Data Layout Types: a type-based approach to automatic data layout transformations for improved SIMD vectorisation

by

Artjoms Šinkarovs

Submitted for the degree of Doctor of Philosophy
at Heriot Watt University
on Completion of Research in the
School of Mathematical and Computer Sciences
5th August 2015

Abstract

The increasing complexity of modern hardware requires sophisticated programming techniques for programs to run efficiently. At the same time, increased power of modern hardware enables more advanced analyses to be included in compilers. This thesis focuses on one particular optimisation technique that improves utilisation of vector units. The foundation of this technique is the ability to chose memory mappings for data structures of a given program.

Usually programming languages use a fixed layout for logical data structures in physical memory. Such a static mapping often has a negative effect on usability of vector units. In this thesis we consider a compiler for a programming language that allows every data structure in a program to have its own data layout. We make sure that data layouts across the program are sound, and most importantly we solve a problem of automatic data layout reconstruction. To consistently do this, we formulate this as a type inference problem, where type encodes a data layout for a given structure as well as implied program transformations. We prove that type-implied transformations preserve semantics of the original programs and we demonstrate significant performance improvements when targeting SIMD-capable architectures.

To *Elizabeth*,

Пожалуй, лучшей из всех Елизавет.

Acknowledgements

I wish to thank both of my supervisors *Sven-Bodo Scholz* and *Greg Michaelson* for their valuable ideas and great patience.

Ideas presented in Chapter 3 are based on a project which started in terms of “Google Summer Of Code 2010”, with *Artjoms Šinkarovs* being a student and *Richard Güenther* being a mentor. I wish to thank the GCC community for their help during the coding. Special thanks to *Richard Güenther*, *Richard Henderson*, *Joseph S. Myers* and *Jakub Jelinek* for their valuable comments and patch reviews.

Finally, many thanks to all my friends and family members who have supported me during my research.

Contents

Publications

Some ideas and figures have appeared previously in the following publications.

Journal papers

1. Artjoms Šinkarovs, Sven-Bodo Scholz, Robert Bernecky, Roeland Douma, and Clemens Grelck. SAC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. *Concurrency and Computation: Practice and Experience*, 26(4):952–971, 2014. http://ashinkarov.github.io/publications/sexynbody.pdf

2. Artjoms Šinkarovs and Sven-Bodo Scholz. Type-driven data layouts for improved vectorisation. *Concurrency and Computation: Practice and Experience*, 2015. http://dx.doi.org/10.1002/cpe.3501 (visited on June 2015)

Conference papers

1. Artjoms Šinkarovs and Sven-Bodo Scholz. Portable support for explicit vectorisation in C. In *16th Workshop on Compilers for Parallel Computing (CPC'12)*, 2012

2. Artjoms Šinkarovs and Sven-Bodo Scholz. Data layout inference for code vectorisation. In *International Conference on High Performance Computing & Simulation, HPCS 2013, Helsinki, Finland, July 1-5, 2013*, pages 527–534, 2013. http://ashinkarov.github.io/publications/data-layouts.pdf

3. Artjoms Šinkarovs and Sven-Bodo Scholz. Semantics-preserving data layout transformations for improved vectorisation. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 59–70, New York, NY, USA, 2013. ACM

Extended abstracts

1. Artjoms Šinkarovs and Sven-Bodo Scholz. Data layout inference for code vectorisation. In *The 24th symposium on Implementation and Application of Functional Languages*, IFL 2012, 2012

2. Artjoms Šinkarovs and Sven-Bodo Scholz. Functionally redundant declarations for improved performance portability. In *The 25th symposium on Implementation and Application of Functional Languages*, IFL 2013, 2013

Notations and definitions

$\mathbb{N}$	Natural numbers starting with zero: 0,1,2,...
$\mathbb{Z}_+$	Natural numbers starting with one: 1,2,3,...
`div`	A binary function denoting integer division. In the context of this thesis it is applied on $\mathbb{N}$ numbers only. Notation is: a `div` b, which denotes $\lfloor \frac{a}{b} \rfloor$. For example: 7 `div` 3 ≡ 2.
`mod`	A binary function denoting modulo operation. In the context of this thesis it is applied on $\mathbb{N}$ numbers only. Notation is: a `mod` b, which denotes $a - \lfloor \frac{a}{b} \rfloor b$. For example: 7 `mod` 3 ≡ 1.
data layout	A mapping of a data structure into flat memory.
$e :: \tau$	An expression e has data layout encoded by τ.
$e : \sigma$	An expression e is of type σ, e.g. integer, a vector of integers, etc.
CPU	Central Processing Unit (CPU) is an electronic device within a computer that executes instructions of the specified program performing arithmetic, logical, input/output operations specified by the program.
GPU	Graphics Processor Unit (GPU) is an electronic device within a computer that is designed to accelerate creation of images that will be output on a display. Nowadays GPUs are used not only for display-related graphics computations but also for general applications. This approach is usually called General-Purpose computing on Graphics Processing Units (GPGPU) and it is being actively used in high-performance systems.

HPC	High Performance Computing (HPC) is an area of research and engineering that has to do with very high-level computation capacities. Usually, the interest of HPC lies in building supercomputers and running applications on them efficiently.
ISA	Instruction Set Architecture (ISA) specifies a set of commands available for a given CPU.
SIMD	Single Instruction, Multiple Data (SIMD), is a class of parallel computers in Flynn's taxonomy [43]. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously.
SIMT	Single Instruction, Multiple Thread (SIMT) is an execution model and abstraction on top of SIMD where multiple independent threads execute concurrently using a single instruction. Used primarily in GPUs.
SPMD	Single Program, Multiple Data (SPMD) is a technique of parallel programming when one and the same task is being executed in parallel on different input data.

Chapter 1

Introduction

The problems are solved, not by giving new information, but by arranging what we have known since long.

Ludwig Wittgenstein [147]

According to the TOP500 (November 2014) [138] the world fastest supercomputer achieves about 60% of its peak performance. This figure is obtained using a set of benchmarks called LINPACK [37] which is considered by most supercomputer manufacturers while building the machines. Realistically, this number can be treated as performance upper bound of a given machine. Such a difference between the peak and sustained performances is usually referred as “performance gap” [118]. HPC machines cost millions of pounds, and in the light of exascale computing, such a price is likely to increase. Losing 40% of the performance is an economical waste, therefore there is an increasing need to tackle this problem and close the performance gap.

H. Seutter in his “Free lunch is over” manifesto [135] gives accounts for the roots of the problem in the context of CPUs. He demonstrates that the usual assumption that next generation CPUs will get faster does not hold anymore. Instead of CPUs with higher frequencies manufacturers switched to CPUs with higher number of cores. F. Pollack in [111] formulates his famous “rule” which says that performance of uni-processors increases with the square root of their complexity. R. Ronen et. al in [114] demonstrate that single-threaded performance does not scale with frequency and area. Memory latencies and bandwidth would not scale as well. Power density of a microprocessor is reaching levels close to nuclear reactors. As a consequence, old programs as they are, originally designed for sequential hardware, do not benefit from new multiprocessor architectures because the code is not aware of newly available resources.

Nowadays modern hardware trends propose even more radically diverse architecture designs. While GPUs or coprocessors offer higher computing power at lower costs, their success on the mass market suddenly makes the overall design of ordinary computers not that different from HPC machines. This brings all the existing HPC challenges, most importantly the performance gap, to ordinary machines.

The big questions are how to write new programs for diverse hardware architectures and how to make the existing programs run efficiently? When a supercomputer costs billions of pounds and is bought to run a single program, one can hire a team of programmers to optimise the application to perfection. Although, even in such idealistic scenario, at the example of TOP500 and LINPACK we can see how challenging such a task actually is. For the clusters built of multicores, the gap is relatively small: the machines three and four in the TOP500 list (as of November 2014) achieve 85% and 93% of their peak performance respectively. The machine number one in the same list is equipped with Xeon Phi [66] coprocessors, achieving 60% of the peak performance. The same percentage is achieved by the machine number two which is equipped with NVIDIA GPUs [146].

If we are required to "squeeze" some performance out of an ordinary machine, the manual optimisation of every program in most cases is not economically viable. Not only is it unrealistic to find a highly professional programmer for every existing program, but also hardware-specific manual optimisations proliferate the amount of code variants of an algorithm. Those variants have to be maintained, supported, tested, etc., which contradicts modern software engineering practices.

An alternative approach to manual optimisation is *compiler technology*. The power of compiler technology lies in its ability to replicate the effort of manual optimisations making them applicable to all the programs recognised by a compiler. To achieve the effects of manual optimisation from within a compiler is a very challenging task as the process has to be formalised not only for a single application but for as wide as possible class of applications. Nevertheless, we believe that at least for mainstream programs, this is the only way to preserve performance portability on diverse hardware and to make existing programs benefit from it.

This thesis focuses on the one particular aspect that contributes significantly to the performance gap, which is underutilisation of vector instructions. Vector instructions provide a way to apply an operation on a vector of values simultaneously. The length of such vectors depends on the type of the data the vector stores. Today vectors typically can accommodate four or eight values of 4-byte real values. Every core of a CPU is equipped with vector units. This means that if the core part of an application is left unvectorised, the consequences are twofold. First, performance may be n times slower than in the vectorised case, where n is a number of elements per vector. Secondly, such a slow-down will happen on every core, which means that for multi-threaded applications the overall slow-down will be even larger. Given that the length of vector registers increases with new generations of hardware, such a penalty will grow larger.

