimization with macros so the entire kernel is viewed as using only double precision elements by the compiler.

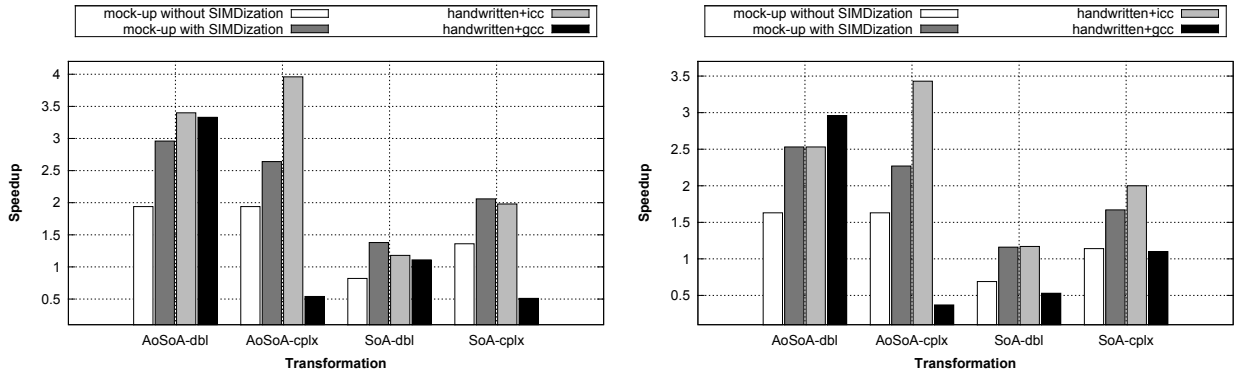


Figure 6.2: Lattice QCD Benchmark without Preconditioning (left), with Even/Odd Preconditioning (right) Speedup, single thread.

Figure 6.2 shows performance prediction results obtained for each transformation, along with the comparison with the hand-optimized version. While code mock-ups are transformed binary versions, the hand-optimized version remains in C, potentially taking advantage of further compiler optimizations (icc and gcc). The whole set of mock-ups performance constitutes a part of user feedback we propose.

All mock-ups predict performance improvement for each of the four transformation presented, which is backed by the user rewritten version, with an average relative error of 16%. Globally, mock-ups approach the actual rewritten kernel performance, except for *AoSoA-cplx* where the performance prediction is somewhat pessimistic, although reasonably reliable. The gap between prediction and reality on this specific case is in fact due to icc complex multiply optimization, which outperforms indisputably the mock-up too naive SIMDization. The mock-up basically uses partial loads to vectorize over the complex type, while icc uses vector operations to perform fast complex multiply. On the other hand, gcc performance collapse when it comes to complex computations, compilation did not entail any SIMDization or complex multiply optimization. Moreover, loop parallelization proposed by mock-ups takes over when it comes to SoA transformation, as it slightly outperforms rewritten versions, and we can notice icc compiler work on complex number on *SoA-cplx* version pushes it towards mock-up prediction, with respect to the *SoA-dbl* version, while general SoA loss of spacial locality lead to lesser performance here on traditional CPU.

6.2.3 Lattice QCD benchmark with even/odd preconditioning

We propose now to assess mock-up prediction on another benchmark, whose initial layout is defined in Section 4.3. Data patterns are beyond compiler understanding here, which does not vectorize the initial kernel once again. Even the properly reconstructed kernels are not vectorized despite apparent feasibility, except for icc complex multiply optimization.

Overall predictions are reliable with a 13% average relative error, as shown in Figure 6.2. Aside from the excellent predictions on both *AoSoA-dbl* and *SoA-dbl* restructuring – roughly 0% on average relative error on both – , slightly pessimistic *AoSoA-cplx* and *SoA-cplx* illustrate here again the gap between mock-ups naive SIMDization and compiler complex multiply optimization. This particular benchmark and previous benchmark both typically illustrate how binary to binary transformations can not compete with full compiler optimizing power, however the gap is not

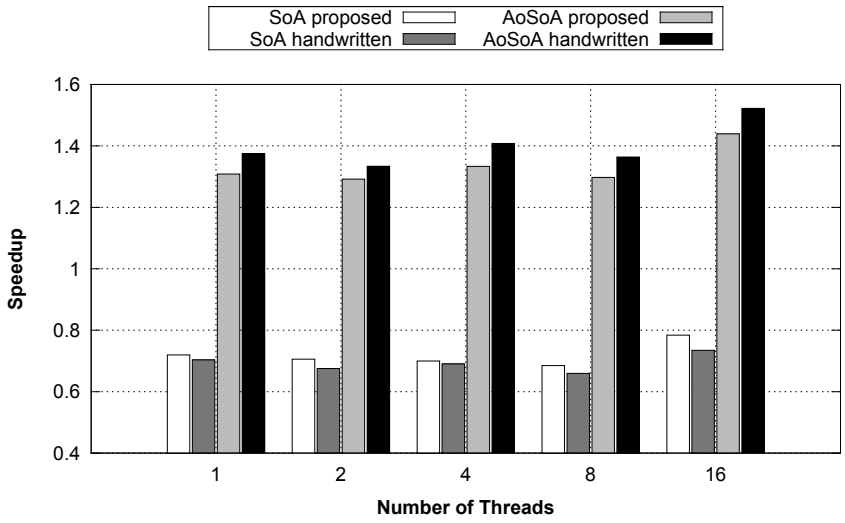


Figure 6.3: Lattice QCD Application Restructuring+SIMDization Speedup — with respect to reference using respectively equal number of threads — average relative error is roughly 4%

big enough to mislead the user, who is provided a realistic gain to expect.

6.2.4 Lattice QCD application without preconditioning

We focus on Qiral application whose initial layout definition is given in Section 4.3. The stride-forming and SIMDization-hindering structures are processed entirely, and the whole layout is used for the computations. We study two different transformations, as defined in aforementioned section, which are basically one SoA and one AoSoA, really close to the fundamental ones as the complex structure is split, thus innermost dimension is contiguous unlike *AoSoA-cplx* and *SoA-cplx* transformations we addressed before.

The initial conditions remain close to the benchmarks as far as compilation is concerned, since no SIMDization occurred, were it for initial layout or the restructured ones. However, we resort to intrinsics to vectorize the kernel of interest, which may be an interesting solution for the programmer to overcome compiler SIMDization failure, and possibly allow to improve still sub-optimized code/layout, which is precisely the very role of mock-ups proposed to highlight. Indeed, if the programmer modified kernel performance is clearly below predicted gain and the kernel has not been SIMDized, predicted gain should be achievable with intrinsics.

Predictions for *SoA* and *AoSoA* are reliable with an average relative error of 4%, as shown in Figure 6.3, as mock-up SIMDization tend closely to user restructured kernel with intrinsics. With a packed thread policy and hyper-threading disabled, the multithread context does not disrupt mock-up prediction, since here the kernel is parallel and compute-bound.

Another takeaway is that indeed AoSoA outperform SoA transformations, experiments made coroborate the relevance of our locality characterization made by our transformation space exploration, meaning we have an acceptable sense of which transformation would perform well.

6.2.5 2D cardiac wave propagation simulation application

We focus on a layout and a set of transformations defined in Section 4.3 For the sake of these experiments, we rewrote reference application code so that original array is allocated in a linearized manner, in order to provide a fair reference for performance evaluation. The application kernel of interest was not vectorized, but was successfully vectorized after data layout restructuring, consequently no intrinsics are used in rewritten user codes. Thread placement policy is packed with no hyper-threading allowed, and mock-up copy-in is multithreaded as well. We study layout restructuring impact on performance on three different datasets, corresponding to three different layout sizes. Initial layout and applied transformations are described in Section 4.3.

As one could intuitively expect, significant gain occurs after data restructuring, as high as roughly $2.4\times$ on average as shown in Figure 6.4 on Dataset-256, which is almost constant independently of the number of threads involved. Furthermore, the gain of SIMDization alone is about roughly $2\times$ which is substantial using 2-elements vectors, even though performance decreases slightly when two NUMA nodes have to be allocated when using more than 16 threads. Mock-up prediction average relative error is 9% too optimistic in this experiment. This overoptimism can be explained by the effect of too hot data on the kernel performance, because in the mock-up implementation, the copy is performed right before the kernel, which may constitute an unnatural warm-up in the context of the application.

The second kernel we study here depends on Dataset-512, whose layout size is augmented by a factor 4 with respect to Dataset-256. In this new configuration, restructuring gain is dramatically higher than before as shown in Figure 6.4, increasing with the number of threads and reaching up to roughly $14\times$ with 8 threads, as application achieves to take full advantage of all private L2 caches. Moreover, prediction remains consistently slightly overoptimistic as memory cache may be warmer before kernel execution than actual real application cache, while still being accurate with an average relative error of as low as 5%.