There exists a number of limiting factors for getting full performance on a vector-capable architecture. Considering past experiences of manual vectorisation of scientific and high performance applications, it can be seen that such a process often requires radical program rewrites [46, 68, 6]. Such rewrites are challenging. First of all, there is no widely adopted standard to target vector operations from within the program. Usually the choices are either to program vector devices directly using machine instructions (possibly in a form of intrinsics) or to rely on auto-vectorisers. Neither of the approaches are satisfactory. Direct programming in assembly is tedious, error-prone, and most importantly the result is non-portable. Auto-vectorisers tend to improve the portability problem by introducing an optimisation that analyses a program, identifies the parts that can be executed using vector instructions and rewrites the code accordingly. Despite the great advances in the algorithms used in auto-vectorisers [39, 96, 74] the performance of the resulting code is far beyond the processor peak.

Part of the reason for auto-vectorisers to fail is the inability to recognise a number of important patterns [88]. Secondly, fully implicit approach means that there is no way to help an auto-vectoriser to recover from failure. Finally, and this is going to be the main interest of our investigation, the way data structures are mapped in memory has a very large effect on an application's vectorisation potential.

Generally, for vector instructions to be efficient, the data that they operate on must be consecutive in memory. However, most programming languages directly relate type definitions and type declarations to one particular data layout in memory. For example, Fortran maps arrays in a column-major order into memory, whereas C uses a row major scheme. The fields of records or objects are typically adjacent in memory and algebraic data types such as those in many functional languages typically are mapped to pointer connected graphs in memory.

In languages that support an explicit notion of memory and pointers there is very little that can be done about this tight coupling between types and their corresponding mapping into memory. One can create yet another level of abstraction between objects/arrays and the way they are being accessed for further manual data layout manipulation. However, such a level of abstraction has to be maintained and for it to be efficient, the underlying compiler has to know how to eliminate it in the resulting binaries. In contrast, languages that completely abstract away from the notion of memory, such as purely functional languages, allow for almost arbitrary mappings of data into memory. While this opportunity may be less relevant in the context of algebraic data types, it can have a huge impact on the performance of programs that operate on arrays.

It is well known from the optimisation of compute-intensive applications in Fortran or C, that a reorganisation of order of data accesses can vastly improve the overall runtime [16, 139, 87]. Improvements typically do not only stem from better cache locality but also from improved chances of the utilisation of vector instructions. However, data dependencies in programs and a fixed data-layout often constrain what can be achieved. The high-performance computing literature provides many cases where re-writes for enforcing different *memory layouts* are crucial for achieving a sufficient level of performance [77, 107, 151, 150, 136].

1.1 Thesis

“The process of identifying data layouts that are better suited for vectorisation than those directly specified and corresponding semantic-preserving program transformations can be automated.”

1.2 Approach

To modify data layouts in a systematic way we have to be able to track data layouts of all the data structures in a given program. As data structures in principle can be mapped in memory in various different ways, we have to find a way to choose the best configuration of data layouts for a given hardware. Our approach is to use types to encode constraints on the mapping of each data structure and employ a type system to resolve those constraints. By treating data layouts as types, cross-function optimisation becomes possible and type checking guarantees that data layouts in a program are consistent. Type inference techniques allow to reconstruct data layouts for every expression in a program with a programmer providing any data layout annotations.

The newly inferred layouts imply program transformation, as access and traversals of data structures depends on their mapping in memory. Furthermore, one and the same function may be applied in various contexts where its arguments may have different layout types. The implication is that several versions of one and the same function have to be created. Finally, external interfaces of a program may imply memory mappings for certain data structures. Such mappings must be preserved in the original form.

After the program is transformed we have to generate the code that will be able to target vector instructions of the underlying hardware. Our goal is to generate such a code in a portable fashion, yet guaranteeing performance portability across SIMD-capable hardware.

1.3 Contributions

The major contributions of this thesis are as follows:

1. We formalise the idea of data layout annotations using types and we introduce a type inference that is capable to generate all the valid layout configurations (according to the type system) for a given program.

2. We introduce automatic high-level program transformations based on the previously inferred layout types and prove that transformed programs preserve the semantics of the original programs.

3. We solve the vector portability problem by extending the C language with explicit vector operations, which we have implemented in the context of GNU

GCC.

4. We implement the proposed inference, transformations and vectorised code generation in terms of the programming language called SaC and its compiler called `sac2c`.

5. We evaluate our approach using a set of benchmarks which is known to be challenging to vectorise and demonstrate the effectiveness of vectorisation by comparing the runtimes with the automatically and manually vectorised C versions.

1.4 Thesis structure

The thesis consists of eight chapters.

Chapter two gives an overview of the existing vectorisation techniques and approaches and presents work done so far with respect to data layout mappings.

Chapter three presents an extension of the C language implemented in terms of the GNU GCC compiler providing performance portable means of expressing vector operations in C.

Chapter four introduces the input language and layout type system used to infer all the possible data layouts for a given program.

Chapter five introduces our program transformation scheme with respect to vector-capable hardware.

Chapter six discusses technical details of the inference and transformations in terms of the `sac2c` compiler.

Chapter seven presents a performance evaluation before we conclude in Chapter eight.

Chapter 2

State of the art

History may not repeat itself but it does rhyme.

Joseph Anthony Wittreich [148]

In this chapter, we explore the context around vector hardware. We start with very basic principles, then, to understand design decisions and potential future developments, we consider the evolution of vector architectures starting from early machines till our time. Next we discuss what kind of vector hardware is available on the market today and classify functionality of commonly available vector operations. Then we consider the way vector hardware can be programmed and what are the main difficulties while doing so. Finally, we discuss the work in the area of data layout modification and its applications.

2.1 Vector hardware basics

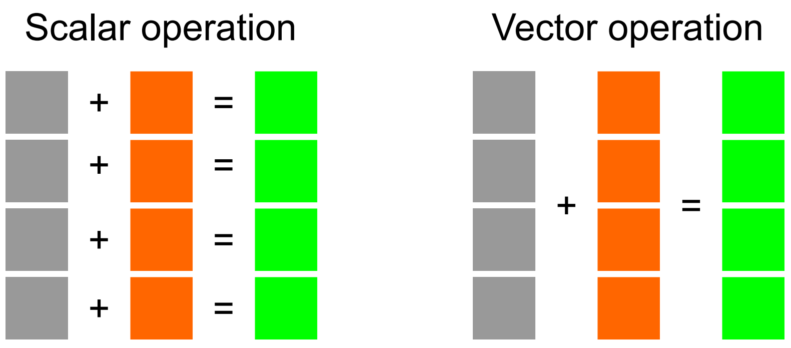


Figure 2.1: Scalar vs SIMD addition.

When we talk about vector hardware, technically we mean SIMD (Single Instruction Multiple Data) parallel hardware according to Flynn's taxonomy [43]. A distinctive feature of such a hardware is availability of multiple processing elements that are capable to perform one and the same operation simultaneously. Such a classification consider only macroscopic level of hardware organisation, therefore the exact implementation of SIMD processors may vary significantly. Nevertheless, to develop an intuition, let us consider a classical use case of SIMD hardware. As it can be seen from Fig. 2.1 a SIMD operation replaces multiple identical scalar operations with a semantically equivalent one, e.g. four additions are replaced with a single vector addition. Keep in mind that the performance of such a SIMD operation, the number of elements that can be processed simultaneously and programming interfaces fully depend on the architecture.

2.2 History of parallel and vector hardware

SIMD architectures are a special case of parallel architectures and their design has a lot of historical reasons. By looking at how parallel hardware appeared and evolved, we can explore the versatility of the existing designs and get a better understanding of why certain things are as they are and what we could expect in the future.

As a general observation, a lot of decisions in computer science are based on trade-offs. For some algorithms, memory consumption can be increased in favour of faster execution times. Sometimes we can decrease the precision of an answer in favour of faster computations, and so on. In hardware design, trends shift when a certain technology gets cheaper or simpler. Tannenbaum in [137] gives the following example when discussing the existence of caches:

> In particular, it frequently happens that a change in technology renders some idea obsolete and it quickly vanishes. However, another change in technology could revive it again. This is especially true when the change has to do with the relative performance of different parts of the system. For instance, when CPUs became much faster than memories, caches became important to speed up the “slow” memory. If new memory technology someday makes memories much faster than CPUs, caches will vanish. And if a new CPU technology makes them faster than memories again, caches will reappear. In biology, extinction is forever, but in computer science, it is sometimes only for a few years. [137, Section 1.5.7, page 44]

As we shall see in the rest of this chapter, such a pattern is generic. In the context of our work it means that instead of focusing on hardware-specific solutions we want to understand basic underlying principles of the problem, as any specific hardware may be rendered obsolete in favour of future technologies.

The second observation is that early machines were designed for a restrictive application domain. The main purpose of first computers was to serve as large programmable calculators. Starting from Charles Babbage [52] designing a mechanical calculator to tabulate polynomial functions in the 19th century; continuing with early electro mechanical computers like the Harvard Mark I which were used for war-related calculations like problems associated with the protection of ships from the destructive action of magnetic mines [34]. Most of the later machines were reusing ideas and technologies of previous generations and even nowadays a large part of instructions of the modern CPUs have to do with numerical operations. We would rarely find instructions designed specifically for domains like text or sound processing.