Dataset-1024 layout is 4 times the size of Dataset-512, and Figure 6.5 shows application reaches peak performance for 8 thread similarly to Dataset-512, at a total speedup of roughly $28\times$. Predictions accuracy is reliable, with average relative error of roughly 10%. Finally, mock-ups mimic well actual application tendencies, independently of multithread and dataset constraints. This highlights here the crucial importance of data structures design, showing how advantageous data restructuring can be for the programmer. In this very case, data layout restructuring is sufficient to enable autovectorization with an excellent gain – up to $28\times$ –, saving as well the user the time of hand vectorization via instrinsics for instance. In the end, from the programmer point of view, after the code restructuring is applied to exhibit a proper stencil, state-of-the-art stencil optimizations techniques can be contemplated for further performance improvement.

6.3 Conclusion

We have presented a methodology to evaluate data layout transformations, the in-vivo evaluation that permits to directly assess the given transformations gain or lack of gain in the context of the user application. We implemented this method at the user application binary level. We explained that using a checkpoint/restart technique we minimize the cost of such an approach, as only the kernel of interest is replayed.

The performance prediction of multiple transformations matches within 5% the performance

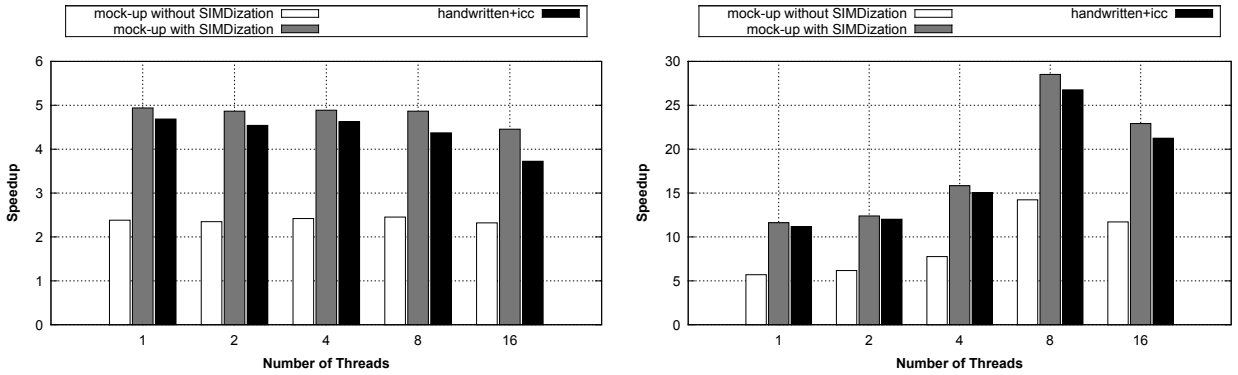


Figure 6.4: 2D Wave Propagation Application Restructuring+SIMDization Speedup on Dataset-256 (left) or on Dataset-512 (right) — with respect to reference using respectively equal number of threads — average relative error is 9% (left), 5% (right)

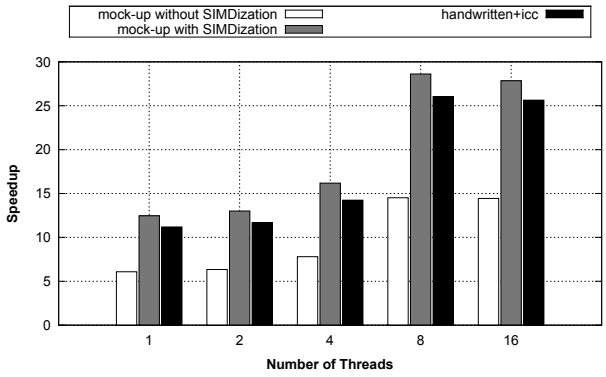


Figure 6.5: 2D Wave Propagation Application Restructuring+SIMDization Speedup on Dataset-1024 — with respect to reference using respectively equal number of threads — average relative error is 10%

of hand-transformed layout code, which shows our evaluation technique is a reliable tool for the user to select an appropriate new layout design out of a set of quantified transformations.

Conclusion and Future Challenges

Memory accesses are costly on modern architectures, as processors performance exceeds memory performance significantly. Applications are slowed down by suboptimal memory access patterns, therefore specific attention to data layouts design is required when writing high performance code. A given layout that performs well on a given architecture is not guaranteed to perform similarly well on another given architecture. Automated programs are needed to tackle this issue of appropriate layout design choice. However, compilers typically have tremendous difficulties when addressing the layout restructuring issue, particularly because of the lack of information at compile time on data structures, pointers, indirections and such. Tools on the other side help programmers locating performance issues in their applications, but do not performance optimizations guidelines with different transformations and their potential gain, as a result the user is left clueless about which layout restructuring to perform.

In turn, both compilers and tools feedback are unsufficient. We proposed a framework for data restructuring that provides the programmer with elaborate feedback, providing the user with a selection of transformations with detailed steps of applications on the source code, along with quantification of each transformation in order to help the user make a choice of restructuring based on expected gain.

6.4 Summary

Tackling the issue of layout restructuring resorts to several steps, namely the analysis of problematic layouts and patterns, then the transformation space exploration, the resulting code rewriting and the user feedback generation and finally the transformations evaluation. First, we detailed a novel formalism to express data layouts and patterns, in order to simplify the discussions on data layouts transformations and to systematically express transformations. We explained how we retrieve the initial layout/patterns definition based on memory traces collected from the user application instrumented binary, and how we can delinearize the obtained layout in order to get a multidimensional representation on a normal form, independent of any compiler optimizations. Using our formalism, we described a set of transformations to reshape the initial layout, by permuting layout dimensions or removing gaps for instance. In particular, common transformations such as AoS to SoA and AoS to AoSoA transformations were conveniently expressed. We then detailed our transformation space exploration, by applying contraints to reduce the number of possibilities as the set of transformations to be selected has to be evaluated in-vivo which is costly in terms of elapsed time. We also constraint the exploration space to preferentially select transformations suitable to SIMDization and multithread, as significant performance improvement can be obtain through this different levels of parallelisms, in particular layout has to be properly designed

not to be badly divided among threads causing additional performance issues.

Transformed layouts then have to be implemented through binary rewriting, where several major code modifications has to be applied. First of all, the initial layout must be copied to the new layout. The copy code is automatically generated and evaluated and can be proposed to the user as such for comprehension purposes. Also, all the individual memory accesses have to be remapped to properly use the new layout. We then detailed how we generate automatic user feedback to clearly notify the user of performance issues on his original application, and how to modify the code to incorporate the suggested modification with respect to each transformation proposed.

In the last chapter, we detailed our evaluation methodology relying on a checkpoint/restart technique, in order to quantify each transformation by a potential gain estimation. The idea of the *in vivo* approach is to implement each transformation on a separate binary and replay the kernel of interest which each layout version in the context of the application for performance assessment. To do so, a checkpoint call is placed right before the kernel of interest and the application is run for reference timing. Then the user application binary is instrumented using probes for each memory instruction and the kernel is re-run to obtain memory traces. After analysis, transformations and binaries rewriting, each binary version substitutes the original kernel and is evaluated. This approach has led reliable predictions below 5% in average of relative error on two real life applications, that are implementations of Lattice QCD and Cardiac wave simulation respectively with different preconditioners and different datasets, showing the quantified hints are trustworthy indicators for the programmer to make layout restructuring decisions.

6.5 Perspectives

As proposed, our data layout restructuring approach feedback allows the programmer to select an appropriate transformation with a potentially high performance gain, saving him the time of performing the code modifications by hand for each of the proposed transformation to determine only afterwards which is the best. However, in the end, following our framework suggestions the user still has to perform one transformation by hand, even though he his well guided in his process. Indeed, our approach being based on memory traces, it is consequently tailored for a given run and therefore optimizations cannot be safely performed. The user can still supervise the output code in order to make adjustements if needed to make the new code valid. User supervision is needed in all cases because neither compiler nor tools nor frameworks has the full visibility on the code and especially the input parameters or datasets that can greatly influence the resulting appropriate layout designs. In the end, the application developer is the only one that has the knowledge of the data layouts. An idea would be to inject more semantics into the source code to specify a class of layouts susceptible to be used. This would allow more general approach to layout transformations. An alternative would be to run several instrumentations to test different datasets/preconditionners or more general runtime-dependent parameters.

In some cases, the mock-up transformation can in fact be semantically equivalent and can constitute a proper code transformation, when the control is not too complicated to analyse statically in the sense that it does not comprise conditionals, indirections or difficult induction variables for instance. We can report to the programmer when it is the case. Here, profiling may help in the case compilers miss optimizations on the source but the output binary is easier to analyse, for instance when inlining has been applied. Also user interaction may be contemplated in this case to go

further, indeed sometimes indirections are used in the code but the corresponding memory access patterns are in fact regular, so one may think of a way to specify such patterns unambiguously to the compiler via pragmas for instance, to guarantee regular behavior through the indirection instructions.