After the success of early machines in 1940s, the diversity of tasks that could have been solved with later computers substantially increased. In the 1960s it became clear that for a number of real world problems like linear programming or solving sets of partial differential equations over grids the computing power of existing machines was several orders of magnitude lower than required. It became clear that strictly sequential computers would not be able to provide desirable computing speeds [11] because of the signal propagation speed barrier. This gave a rise to the idea of using multiple processing units on a single machine.

A variety of designs has been proposed in the early stages of research. For example, one of the earliest papers [130] presents a SOLOMON computer which proposes a design based on a 2-d array of processing units each of which is interconnected with its four neighbours, yet controlled by a single control unit. The SOLOMON itself was never actually built, but the ideas were used in further projects like ILLIAC IV [11].

Another kind of parallel design manifested itself in vector processors which still exist and are being actively developed nowadays. The first vector machines like TI-ASC [143] and STAR-100 [58] appeared in the early 1970s and gave rise to vector supercomputers. The main idea lies in having a vector unit that takes streams of data from memory, operates on them and puts results back into the memory. Conceptually such a vector operation was happening in a single instruction, although that individual scalar operations (steps of the vector operation) did not have to happen simultaneously. Such a design required advanced high-bandwidth memory systems together with long pipelines to make vector operations efficient. On the other hand, the entire loop could have been reduced to a single instruction. That in itself was a great advantage, as it removed the necessity to stall the pipeline at every element and spending time on fetching and decoding instructions of the loop body. Let us consider example codes taken from [54] to demonstrate the approach. This is a classic DAXPY benchmark which can be formulated as:

$$Y = a \cdot X + Y$$

where a is a scalar and X and Y are vectors.

```
        L.D      F0,a          ;load scalar a
        DADDIU   R4,Rx,#512    ;last address to load
Loop:   L.D      F2,0(Rx)      ;load X(i)
        MUL.D    F2,F2,F0      ;a x X (i)
        L.D      F4,0(Ry)      ;load Y(i)
        ADD.D    F4,F4,F2      ;a x X(i) + Y(i)
        S.D      0(Ry),F4      ;store into Y(i)
        DADDIU   Rx,Rx,#8      ;increment index to X
        DADDIU   Ry,Ry,#8      ;increment index to Y
        DSUBU    R20,R4,Rx     ;compute bound
        BNEZ     R20,Loop      ;check if done
```

Listing 2.1: Scalar code implementing DAXPY

The scalar version is presented in Listing 2.1, the vector version of the same code is presented in Listing 2.2.

```
L.D          F0,a          ;load scalar a
LV           V1,Rx         ;load vector X
MULVS.D      V2,V1,F0      ;vector-scalar multiply
LV           V3,Ry         ;load vector Y
ADDV.D       V4,V2,V3      ;vector add
SV           Ry,V4         ;store the result
```

Listing 2.2: Vector code implementing DAXPY

The disadvantage though was the high latency of the vector unit and as a result very long switches between vector and scalar modes. This problem was addressed in the CRAY-1 [116], which according to [40] was the beginning of "Golden Era" of vector supercomputers. The solution was to introduce vector registers, very similarly to what we have nowadays. The advantage was that the data in registers could be reused, in case an algorithm performed multiple operations on the same data. Vector registers allowed one to save successive memory reads and writes. This approach had some limitations — vector registers were expensive in terms of circuitry, so only a limited number could be provided. That had an impact on the sizes of the vectors that could be processed. Vector operations had to be performed in chunks limited by the length of vector registers, but multiple operations could have been executed on each chunk.

Complex instructions Early machines were equipped with what is now called CISC[1] instructions. Some of the instructions were representing very complex operations. For example, TI-ASC provided matrix multiply in a single instruction, or IBM 3090 [18] vector load instruction which were combining multiple actions in one instruction. This is helpful in case one writes code in assembly directly, however, when targeting such an architecture from a compiler, it gets very hard to do so efficiently.

Dawn of vector supercomputers[2] In the early 1990s it became technologically possible to put many more transistors on a single die than earlier. Microprocessors made a significant progress, their clock frequency steadily increased every year and their performance started to be comparable with processors used in expensive vector supercomputers. Due to their mass production microprocessors also were substantially cheaper than tailor-made ones used in supercomputers. That introduced a shift in supercomputer design towards building machines using many microprocessors.

2.3 Vector architectures today

Despite the shift of technologies in supercomputers, the key ideas that were understood in vector supercomputers manifest themselves in modern CPUs and are

[1]Complex Instruction Set Computing (CISC) is a CPU design where single instructions can execute several low-level operations.

[2]This paragraph is based on [40].

known as SIMD extensions. Later the same SIMD extensions appeared on GPUs and accelerators like Intel Xeon Phi [66]. These extensions are the primary focus of our thesis.

The important property all the extensions share is that SIMD operations are just usual instructions, same as scalar arithmetic operations or data movement operations. To execute a SIMD instruction its operands have to be loaded into SIMD registers. SIMD registers are not very different from scalar registers, except for their size. For example, if we assume that the operation on Fig 2.1 uses registers, then SIMD registers on that picture can accommodate four scalar elements. The exact length of SIMD registers varies from platform to platform.

The length of a SIMD register is most commonly defined in bits, rather than in a number of scalar elements. Similarly to scalar registers, the type of its value is not fixed and may be interpreted differently depending on the instruction that is being executed. The same holds for SIMD registers — a 128-bit SIMD register can accommodate four 32-bit floats or 16 eight-bit characters.

All SIMD extensions provide instructions to load data from memory into SIMD registers and to store data from SIMD registers back into the memory. The way it works is that a SIMD register is mapped into a continuous region of memory of the size equal to the SIMD register. On some of the architectures, like for example Intel, load and store instructions can be of two kinds: aligned and unaligned. Aligned load/store implies that the memory address used to load/store a SIMD register is a multiple of the size of a SIMD register. Aligned operations are usually faster. Application of aligned move operations to unaligned memory results in a hardware exception.

Operations provided by SIMD extensions vary significantly on different platforms, but usually all of the extensions provide some form of arithmetic operations on integers and floats, some element reordering operations, comparisons and mathematical functions. For more details on the functional classification please refer to Section 2.4.1.

2.3.1 CPUs

The primarily motivation to introduce SIMD extensions on CPUs was to accelerate multimedia-related tasks. That is why very often early SIMD instructions were referred as “multimedia extensions”. Most of the modern CPUs nowadays are equipped with some form of SIMD instructions. Here are several examples: MVI (Alpha) [115], Altivec (IBM) [47], MAX (HP) [83], MMX/SSE/AVX (Intel) [60], VIS (SPARC) [132], NEON (ARM) [5], etc. All these architectures provide a set of vector registers and some form of floating point arithmetic, logical operations, comparisons and data movements.

The table in Fig. 2.2 shows the evolution of various SIMD instruction sets in time. From these figures we can see that the size of vector registers doubles every few years. Such a growth suggests that SIMD technology is unlikely to disappear in

ISA extension name	**Year**	**Bits**	**First appeared**
MAX-1	1994	32 bits	HP, PA-7100LC
MAX-2	1996	64 bits	HP, PA-8000
MMX	1997	64 bits	Intel, Pentium I
ALTIVEC	1998	128 bits	Apple, IBM, Motorola
3DNOW!	1998	64 bits	AMD, K6
SSE	1999	128 bits	Intel, Pentium III
SSE2	2001	128 bits	Intel, Pentium IV
SSE3	2004	128 bits	Intel, Pentium IV
SSSE3	2006	128 bits	Intel, Xeon
SSE4	2007	128 bits	Intel, Core 2
AVX	2011	256 bits	Intel, Sandy bridge
AVX2	2013	256 bits	Intel, Haswell
MIC	2012	512 bits	Intel, Xeon Phi

Figure 2.2: Length of a vector register

the near future, unless a very radical breakthrough in hardware happens. Also, there is still a large room for improvement. For instance, the latest vector supercomputer SX-ACE [91] from NEC is equipped with vector registers which can accommodate 256 doubles. That is an order of magnitude larger than what we have on standard CPUs. That suggests that speedups we can achieve today due to SIMD will get larger on the newer generations of hardware.

SIMD operations on the Xeon Phi accelerator are not very different from those that can be found on latest Intel CPUs. The important difference though is the long vector register (512 bits) and support for predicated execution of operations, i.e. normal operations like for example addition can get a mask that determines which of the elements in the register to add. Predication is very useful when expressing conditions in vector form.

2.3.2 Graphics Processing Units (GPUs)

Latest generations of Nvidia's GPUs introduce [98] SIMD instructions as part of their pseudo-assembly language specification. At first glance it seem to be controversial as GPUs according to Flynn's taxonomy are clearly SIMD processors. However, SIMD instructions provide much more fine-grain parallelism than the same code implemented in a usual GPU programming model — when a kernel is executed by a number of SIMT threads. Most importantly, for the diverging program flow, SIMD operations allow to avoid lock-step execution, which is the way SIMT threads evaluate a conditional statement.

The usefulness of having SIMD instructions on the GPUs is demonstrated for example in the work of Y. Liu et. al. [86] where authors demonstrate significant speedups of the protein sequence database search algorithm over existing GPU implementations. The main reason lies in gaining more data parallelism beyond the SIMT execution model on CUDA-enabled GPUs because of using n CUDA PTX SIMD video instructions.