Our in-vivo approach uses code instrumentation for memory traces and therefore is usually applied on a small portion of the code, usually the kernel of interest which is typically the most time consuming kernel. This constrains us to perfom a copy, which implies that the original sub-optimal layout has to be copied in a new hopefully optimized layout. The placement of the copy itself is a non trivial question. Ideally, for the user, the best case scenario would be to restructure the layout from the very definition/allocation of it, this way no overhead could be accounted during the run of the program. Alas, for our evaluation purposes in particular it is not doable, we must find a suitable spot within the kernel of interest. Theoretically, the farther from the innermost loops using the layout the better since it is where its influence is the lowest on kernel performance. However, cache effects being extremely difficult to model and therefore to predict, nothing guarantees the copy will not have a negative impact on performance, especially on the other kernels. Inversely, performing the copy close to the innermost may have benefits since the copy brings the layout elements into the cache, and thus they may already be available for use in the innermost kernels.

Besides, our approach focuses on one hotspot, usually the function which monopolizes the execution time. In some cases however, execution time is mainly shared between multiple hotspots. In this context, the local aspect of our approach is challenged, as the evaluation of the global impact of the respective hotspot is difficult. Indeed, applying a given transformation on a given hotspot may be profitable locally but nothing guarantees in theory the other hotspots would benefit from it or at least not be impaired and that the benefit would be global. Therefore, the space exploration of transformations of multiple hotspot accordingly is challenging. Because the cost of instrumentation for memory traces is high especially when multiple hotspots are involved, it may be useful to determine beforehand if the hotspots perform computations on the same data layouts. In the case where the hotspots does not use the same memory areas, transformations can be thought independently. A lighter alternative to memory traces is the page traces, where instead of tracing each of the individual memory access, we detect page faults and note which pages are accessed by which hostpots, as it has been done in [16]. It allows to pre-emptively select candidates before tracing all memory accesses. Similarly on a multithreaded context, we can also characterize which memory areas are accessed by which thread, and identify if given layouts are accessed by multiple threads. Still, the global evaluation is challenging, in particular when multiple instances of a given hotspot are performed, for instance with different parameters, or different layouts.

The operations we defined are simple sums and cartesian products of layouts. One may contemplate more elaborate operations in some specific domains such as tensor operations for QCD or deep learning. Common patterns such as convolution patterns may be analysed and expressed with an enriched formalism , with specific transformations applications according to new operators. For this matter, layout duplication may be considered, which might help alleviate the kernels of some dependences and, also might permit better pattern alignment and better vectorizations as stencils for instance uses neighbor elements at each step of computation which causes alignment issues. There is here a performance compromise with the supplementary memory space allocated, which must be evaluated. Similarly, diagonal patterns or triangle matrices and their operations may also be formalised to enrich our transformation power.

Bibliography

[1] Numpy, library for scientific computing in python. `https://github.com/numpy/numpy`.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[3] Chadi Akel, Yuriy Kashnikov, Pablo de Oliveira Castro, and William Jalby. Is source-code isolation viable for performance characterization? In *Intl. Workshop on Parallel Software Tools and Tool Infrastructures*, 2013.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[5] AMD AMD. Core math library (acml). *URL http://developer. amd. com/acml. jsp*, page 25, 2012.

[6] Olivier Aumage, Denis Barthou, Christopher Haine, and Tamara Meunier. Detecting simdization opportunities through static/dynamic dependence analysis. In *Workshop on Productivity and Performance (PROPER)*, 2013.

[7] Denis Barthou, Gilbert Grosdidier, Michael Kruse, Olivier Pene, and Claude Tadonki. QIRAL: A High Level Language for Lattice QCD Code Generation. In *Programming Language Approaches to Concurrency and Communication-centric Software Workshop*, Tallinn, Estonia, 2012. arXiv:1208.4035.

[8] Denis Barthou, Andres Charif Rubial, William Jalby, Souad Koliai, and Cédric Valensi. Performance tuning of x86 OpenMP codes with MAQAO. In *Tools for High Performance Computing*. Springer Berlin Heidelberg, 2010.

[9] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. *GPU computing gems Jade edition*, 2:359–371, 2011.

[10] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W O'hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *International Conference on Computer Aided Verification*, pages 178–192. Springer, 2007.

[11] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[12] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 2008.

[13] Pierre Boulet. *Array-OL revisited, multidimensional intensive signal processing specification*. PhD thesis, INRIA, 2007.

[14] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: a test suite and results. In *Conf. on Supercomputing*, 1988.

[15] Steve Carr, Kathryn S McKinley, and Chau-Wen Tseng. *Compiler optimizations for improving data locality*, volume 29. ACM, 1994.

[16] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(1):6, 2015.