Currently the set of supported SIMD operations is not very rich: PTX ISA v4.1 [98] defines integer addition and subtraction, average, absolute differences, minimum/maximum and masking. Most likely the capabilities of SIMD instructions on GPUs will get more and more advanced, which means that any results obtained for SIMD instructions on a CPU can be transfered to GPUs.

2.4 SIMD functionality

As it can be seen from Fig. 2.2 there exists a large number of different SIMD instruction sets available on the market. Despite being non-uniform in the way the code has to be expressed and exact operations they support, they all share common functionality. We are going to survey this functionality based on the analysis of instruction sets provided by three most widely used manufacturers. After that we will consider potential difficulties when integrating SIMD operations into programs.

2.4.1 Classes of SIMD instructions

We propose a macroscopic classification of the operations of the following instruction sets: Intel (MMX/SSE/AVX) [60], Altivec [47] and ARM's NEON [5]. The classification is concerned with the functionality of vector operations that a programmer can use in the code replacing scalar operations with vector ones. By joining the functionalities of individual architectures we get the following list.

Data movement defines instructions to load and store data from vector registers to memory and vice versa.

Scientific codes like stencil computations or image processing often require non-continuous memory access. Such an operation, when elements are being loaded/stored from/to non-continuous memory are usually referred as *gather*/*scatter*. Some of the latest architectures provide native support for scatter/gather operations, but it is limited and not very efficient.

One important data movement operation that is found on almost all (but not every) architecture is *broadcast* which assigns a scalar value to all the elements of a vector.

Basic arithmetic defines arithmetic operations like addition, multiplication, etc. which usually work as $a_i + b_i$, where a and b are vector operands.

Advanced arithmetic defines operations that involve more sophisticated patterns compared to basic arithmetics. As an example Intel SSSE3 introduces horizontal operations like PHSUBW, PHSUBD which for two operands a and b compute $[a_0 - a_1, a_2 - a_3, \dots b_0 - b_1, b_2 - b_3, \dots]$. ARM NEON's VPADD, VPADAL instructions operate in a similar fashion, performing addition.

Another example of advanced arithmetic are MIN/MAX instructions that chose a minimum/maximum element of the given vector.

Comparison operations define the element-wise application of standard comparison operations like equal, less than, etc. The result is usually a vector of boolean values which indicates result of the comparison for every pair. Later such a result is commonly used in masking operations which select components either from the first or second operand according to the boolean vector. Such operations are essential for vectorising the code containing control-flow like:

```
for (i = ...)
    if (a[i] > 3)
        b[i] = 5;
    else
        b[i] = 6;
```

in which case it is being transformed into something similar to:

```
for (vec_i = ...)
    vec_b[vec_i] = mask (vec_lt (vec_a[vec_i], [3,3,...]),
                         [5,5,...], [6,6,...])
```

where `mask` choses elements from the second operand at positions where the first operand is true and from the third operand at positions where first operand is false.

Finally, some of the architectures have instructions to set the flag that can trigger branching if all the elements in a boolean vector are true/false. For example Intel's PTEST instruction sets the Zero flag to the result of bitwise and of all vector components. Altivec defines *all* and *any* variants of comparison operations which are element-wise conjunctions and disjunctions of the vector elements accordingly.

Mathematical operations defines the element-wise application of a mathematical operation like *sin*, *sqrt*, etc. Most of the mathematical instructions operate on a single or floating point vectors, however for the sake of classification, we do not fix the type of mathematical operations.

Logical operations are element-wise applications of scalar logical operations like bitwise and, bitwise or, etc.

Shift operations define element-wise shifts of vector components.

Conversion operations define conversions between different data types. For example, a vector of integers can be converted to a vector of floats, or first two elements of a vector of four floats can be converted into a vector of two doubles.

Element reordering define instructions that change the order of elements in a given vector. Those operations are very broad, and can vary from interleaving odd and even elements from two vectors to reversing elements to performing a general permutation of elements.

Operations on parts of the vector operate not on all the elements of the vector but on some subset. For example, indexing operations and assigning an element of a vector are of that nature. Intel SSE defines a set of operations with a *-SS* postfix which operate on the low values of vectors only.

Complex operations define operations that combine several actions in one. For example: multiply-add $(a := a + (b \times c))$, absolute difference $(|a| - |b|)$, etc.

Application-specific instructions define instructions that implements application-specific functions in hardware. For example Intel AVX introduces a set of AES[3]-specific instructions for faster cryptography.

Such a classification covers nearly all the instructions of the architectures we have considered. Please note that as SIMD instructions encode the expected type of the elements in SIMD registers, we have to be careful when we say that a certain functionality is present. It often happens that an architecture supports an operation on vectors of floats, but does not support the same operation on 8-bit integers. This is especially noticeable on Intel-based CPUs as SIMD extensions were introduced gradually, not only introducing functionally new operations, but extending the types that the existing operation can be applied to.

Some specific instructions like memory prefetching or data movement without polluting caches are left outside of this classification. Their functions do not reflect any logical operations that help to express parts of algorithms, but they are more of architecture-specific tricks to optimise data movement.

Most of the SIMD instructions in the proposed classification have standard scalar counterpart, i.e. their functionality can be directly mapped into several scalar instructions from the basic instruction set of a given architecture. However, there are exceptions from this rule, specifically we are talking about group of instructions for:

- saturated arithmetics[4],
- half-floating point (16-bit floats) arithmetics, and
- estimated mathematical functions, for example estimated reciprocal — a faster but less precise version of the expression $1/x$

which all the three manufacturers support. The proposed classification covers the functionality that these instructions provide, but from a programmability perspective we believe that they deserve a special treatment.

[3]Advanced Encryption Standard (AES) is a specification for the encryption of data established by the National Institute of Standards and Technology (NIST) in 2001.

[4]A standard arithmetic on integers, but in case the result of an operation overflows, either a minimum or maximum value of the integer type is returned.

2.4.2 Known difficulties

Programs that target SIMD architectures in replacing some scalar instructions with vector ones does not necessarily results in faster execution. One can construct a number of cases when SIMD implementations will run slower than its scalar analogues. The main reasons for that are as follows:

Scatter/Gather Gathering data into SIMD registers and scattering it into the memory is tricky and can be inefficient. Some of the architectures do not provide any instructions to address this, those that provide them are not very efficient. This means that whenever operations include non-continuous access to memory, a programmer has to make choices on how to implement it.

Alignment For sequential memory loads one has to be aware of alignment issues. Some of the architectures due to technical limitations with load/store operation require memory address to be aligned. This means that whenever one loads/stores non-aligned data, two vector loads/stores are required. That might be done via a special instruction, to avoid manual scaffolding, but the execution time will be still lower than the aligned counterpart. In order to tolerate that one would have to align the data in memory (if possible) or fallback to unaligned loads/stores in case alignment is unknown, like in the case when a function gets a pointer.

Missing instructions Some instruction sets might not include operations that might be required in terms of an algorithm. For example shuffling or rotates. In that case one could try to emulate an operation using existing instructions, but the risk is that its performance would be so bad that all the achieved gains will be lost. Alternatively one could fall back on scalar mode, but again, instruction pipelining, and accessing vector elements might destroy all the vectorisation gains.

Conditions When vectorising conditions, the usual technique is to execute both branches and then mask the result. This is inefficient from the power perspective as we compute vector elements which are going to be discarded. In case conditions are nested we will have to execute twice the number of operations than in the original program. To address this problem, the latest instruction sets introduce predication. This allows one to pass a mask to every vector instruction and avoid operations on the elements that will be discarded. If predication is not present, it has to be carefully emulated. This is tricky and might be inefficient.

Loops In imperative languages like C or Fortran loops are the natural source of data parallelism. However, building a generic loop vectorisation framework is challenging, as such a framework has to perform non-trivial code analysis and capture pretty large context. Loops might contain conditions, control flow or other loops. The iterations of the loop might have dependencies. The iteration space of the loop might be unknown at compile time.

2.5 Programming models

Programming SIMD architectures is a big challenge, as it requires to reformulate algorithms in a data-parallel style and to take into account all the above-mentioned difficulties. Programming SIMD architectures in a portable way is a double challenge, as the means of programming either target a limited number of instruction sets or have severe limitations on expressibility. Finding the right programming abstractions that will maximise program performance on the SIMD-capable architectures is an open problem. Performance characteristics of the instructions vary from architecture to architecture at the same time exactly those characteristics determine the way algorithms have to be expressed.

The existing programming models for SIMD architectures very often depend on the surrounding context. The way a programming model and a programming language target parallelism in general very much determine how one can program SIMD. To understand the SIMD choices better we are going to review them in the context of programming models and languages for general parallelism.

2.5.1 Programming languages overview

Almost all of the early programming languages happened to be imperative. One of the reasons for that is that most of the early computer architectures were based on the von Neumann model [141] which is imperative itself, and a lot of languages were developed by creating abstractions for the parts that were hard to read or long to write. The first such an abstraction was an assembly language which was just a symbolic representation of machine codes. Further, languages like Fortran [9], Algol [92], Cobol [117] started to appear, providing higher and higher levels of abstractions. As a consequence, the mapping from the language into hardware was not so obvious anymore, plus, when a level of abstraction reached a certain threshold the complexity of compilers increased dramatically. That in turn had an influence on the quality of the generated code.