[17] Shuai Che, Jiayuan Meng, and Kevin Skadron. Dymaxion++: a directive-based api to optimize data layout and memory mapping for heterogeneous systems. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 916–924. IEEE, 2014.

[18] Trishul M Chilimbi, Bob Davidson, and James R Larus. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, volume 34, pages 13–24. ACM, 1999.

[19] Trishul M Chilimbi, Mark D Hill, and James R Larus. Cache-conscious structure layout. In *ACM SIGPLAN Notices*, volume 34, pages 1–12. ACM, 1999.

[20] Doosan Cho, Sudeep Pasricha, Ilya Issenin, Nikil Dutt, Yunheung Paek, and SunJun Ko. Compiler driven data layout optimization for regular/irregular array access patterns. In *ACM Sigplan Notices*, volume 43, pages 41–50. ACM, 2008.

[21] Michał Cierniak and Wei Li. *Unifying data and control transformations for distributed shared-memory machines*, volume 30. ACM, 1995.

[22] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[23] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Prog. Lang. and Systems*, 1991.

[24] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *ACM SIGPLAN Notices*, volume 34, pages 229–241. ACM, 1999.

[25] Chen Ding and Ken Kennedy. Inter-array data regrouping. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 149–163, London, UK, UK, 2000. Springer-Verlag.

[26] Wei Ding and Mahmut Kandemir. Improving last level cache locality by integrating loop and data transformations. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 65–72. IEEE, 2012.

[27] H.C. Edwards and C.R. Trott. Kokkos: Enabling performance portability across manycore architectures. In *Extreme Scaling Workshop (XSW), 2013*, pages 18–24, Aug 2013.

[28] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 2004.

[29] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. Boost. simd: generic programming for portable simdization. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, pages 1–8. ACM, 2014.

[30] Louis-Noël Pouchet Eunjung Park, John Cavazos, Albert Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *ACM/IEEE Intl. Conf. on Code Generation and Optimization*, 2011.

[31] G. Carl Evans, Seth Abraham, Bob Kuhn, and David A. Padua. Vector seeker: A tool for finding vector potential. In *Workshop on Prog. Models for SIMD/Vector Processing*, pages 41–48, New York, NY, USA, 2014. ACM.

[32] Timothée Ewart, Fabien Delalondre, and Felix Schürmann. Cyme: A library maximizing simd computation on user-defined containers. In Julian Martin Kunkel, Thomas Ludwig, and HansWerner Meuer, editors, *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 440–449. Springer International Publishing, 2014.

[33] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103. Springer, 1996.

[34] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[35] Tobias Grosser, J. Ramanujam, Louis-Noel Pouchet, P. Sadayappan, and Sebastian Pop. Optimistic delinearization of parametrically sized arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 351–360, New York, NY, USA, 2015. ACM.

[36] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.

[37] Saurabh Gupta, Ping Xiang, Yi Yang, and Huiyang Zhou. Locality principle revisited: A probability-based quantitative approach. *Journal of Parallel and Distributed Computing*, 73(7):1011–1027, 2013.

[38] Christopher Haine, Olivier Aumage, and Denis Barthou. Rewriting system for profile-guided data layout transformations on binaries. *Euro-Par*, 2017 (to appear).

[39] Christopher Haine, Olivier Aumage, Enguerrand Petit, and Denis Barthou. Exploring and evaluating array layout restructuring for simdization. In James Brodman and Peng Tu, editors, *Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 351–366, Cham, 2015. Springer International Publishing.

[40] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conf. Series*, 46(1):494, 2006.

[41] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Proceedings of the 20th Intl. Conf. on Compiler Construction: Part of the Joint European Conf.s on Theory and Practice of Software*, CC'11/ETAPS'11, pages 225–245, Berlin, Heidelberg, 2011. Springer-Verlag.

[42] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Intl. conference on compiler construction: part of the joint European conferences on theory and practice of software*, 2011.

[43] Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 2012.

[44] Robert Hundt, Sandya Mannarswamy, and Dhruva Chakrabarti. Practical structure layout optimization and advice. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 233–244. IEEE Computer Society, 2006.

[45] Intel. Vtune, 2014. http://software.intel.com/en-us/intel-vtune-amplifier-xe.

[46] MKL Intel. Intel math kernel library, 2007.

[47] Julien Jaeger and Denis Barthou. Automatic efficient data layout for multithreaded stencil codes on cpus and gpus. In *IEEE Intl. High Performance Computing Conf.*, pages 1–10, Pune, India, December 2012. IEEE Computer Society.