The next non-trivial problem was automatic refactoring. Programming languages do evolve and introduce new features especially on the early stages of development. Sometimes it comes as a desire to abstract a certain hardware feature, like for example the ++ operator being a direct abstraction of an increment instruction; sometimes it can be a revisit of some earlier design decisions. However, such changes in programming languages impact existing codes, requiring them to be modified. Ideally, one would like to refactor the code automatically and to get new language features for free. Unfortunately, this is not a simple task, especially in the imperative setup. Realisation of this problem raised a number of proposals explaining how to make languages more analysable. Peter Landin in [80, 81] demonstrated how the semantics of Algol 60 can be formalised using Church's lambda calculus. Later in 1977 John Backus named his Turing award lecture [10] "Can Programming Be Liberated from the von Neumann Style?", where he proposed to use functional programming to reason about programs using what he calls an algebra of programs. It seems that the trend was to get rid of imperative programming style and use a functional style instead. One could claim that this is still valid today.

The main difficulty with such a language shift is the amount of code that exists and is actively used in day-to-day life. Large projects like OS kernels or compilers have accumulated millions of lines of code during several decades and they are still being actively developed. Rewriting them in a new language is a major undertaking, and as not all the imperative constructions can be easily mapped into declarative languages, automating this process is arbitrarily hard. Finally, before embarking on such a rewriting journey, one has to be absolutely sure that the new compilers will generate code at least as good as the existing one. Unfortunately, none of the existing functional languages can give such guarantees today.

A lot of mainstream programming languages today were originally designed for sequential hardware, therefore they do not provide any parallel constructs. When intending to target parallel hardware using such languages, there is a choice of how to express parallelism. Most commonly in practice the following three ways are being used:

1. To use low-level abstractions like inline assembly in the case of SIMD instructions or threading libraries in case of multicores. The problem with this approach is the lack of code portability — to support a new hardware a new version of the code has to be written.

2. Extending a language. The problem with language extensions lies in finding a compromise between genericity of extensions and tight control of the underlying hardware.

3. Recognising parallelism by means of program analysis. Although this is a very desirable solution, as we mentioned above, analysing imperative languages is a very challenging problem.

Further we are going to consider existing solutions to the parallel language conundrum as well as to the general SIMD portable programming problem.

2.5.2 Low-level approaches to SIMD programming

The standard way of programming SIMD directly is either by using assembly (if a language allows, it can be inline assembly), or to use compiler intrinsics. The latter is a compiler-defined functions that expand into specific assembly instructions. As we have mentioned earlier, such an approach is bound to produce non-portable code. At the same time, it allows to explore all the possible capabilities of the architecture, as the programming happens at the lowest possible level.

Library-based

To abstract things away, a number of projects propose tailor-made libraries to target SIMD instructions [142, 75, 41]. For example, M. Kretz et. al. describe a C++ library called *Vc* [75] and they demonstrate that the overheads of the library are negligible which means that it can be used as a replacement for intrinsics or assembly kernels. In despite of someone has to create and maintain a library, updating it whenever a new architecture appears etc., for performance reasons the level of abstraction that the library creates has to be eliminated by the underlying compiler. Guaranteeing such a behaviour is very difficult, but the performance penalty might get rather high.

Intrinsics to intrinsics mapping

To increase a portability of architecture-specific codes [152] presents a mapping from the ARM intrinsics to MMX/SSE intrinsics, allowing one to run applications created for ARM-tuned applications on Intel.

Virtual instruction sets

The HSA[5] foundation proposes [45] a design of a processor which integrates heterogeneous hardware, most importantly CPUs and GPUs, on the same bus and provides a unified programming model to target them. HSA introduces virtual memory address space which can be mapped into the main memory and the memory of GPUs. The virtual instruction set is kept ISA-agnostic, providing support for parallel sections of the algorithm. Virtual instructions are dispatched to concrete devices by the runtime system which is a part of the HSA programming model. One of such devices can be SIMD, although while the project is at the design phase and no actual hardware is built, for the time being it is difficult to evaluate the validity of such an approach in application to SIMD.

R. Bocchino et. al. [15] address the problem of portability of SIMD instructions by introducing a virtual instruction set specially designed for vector operations. The main design criteria is to provide an abstraction for various vector architectures and architecture classes. Architecture classes include sub-word SIMD like Intel SSE and PowerPC Altivec and streaming processors like RSVP [30]. The proposed virtual instruction set allows both arbitrary length and fixed vectors providing asynchronous load and store semantics for long vectors and introducing alignment attributes. The compilation scheme involves a compiler which can generate a portable vector code and a translator with full information about the target architecture and system configuration.

[5]Heterogeneous System Architecture (HSA) foundation is a not-for-profit industry standards body focused on making it dramatically easier to program heterogeneous computing devices.

2.5.3 Annotations

One of the most pragmatic and yet powerful approach to extending a programming language with extra functionality is to come up with a set of annotations on top of the existing syntax. OpenMP [102] has started out as a set of extensions for C and Fortran for multi-platform shared memory multiprocessing programming. OpenMP uses a fork-join multithreaded execution: it spawns a group of tasks at the beginning of a parallel section and waits until all the tasks are terminated at the end of the section. OpenMP provides a set of directives to orchestrate the process. A typical example is:

```
#pragma omp parallel for private(i) shared(x)
for (i = 0; i < N; i++)
    x[i] = x[i] + 1;
```

The annotation on top of the for-loop informs the compiler that the loop should be executed in parallel, that the variable `x` should be shared between the threads and that the variable `i` is private for the individual threads. The variable `i` is an iteration variable of the loop, so every thread would range over the slice of the iteration space. How exactly the iteration space is going to be divided between the threads depends on the implementation of OpenMP. However, the default schedulers can be controlled for every parallel section.

Comparing with manual approaches for multi-threaded execution, like for example when using pthreads [93], OpenMP is much simpler for a programmer and the resulting code is suitable for both single threaded and multi threaded executions. The annotations can be ignored by means of a compiler switch. The latest OpenMP standard (version 4.0) allows one to use the code on GPUs.

OpenMP 4.0 [103] and *Cilk+* [62] introduce high-level constructs targeted at SIMD instructions. Cilk+ introduces a pragma called `simd` with the following description: "The pragma can be applied to a loop, to indicate the intention that the loop needs to be implemented using vector instructions." OpenMP 4.0 introduces an analogous prgama called `pragma omp for simd`. Both extensions introduce a way to annotate a function operating on scalar elements, which creates a version of the function where scalar operations are projected into corresponding SIMD operations, and as a result a function can be applied to SIMD arguments. Cilk+ calls this *elemental functions*, and OpenMP 4.0 — *declare simd construct*.

The limitations of the annotation-based approach are first of all that the annotations have to be inserted manually. That raises risks for introducing race conditions and synchronisation bugs which are really hard to find. OpenMP uses a threading model called fork-join model in which program execution branches off at designated points in the program and has to be joined at a subsequent point, resuming sequential execution. Such a high level threading abstraction does not allow one to fine-tune the parallel execution. Finally, a compiler has to support OpenMP which implies that various compilers may not produce equally performing code (time wise).

OpenACC [101] is another set of compiler directives for C and Fortran with a primary goal to offload computations to accelerators. A typical code looks like this:

```
#pragma acc data copyin(A[0:n*n],B[0:n*n], n), \
                copyout(C[0:n*n]), create(tmp,i,j,k)
{
    #pragma acc parallel loop private(tmp)
    for (i = 0; i < n; i++)
        #pragma acc loop private(tmp)
        for (j = 0; j < n; j++) {
            tmp = 0;
            #pragma acc loop reduction(+:tmp)
            for (k = 0; k < n; k++)
                tmp += A[i*n+k] * B[k*n+j];


            C[i*n+j] = tmp;
        }
}
```

This is a matrix multiply example. The annotations on top of the loops are very similar to what we have seen in the case of OpenMP. They indicate that the annotated loop should be executed in parallel and specify which variables are not shared. In reduction loops, as in the case of the inner loop that sums results into `tmp` variable, the annotation indicates the reduction operation and reduction variable. Also, we have to specify which data has to be transferred to an accelerator and which should be copied back to the host.

OpenCL [73] is another popular approach that allows to target CPUs and accelerators like GPUs. OpenCL positions itself as a standard for writing programs to be executed on heterogeneous systems. It is not directive-based, instead it views a computing system as consisting of compute devices which normally are CPUs and accelerators, and it defines a C-like language to create kernels that execute on compute devices. This approach still allows one to reuse existing programs written in C, but it requires one to introduce a lot of boilerplate code to establish communication with compute devices.

The C-like language for kernel programming defines abstractions to express SIMD operations. In case kernels are executed on a CPU, depending on the implementation of OpenCL, these operations may be mapped into SIMD extensions of the given processor. In case kernels are executed on GPUs, the abstractions may be mapped into SIMD instructions of a graphics card.

2.5.4 Extensions of the existing languages

An alternative to using annotations is to introduce new constructs in the language. The main difference between the two approaches is that annotations can be easily ignored and the program would be still usable in terms of the original language, where extensions irrevocably change semantics of programs.

Languages like UPC [20], Coarray Fortran [94], HPF [56] follow this approach. UPC extends C99, Coarray Fortran extends Fortran 95/2003 and HPF extends Fortran 90. HPF (High Performance Fortran) uses a data parallel view of computations which allows one to spread a computation on a given array across several processors. HPF defines an implicit parallel statement such as FORALL and introduces a way to define side-effect-free procedures via the IMPLICIT keyword. Additionally HPF allows one to control how data is being distributed across processors by annotating data allocations. Finally it defines additional routines for message passing, in case parallelism does not fit into HPF model and a number of library routines.

UPC and Coarray Fortran are based on the PGAS [32] model. The main idea of PGAS (Partitioned Global Address Space) is to introduce locality awareness on top of message passing. PGAS presents distributed memory as a continuous address space. However, the address space is also logically partitioned between threads or processes which allows local computations, and at the same time all the memory is shared which allows SPMD style programming. Both UPC and CAF use source-to-source translation, producing the C or Fortran code appended with communication routines. The main difference between the languages is that UPC allows uniform access to any memory cell (potentially with some time overheads) when CAF is strict with respect to the boundaries of distributed arrays. In the case of the need to access remote memory a special syntax has to be used.

All the above mentioned examples did not target SIMD instructions as their primary goal. The main architectures in mind were multi-threaded machines or clusters. However, the presence of FORALL-like parallel constructions in the language allows a compiler to chose how to implement an operation. For example a compiler can chose to express it via SIMD instructions.

2.5.5 Fully implicit parallelism

All the languages we have discussed so far require a programmer, in one form or another, to take decisions on how to parallelise the program. For example explicit annotations as in case of OpenMP or the way data is going to be distributed across the cluster. In principle, there is nothing wrong with being explicit, however, by providing explicit information one lowers portability of such a code. For example, different architectures may require different ways of program parallelisation. As a consequence, in the explicit case a program would have to be rewritten manually.

Languages with fully implicit parallelism solve this problem by prohibiting any user annotations on how to parallelise programs. It is assumed that a programmer provides a specification and nothing more and then the compiler, by means of analysis and heuristics decides how to execute it for a given architecture. This shifts responsibility from programmers to a compiler. Also, in case a compiler made a wrong or inefficient choice, a programmer can do very little. However, assuming that compilers do a good job, we will get a program which is much more likely to be ported on a novel hardware architecture than its explicit counterparts; and a programmer would be able to concentrate his or her time on more important issues than low-level optimisations. The question whether compilers can make better choices than humans is of a philosophical nature, however one might think of the following analogy. In the early days when most of the programs were written in assembly a lot of programmers sincerely believed that languages like Fortran would never be able to replace skilled assembly programmers. In principle it might be true, but from the practical perspective, we do not have a lot of assembly code around these days. Implicit languages are an important instrument in a toolbox of a performance engineer, the more is done by a compiler successfully, the more time an engineer has to work on

Examples for languages with fully implicit parallelism are SAC [50] and NESL [13]. Both are functional languages and both were developed in the early 1990s. NESL uses the sequence as a basic building block to express parallelism, and the main power of the language lies in ability to flatten algorithms on nested sequences, preserving parallelism. SAC uses arrays as building blocks, borrowing ideas from APL [65] (not the syntax though), defining all the APL-like combinators via a single data-parallel primitive called the with-loop. This approach allows compilers to perform aggressive cross-operator optimisations and to generate code performance-wise compatible with C programs. The compiler `sac2c` [119] in its current state supports compilation for shared memory machines and for NVIDIA GPU cards. Both languages are strongly typed first order functional languages.

Another language that builds on the ideas of NESL is DpH (Data Parallel Haskell) [21]. The functional setup of Haskell seem to be very useful to support data parallel operations. However, the fact that it uses lazy evaluation makes it rather difficult to generate high-performing parallel code. DpH introduces unboxed arrays, which are the main building blocks for data parallelism and explicit primitives that allow one to distribute an array between threads and join it back. These mechanisms allow one to implement data-parallel operators and a number of techniques like flattening (similar to NESL) and fusion (similar to SAC with-loop fusion) are employed to make it efficient.

Automatic vectorisation

Automatic vectorisation is one of the standard ways to target SIMD and is being supported by most of the existing compilers. The general idea is to perform program analysis, identify the parts of a program that can be executed on the underlying SIMD hardware and rewrite those parts accordingly. Automatic code vectorisation has been known [2, 48] since the appearance of first vector supercomputers. Most of the programs those days were written in imperative languages like Fortran. Consequently all the vectorisation techniques assumed a Fortran-like input language. The main challenge there is to identify identical operations and rewrite them into vector form. The main source of identical operations is a loop, or generally speaking a loop-nest. When transforming loops into vector form like as proposed in [2] the main challenge is to identify dependencies between the statements in the loop nest. This can be understood from the following example:

```
for (i = 0; i < N; i++)
    x[i] = x[i] + e;
```

where all the iterations can be done concurrently; however, consider this:

```
for (i = 0; i < N-1; i++)
    x[i+1] = x[i] + e;
```

where dependencies make this loop non-vectorisable. A lot of research [105, 76, 79] has been dedicated to developing a theory of dependencies and further loop transformations.

A more advanced technique which continues the idea of loop dependencies is the polyhedral model [42]. In classical dependency analysis [71], the fundamental representation is a dependency graph. All the reasoning is based on this graph. In the polyhedral model, the dependency graph is represented by means of a system of linear inequalities. The representation allows one to use linear programming techniques to identify all semantically preserving orders in which the statements inside the loop-nest can be executed. The order in which iterations are being executed is called a schedule. Depending on the kind of optimisation we are after, we might chose different schedules.

Both, the polyhedral model and classical dependency analysis allow one to automatically vectorise some programs. The problems with both approaches are the following:

- As optimisations are performed on an imperative language it becomes difficult to deal with function calls inside loop nests. Functions are not pure, as they might have side effects, and in case a function cannot be inlined, the analysis might just give up. The effect system is not decomposable across function boundaries, so vectorisation decisions have to be taken locally.

- Data layouts are statically fixed. In a number of cases that inhibits vectorisation, as strided loads and stores are inefficient for both cache locality reasons and vector load/store reasons.

Modern compilers like GCC [44], LLVM [28] and ICC [61] are equipped with auto-vectorisers. Auto-vectoriser's code base is large and complex — occupying about 25 000 lines of code in GCC, an auto-vectoriser considers data dependencies of the loops, internals of the target architecture and using a cost-model it determine whether it is reasonable to vectorise a given loop. A cost model is the essential part of any auto-vectoriser. SIMD instructions, as any parallel technology can not only increase performance of programs, but also decrease it dramatically when being used inconsiderately. Many factors have an influence on vectorisation. SIMD instructions can take different number of cycles to execute; memory alignment has a large effect on some architectures; high level operations can be implemented with a large number of expensive SIMD instructions. In order to tackle these problems most modern compilers include a cost model for every supported CPU, where each instruction is annotated with its execution time. Based on this information the auto-vectorisers checks if a certain vectorised operation does not decrease performance (in comparison with the original code).

An automatic vectoriser does not require any effort from the side of a programmer, who just compiles the code and the vectoriser uses knowledge and heuristics to do the job. This is a very pragmatic approach to exploit SIMD instructions. As a result, the program is being decoupled from a programmer's intuition about hardware and in case patterns are recognised, the original program starts to perform much better with no effort from the side of a programmer. However, in case an auto-vectoriser fails to identify vectorisable parts of a program, there is no way for a programmer to pass extra information to it.

2.5.6 New languages

There are a number of proposals for new languages that improve the programmability of vector instructions. As one such example consider the ISPC[6] language [109] proposed by Intel. The language uses what they call an SPMD on SIMD computation model which is very similar to the GPU programming approach, but in the case of ISPC exclusively targets the SIMD instructions of normal CPUs. The syntax is based on C89 [3] with some support for C99 [63] features. The underlying execution model assumes multiple programming instances running concurrently. The group of running program instances is called a gang and usually equals the number of SIMD lanes available. The gang always runs on a single core and never introduces implicit thread creation or context switches. Writing an ISPC program largely reminds one of writing a kernel for CUDA or OpenCL. For example primitives like the thread identifier in the gang and the number of threads in the gang are made explicitly available to a programmer. Explicit synchronisation primitives are not present as convergence guarantees during the execution are stricter. The language introduces `uniform` and `varying` qualifiers for data types to mark if a variable has to be replicated across program instances or not. The language allows one to annotate an array of structures to be stored in memory as a hybrid structure of arrays which allows to store multiple fields of multiple data structures adjacent in memory. That in turn allows one to load and store data into SIMD very efficiently. Pointer operations are extended to SPMD, which allows one to mimic vector operations on pointers.

Another language that is concerned with targeting SIMD instructions of standard CPUs is Vector Pascal [33]. Vector Pascal enriches the standard Pascal language with data parallel abstractions inspired by APL and functional programming. Arithmetic operations are applicable to whole arrays and APL-like combinators (reduce, slice, etc.) are added as new primitives. The language is concerned with data parallelism and uses sophisticated code generation techniques to implement data parallel operations efficiently on SIMD-capable hardware.

[6]Intel SPMD Program Compiler (ISPC) is a compiler for a variant of the C programming language, with extensions for "single program, multiple data" (SPMD) programming. ISPC targets the SIMD units of CPUs and the Intel Xeon Phi architecture. For more details see `https://ispc.github.io/`

2.5.7 High productivity languages[7]

In 2002, DARPA launched the HPCS [38] (High Productivity Computing Systems) programme to fund research and development of high performance computing systems focusing on high productivity. The main motivation of the project was an observation that modern HPC systems lack programmability. As HPC systems became more and more advanced it is getting more and more difficult to exploit the resources of such systems. The languages developed under the programme should increase productivity by supporting general parallelism and separating algorithms from implementations.