[48] Mahmut Kandemir, Alok Choudhary, J Ramanujam, and Prithviraj Banerjee. A framework for interprocedural locality optimization using both loop and data layout transformations. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 95–102. IEEE, 1999.

[49] Mahmut Kandemir, J Ramanujam, and Alok Choudhary. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers*, 48(2):159–167, 1999.

[50] MahmutTaylan Kandemir. Array unification: A locality optimization technique. In Reinhard Wilhelm, editor, *Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 259–273. Springer Berlin Heidelberg, 2001.

[51] Alain Ketterlin and Philippe Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *ACM/IEEE Intl. Conf. on Code Generation and Optimization*, pages 94–103, New York, NY, USA, 2008. ACM.

[52] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):490–505, 2000.

[53] Peter MW Knijnenburg, Toru Kisuki, and Michael FP O'Boyle. Iterative compilation. In *Embedded processor design challenges*, pages 171–187. Springer, 2002.

[54] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The vampir performance analysis toolset. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.

[55] Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic data layout optimizations for gpus. In *European Conference on Parallel Processing*, pages 263–274. Springer, 2015.

[56] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 2013.

[57] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *Intl. J. of Parallel Programming*, 2000.

[58] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E. Nagel. Scout: a source-to-source transformator for SIMD-optimizations. In *Workshop on Productivity and Performance (PROPER)*, 2011.

[59] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 2000.

[60] Yoon-Ju Lee and Mary Hall. A code isolator: Isolating code fragments from large programs. In *Langages and Compilers for High Performance Computing*, 2004.

[61] Shun-Tak Leung and John Zahorjan. *Optimizing data locality by array restructuring*. Department of Computer Science and Engineering, University of Washington, 1995.

[62] Peng-yuan Li, Qing-hua Zhang, Rong-cai Zhao, and Hai-ning Yu. Data layout transformation for structure vectorization on simd architectures. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*, pages 1–7. IEEE, 2015.

[63] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. A compiler framework for extracting superword level parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '12, pages 347–358, New York, NY, USA, 2012. ACM.

[64] Xu Liu, Kamal Sharma, and John Mellor-Crummey. Arraytool: A lightweight profiler to guide array regrouping. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 405–416, New York, NY, USA, 2014. ACM.

[65] Vincent Loechner, Benoît Meister, and Philippe Clauss. Precise data locality optimization of nested loops. *The journal of Supercomputing*, 21(1):37–76, 2002.

[66] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, Ponnuswamy Sadayappan, Yongjian Chen, Haibo Lin, et al. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 348–357. IEEE, 2009.

[67] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[68] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 2005.

[69] Chi-Keung Luk and Todd C Mowry. Compiler-based prefetching for recursive data structures. In *ACM SIGOPS Operating Systems Review*, volume 30, pages 222–233. ACM, 1996.

[70] Deepak Majeti, Kuldeep S. Meel, Rajkishore Barik, and Vivek Sarkar. Adha: Automatic data layout framework for heterogeneous architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 479–480, New York, NY, USA, 2014. ACM.

[71] Jonathan Mak and Alan Mycroft. Limits of parallelism using dynamic dependency graphs. In *Intl. Workshop on Dynamic Analysis*, 2009.

[72] S. Maleki, Yaoqing Gao, M.J. Garzaran, T. Wong, and D.A. Padua. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 Intl. Conf. on*, pages 372–382, Oct 2011.

[73] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorization compilers. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[74] Sandya S Mannarswamy, Ramaswamy Govindarajan, and Rishi Surendran. Region based structure layout optimization by selective data copying. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 338–347. IEEE, 2009.

[75] Gordon E Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[76] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[77] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In *ACM/IEEE Intl. Conf. on Code Generation and Optimization*, 2011.

[78] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *ACM/IEEE Intl. Conf. on Code Generation and Optimization*, 2006.

[79] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 2006.

[80] Dorit Nuzman and Ayal Zaks. Autovectorization in GCC—-two years later. In *Proceedings of the GCC Developers' Summit*, 2006.

[81] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: revisited for short simd architectures. In *ACM/IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[82] David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[83] Paul M. Petersen and David A. Padua. Static and dynamic evaluation of data dependence analysis. In *Intl. Conf. on Supercomputing*, 1993.

[84] Eric Petit, François Bodin, Guillaume Papaure, and Florence Dru. ASTEX: a hot path based thread extractor for distributed memory system on a chip. In *HiPEAC Industrial Workshop*, 2006.

[85] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *ACM SIGPLAN Notices*, volume 37, pages 101–112. ACM, 2002.