As a result of this program three new languages were proposed: X10 [23], Chapel [22] and Fortress [1]. Fortress is a language developed by SUN with the intention to be syntactically very close to mathematical notation, yet providing all the abstractions for handling parallelism. It is assumed that parallelism would be achieved implicitly via constructions like *parallel for*, *do ... also do ... end*, tuples, etc.; however explicit threads can be spawned as well. The memory consists of *locations*, which are placed in specified *regions*, which in turn are organised in a tree structure, describing the structure of the machine. The Fortress implementation is interpreted running on top of JVM.

X10 is developed by IBM and is an extension to Java. X10 uses the notion of *places* which is a computational unit with a local shared memory. Each X10 program runs over a set of places, where each place either hosts data or runs an *activity*. X10 also differentiates values (read only data) from referenced objects (read and write data). A number of explicit synchronisation mechanisms like barriers are introduced in order to drive parallel execution. X10 compiles down to Java.

Chapel does not directly extend the syntax of any existing language but borrows it from the languages: C, Fortran, Java and Ada. It does support object oriented features, and similarly to X10 introduces a notion of *value classes*. When normal classes are passed by reference, value classes are similar to C structures. Similarly to X10, Chapel introduces a notion of *locales* which intuitively corresponds to a shared memory node in a cluster. A program is presented with an array of locales which represents the memory of the machine that can be accessed. Parallelism is mainly achieved implicitly via parallel constructions. The Chapel compiler generates C code.

All of the languages support explicit distribution of data structures (mainly mutli-dimensional arrays) across the available memory. The mapping is done explicitly by choosing a distribution for a particular data.

Similarly to the languages discussed in Section 2.5.4, HPCS did not include any explicit means to target SIMD instructions. Nevertheless, the availability of general parallel constructions makes potential vectorisation of such programs easier.

[7]This description is largely based on the Michele Weiland report [144]

2.6 Data layout transformations

In our work we have identified that fixed data layouts in many cases prevent efficient vectorisation. The observation that fixed data layouts prevent some of the optimisations and relaxing data mappings can lead to more efficient code is not new and was observed in various contexts. Let us explore the existing techniques in the context of data layout problem.

2.6.1 Data distribution on clusters

When computation happens on a distributed system, the data used in a program is distributed across the nodes of the system. Normally, communication between the nodes is expensive, so ideally we want all the data required for a computation to be available on a local node. In order to do so one has to decide how to distribute the data in the first place. This problem is addressed by Kennedy et. al. [72] where they propose an automatic tool that performs a whole program analysis, generates layouts for program parts and uses integer programming techniques to select optimal layout combination for the overall program under a given cost model. [110] solve a similar problem, but additionally to data distribution they ensure that in terms of one parallel loop, arrays are aligned towards each other with respect to offsets, strides and general axes relations. For clusters such an alignment results in more efficient runtime. Their technique allows to analyse whole programs including branching, loops and nested parallelism. As a result of such an analysis arrays might change dimensionality, or padding, or store data in a strided fashion, i.e. use every n-th element.

2.6.2 Optimisation of cache misses

Data layouts are known to be used for optimising cache misses. One of the simplest techniques briefly discussed in [29] is to transpose an array. In case of two dimensions, transpose means switching from row-major to column major representation. One of the simplest examples to illustrate the benefit is a standard matrix multiply: $C_{ij} = \sum_k A_{ik} B_{kj}$. From the indexes we can see that if the data layouts of A and B are the same, then one of the arrays will have strided access, which increases cache misses. In case A is row-major and B is column-major, both accesses will happen with stride one. [113] decrease cache misses by adding paddings to the arrays used in a program. Paddings are added on the individual dimensions of the arrays as well as between the allocated arrays. This is useful because, as the authors identified, when arrays are allocated at addresses which are multiples of the cache size, they might be mapped into the same cache-line when loading the data. This results in a cache miss on every access. This does not happen when paddings are introduced. [24] investigates usage of non-linear data layouts. Non-linear means that the mapping of the array indexes cannot be expressed as a linear function. They consider two cases of such layouts:

blocked layout and Morton order. The idea to cut arrays into blocks comes from the result obtained in [78] that if an array of size $t_R \times t_C$ is continuous in memory and fits in cache then it creates no self-interference misses. Such a block is used as a building block for new data layouts. One layout that is considered converts a two-dimensional array $m \times n$ into a four-dimensional $m \texttt{ div } t_R \times n \texttt{ div } t_C \times m \texttt{ mod } t_R \times n \texttt{ mod } t_C$. Another layout considered is Morton order. Morton order is one of the recursive data layouts, discussed in [25] and can be described as follows: divide the original matrix into four quadrants, and lay out these sub-matrices in memory in the order NW, NE, SW, SE, then apply the same procedure to every quadrant. The stopping point could be either when 1×1 size is reached, or as in the case of [24] when the quadrant size reaches $t_R \times t_C$ which is laid in memory using row-major order. Both cases introduce better spatial locality, and as authors suggest, are better suited for hierarchical memory systems. Finally, [145] introduces a source to source compiler for the C language that implements matrices in Morton order.

Typical questions for the layout type analysis are:

1. What to do with conflicting data layouts? For example, in the case of transposing arrays, what happens if the same array is referenced in two expressions which imply contradicting layouts. We can either abandon layout transformation for such an array, or try to estimate which layout is more beneficial, or to change the layout dynamically.

2. How to make sure that transformation does not decrease performance? We can be either very conservative and reject programs that might decrease performance or we can try to use a cost model to decide.

3. Applicability. Are there any factors which make it impossible to transform layouts? One of the factors could be hard-analysable language constructs. For example, in C one can obtain a pointer into an array and access data via this pointer. And what happens if a program is split into modules which are compiled separately?

2.6.3 Vectorisation

When looking at manual optimisations of data layouts for better vectorisation, transposing the data becomes very important as it improves vector loads and stores. These transformations are commonly known as "transforming array of structures into structure of arrays" or (AoS-to-SoA). Here is an example described in [108]. Assume that we store an array of triplets in the following way:

```c
/* Define a structure that holds three elements. */
struct triplet {
    double x, y, z;
};

/* Define an array of triplets of size N. */
struct triplet A[N];
```

and we access it like this:

```
for (i = 0; i < N; i++) {
    ...
    A[i].x
    ...
}
```

Let us assume that the above loop can be vectorised over `i`, which means that we replace V subsequent loop iterations with vector operations, where V is a number of elements in a SIMD register. In that case we will have to load V components of the array `A` at positions $i, i+1, \ldots, i+V-1$. This creates a strided access into memory. To accommodate this we need to reshuffle the elements. This is what is called AoS-to-SoA transformation. There are number of different ways to do this:

1. We can do it dynamically on every load. Some of the architectures provide instructions to make such a load. If not, we can manually access individual components in memory and put them in the corresponding positions of a vector. Usually this is not very efficient, as we mix vector and scalar instructions, which affects pipelines and strided access affects caches. Alternatively we can load $3V$ elements into SIMD vectors, and reshuffle them within registers. In that case after reshuffling we get three vectors with `x`, `y` and `z` components accordingly. This improves memory access, but reshuffling pattern gets rather complicated and might be inefficient.

2. We can change data layouts and store a transposed version of `A` in memory:

   ```
   struct triplet_tr {
       double x[N], y[N], z[N];
   };

   struct triplet_tr A_tr;
   ```

   This can be done locally i.e. before entering a loop we want to vectorise, we transpose an array, either in-place or copying data to a newly allocated memory and then we update data accesses to the array replacing it with accesses to the transposed data structure. For the considered example it will look like:

   ```
   /* Transpose.  */
   for (i = 0; i < N; i++) {
       A_tr.x[i] = A[i].x;
       A_tr.y[i] = A[i].y;
       A_tr.z[i] = A[i].z;
   }

   /* Updated references in the loop.  */
   for (i = 0; i < N; i++) {
       ...
       A_tr.x[i]
       ...
   }

   /* Update array A, in case it is in use.  */
   ```

   Note that if the elements of the array `A` are in use after the loop, array `A` has to be updated by copying data from the `A_tr`. This approach is described in [106], where the authors pay attention on how to do vectorise the transpose itself.

   Alternatively, data layouts can be changed globally. For our example it means that we replace all the references to `A` with a modified reference to `A_tr` which means that we do not need to copy memory at runtime, but the price for that is a whole program analysis.

The program analysis that is required to change data layouts automatically across the whole program is highly non-trivial process. There were a number of attempts to formalise such a process, for example in the work of O'Boyle et. al. in [100, 99]. The authors describe an algebraic transformation framework for data layouts and introduce how it can be modified with polyhedral-like local loop transformations. The main idea is to present data accesses to arrays inside a loop-nest as a system of linear inequalities. Further, layouts of the arrays can be transformed in a systematic way even in the presence of loop transformations. However, the global layout transformations involving data-layout-related questions formulated above are left out of the scope.

2.7 Summary

From this chapter we have seen that in principle automating the process of changing data layouts is technically possible. If the desired data layout is known, there are very powerful code generation techniques allowing to achieve this. From previous manual optimisation experiences we have seen that automating support for certain kinds of data layouts for better vectorisation is very desirable. However, none of the works as it seems explains how to choose data layouts in a systematic way for the overall program, how to express the transformed code in a portable fashion and how to make sure that the code with adjusted layouts preserves semantics of the original program. These are the questions that we are going to answer further in the thesis.