[86] Matt Pharr and William R. Mark. ispc: A SPMD compiler for high performance CPU programming. In *Conf. InPar*, 2012.

[87] Easwaran Raman, Robert Hundt, and Sandya Mannarswamy. Structure layout optimization for multithreaded programs. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 271–282. IEEE, 2007.

[88] James Reinders. *VTune performance analyzer essentials*, volume 14. Intel Press, 2005.

[89] Bin Ren, Gagan Agrawal, James R. Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. SIMD parallelization of applications that traverse irregular data structures. In *ACM/IEEE Intl. Conf. on Code Generation and Optimization*, 2013.

[90] Gang Ren, Peng Wu, and David Padua. Optimizing data permutations for SIMD devices. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 2006.

[91] Probir Roy and Xu Liu. StructSlim: A lightweight profiler to guide structure splitting. In *ACM/IEEE Intl. Conf. on Code Generation and Optimization*, 2016.

[92] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *ACM SIGPLAN Notices*, volume 37, pages 140–153. ACM, 2002.

[93] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 440–451, Washington, DC, USA, 2012. IEEE Computer Society.

[94] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ACM SIGARCH Computer Architecture News*, volume 40, pages 440–451. IEEE Computer Society, 2012.

[95] Matthew L Seidl and Benjamin G Zorn. Segregating heap objects by reference behavior and lifetime. In *ACM SIGPLAN Notices*, volume 33, pages 12–23. ACM, 1998.

[96] Kamal Sharma, Ian Karlin, Jeff Keasler, James R. McGraw, and Vivek Sarkar. Data layout optimization for portable performance. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par*, pages 250–262, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[97] Xipen Shen, Yaoqing Gao, Chen Ding, and Roch Archambault. Lightweight reference affinity analysis. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 131–140. ACM, 2005.

[98] Sameer S Shende and Allen D Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[99] Jaewook Shin, Mary Hall, and Jacqueline Chame. Superword-level parallelism in the presence of control flow. In *ACM/IEEE Intl. Conf. on Code Generation and Optimization*, 2005.

[100] Robert Strzodka. Abstraction for aos and soa layout in c++. *GPU Computing Gems: Jade Edition*, 2011.

[101] I-Jui Sung, G.D. Liu, and W.-M.W. Hwu. Dl: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*, pages 1–11, May 2012.

[102] I-Jui Sung, Geng Daniel Liu, and Wen-Mei W Hwu. Dl: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*, pages 1–11. IEEE, 2012.

[103] Salvador Tamarit, Julio Mariño, Guillermo Vigueras, and Manuel Carro. Towards a semantics-aware transformation toolchain for heterogeneous systems. In *Program Transformation for Programmability in Heterogeneous Architectures (PROHA) workshop*, 2016.

[104] Xinmin Tian and BR De Supins. Explicit vector programming with openmp 4.0 simd extension. *Primeur Magazine*, 2014, 2014.

[105] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[106] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 2009.

[107] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 327–337. IEEE, 2009.

[108] Dan N Truong, François Bodin, and André Seznec. Improving cache behavior of dynamically allocated data structures. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 322–329. IEEE, 1998.

[109] Ying-Yu Tseng, Yu-Hao Huang, Bo-Cheng Charles Lai, and Jiun-Liang Lin. Automatic data layout transformation for heterogeneous many-core systems. In Ching-Hsien Hsu, Xuanhua Shi, and Valentina Salapura, editors, *Network and Parallel Computing*, volume 8707 of *Lecture Notes in Computer Science*, pages 208–219. Springer Berlin Heidelberg, 2014.

[110] Reinhard v. Hanxleden and Ken Kennedy. Relaxing simd control flow constraints using loop transformations. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 1992.

[111] Brice Videau, Vania Marangozova-Martin, Luigi Genovese, and Thierry Deutsch. Optimizing 3d convolutions for wavelet transforms on cpus with sse units and gpus. In *Intl. europar conference on parallel processing*, 2013.

[112] Wei Wang, Lifan Xu, John Cavazos, Howie H. Huang, and Matthew Kay. Fast acceleration of 2d wave propagation simulations using modern computational accelerators. *PLoS ONE*, 9(1):1–10, 01 2014.

[113] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

[114] David C. Wong, David J. Kuck, David Palomares, Zakaria Bendifallah, Mathieu Tribalat, Emmanuel Oseret, and William Jalby. Vp3: A vectorization potential performance prototype. In *Workshop on Programming Models for SIMD/Vector Processing*, Feb 2015.

[115] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *ACM SIGPLAN Notices*, volume 48, pages 57–68. ACM, 2013.

[116] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.