Chapter 3

Portable vectorisation as a backend

In this chapter we present an abstraction layer implemented as a set of C language extensions within the GNU GCC compiler which provides an interface for SIMD vectors and operations independently from the architecture. First of all, these abstractions allow to exploit SIMD extensions of a CPU explicitly, which is useful when an auto-vectoriser fails. Secondly, the abstractions are general enough to be mapped to any hardware supporting SIMD paradigms; hence the new abstractions could be considered as a step forward to a new C language standard. Most importantly we argue that for the best performance and the least implementation effort, the abstraction layer for explicit SIMD should be implemented as a language extension, which allows to reuse large parts of the existing infrastructure of compilers like auto-vectorisation backends and general optimisations on scalars.

3.1 Introduction

In an ideal world we would like all the inner-loop refactoring to happen automatically under the hood of a compiler. A lot of successful research in this direction has been done already [95, 131, 112, 28]. Most modern compilers including GNU's GCC, Intel's ICC, and LLVM are equipped with some form of auto-vectoriser. However, even the smartest auto-vectoriser is bound to be limited to the code pattern it has been programmed to recognise.

We are concerned with the cases when auto-vectorisers fail despite dealing with some code that can be transformed into a SIMD suitable form. In that situation, it would be desirable if a programmer could explicitly instruct the compiler where to insert SIMD instructions. While this can always be achieved by inserting inline-assembly into the program, this constitutes an inherently non-portable solution. Not only does this imply a lock-in into a particular architecture, it also inhibits an immediate benefit from larger vector sizes in the next generation.

Secondly, it puts some additional burden on the programmer as she has to acquire an in-depth understanding of the architecture that is being targeted. She also has

This chapter is based on the CPC-2012 paper [124] to make sure that the interfacing between the inline assembly and the C context is handled properly which either requires some difficult to read and maintain wrapper code or the use of intrinsic operations, which are provided by most modern compilers. However, these are typically translated into literal wrappers which improve readability but do not resolve any of the other issues.

Also, an inline assembly approach inhibits any optimisations across these operations such as constant propagation, code reorganisation or any optimisation that requires in-depth knowledge of the operation. Truth to be told, this is applicable to compilers that can target multiple architectures. For instance the Intel compiler can incorporate some of the intrinsics into optimisation cycle, but the binary it generates can only run on Intel architectures.

The situation gets even worse when C is being used as a backend language for some compiler. This is an example we are dealing with in this thesis — the SAC compiler generates C code on output, and during compilation we possess the knowledge that some of the code can be vectorised. How could we express this fact being sure that C compilers could pick it up?

We propose a set of C language extensions which provide full support for vectors and vector-operations independently from the architecture. The vector operations are being dispatched to the SIMD extensions of a CPU if they are present, or implemented with a number of scalar operations, otherwise. The key design criteria is to come up with as small as possible set of extensions that is large enough to:

1. benefit from the various existing SIMD instruction sets available on the market today, and
2. enable the programmer to conveniently express various SIMD applications.

Rather than inventing a completely new set of abstractions that suit these criteria, we build on the set of abstractions for SIMD operations that have been proposed in the context of OpenCL [73]. However, in contrast to OpenCL, we have integrated these operations into the GNU GCC compiler [44].

3.2 Motivation

As a running example we will consider a Move To Front (MTF) transformation which is used in modern compressing algorithms like BZIP2 [120]. This algorithm can be vectorised, however the pattern of the vectorisation is non-trivial and none of the auto-vectorisers we have tried out succeeded.

3.2.1 Move To Front (MTF) Algorithm

In order to improve compression algorithms which use the Burrows-Wheeler Transformation (BWT) [19] as an additional post-processing step one can use the Move To Front (MTF) transformation. After applying BWT we expect to get a string containing groups of repeating characters; for example '`bbbcccaaa`'. In order to decrease the entropy of the message and improve the efficiency of further Huffman encoding [59] we replace each symbol of the message with its index in the list of recently used symbols. The way the MTF works can be understood in Fig. 3.1. As

Original message	**Encoded message**	**Alphabet**
bbbcccaaa	∅	abcdefghijklmnopqrstuvwxyz
bbbcccaaa	1	a**b**cdefghijklmnopqrstuvwxyz
b**b**bcccaaa	1,0	**b**acdefghijklmnopqrstuvwxyz
bb**b**cccaaa	1,0,0	**b**acdefghijklmnopqrstuvwxyz
bbb**c**ccaaa	1,0,0,2	ba**c**defghijklmnopqrstuvwxyz
bbbc**c**caaa	1,0,0,2,0	**c**badefghijklmnopqrstuvwxyz
bbbcc**c**aaa	1,0,0,2,0,0	**c**badefghijklmnopqrstuvwxyz
bbbccc**a**aa	1,0,0,2,0,0,2	cb**a**defghijklmnopqrstuvwxyz
bbbccca**a**a	1,0,0,2,0,0,2,0	**a**cbdefghijklmnopqrstuvwxyz
bbbcccaa**a**	1,0,0,2,0,0,2,0,0	**a**cbdefghijklmnopqrstuvwxyz
∅	1,0,0,2,0,0,2,0,0	acbdefghijklmnopqrstuvwxyz

Figure 3.1: The MTF transformation steps

it can be seen, the encoding process at each step consists of finding an index of the symbol in the current state of the alphabet followed by the alphabet update, where the symbol is being moved to the first position of the alphabet.

The MTF is being used in BZIP2 compression, but while decompression the reverse version (unMTF) is used. This reverse version and its vectorisation will be used as a motivating example in this chapter. The decoding procedure is very similar to the encoding one. We will need the encoded message and the original alphabet used during encoding. We traverse the encoded message from left to right and replace every number it with the symbol in the alphabet at the position equal to the number, and, as during the encoding, move the symbol to the front of the alphabet. Fig 3.2 demonstrates this process.

Encoded message	**Decoded message**	**Alphabet**
1,0,0,2,0,0,2,0,0	∅	abcdefghijklmnopqrstuvwxyz
1,0,0,2,0,0,2,0,0	b	a**b**cdefghijklmnopqrstuvwxyz
1,**0**,0,2,0,0,2,0,0	bb	**b**acdefghijklmnopqrstuvwxyz
1,0,**0**,2,0,0,2,0,0	bbb	**b**acdefghijklmnopqrstuvwxyz
1,0,0,**2**,0,0,2,0,0	bbbc	ba**c**defghijklmnopqrstuvwxyz
1,0,0,2,**0**,0,2,0,0	bbbcc	**c**badefghijklmnopqrstuvwxyz
1,0,0,2,0,**0**,2,0,0	bbbccc	**c**badefghijklmnopqrstuvwxyz
1,0,0,2,0,0,**2**,0,0	bbbccca	cb**a**defghijklmnopqrstuvwxyz
1,0,0,2,0,0,2,**0**,0	bbbcccaa	**a**cbdefghijklmnopqrstuvwxyz
1,0,0,2,0,0,2,0,**0**	bbbcccaaa	**a**cbdefghijklmnopqrstuvwxyz
∅	bbbcccaaa	acbdefghijklmnopqrstuvwxyz

Figure 3.2: The unMTF transformation steps

The trivial implementation of a single step of unMTF is the following one:

```
char unMTF(char alphabet[256], int idx)
{
  char c = alphabet[idx];

  for (; idx > 0; idx--)
    alphabet[idx] = alphabet[idx-1];
    return alphabet[0] = c;
}
```

The variable `idx` is a number in the encoded message, and `alphabet` is the current state of the alphabet. The algorithm implemented as above is uses inefficient alphabet rotation — it has $O(N)$ worst case complexity, where N is the length of the alphabet. This rotation happens on every symbol of the encoded message so BZIP2 uses a more advanced implementation which makes it possible to reduces the worst case complexity to $O(\sqrt{N})$.

By dividing the alphabet into $\sqrt{N}$ chunks, the chunk that contains the symbol has to be updated, shifting the elements as in the above code, but all the chunks before can be updated by changing their first and last symbols. The implementation of this approach looks as follows:

```
#define N 4096
char alphabet[N];
short ptr[16] = {N-256, N-256+16, N-256+16*2, N-256+16*3, ... };

void rotate_segment(char *v, int idx)
{
  if (idx == 0)
    return;
  do
    v[idx] = v[idx-1];
  while (--idx);
}

void rearrange_alphabet ()
{
  int i, j, k = N-1;
  for (i = 15; i >= 0; i--)
    {
      for (j = 15; j >= 0; j--)
        alphabet[k] = alphabet[ptr[i] + j], k--;
      ptr[i] = k + 1;
    }
}

void unMTF(int idx)
{
  int i, q, r, c;

  if (idx == 0)
    return;

  q = idx / 16;
  r = idx % 16;
  c = alphabet[ptr[q] + r];

  rotate_segment(&alphabet[ptr[q]], r);

  ptr[q]++;
  for (i = q; i > 0; i--)
    {
      ptr[i]--;
      alphabet[ptr[i]] = alphabet[ptr[i-1]+15];
    }

  alphabet[--ptr[0]] = c;
  if (ptr[0] == 0)
    rearrange_alphabet ();

  return c;
}
```

The chunk q contains the symbol we are after (stored in c) at position r. The alphabet is stored in the variable alphabet which has bigger size than the length of the actual alphabet. Chunks are represented as parts of alphabet of a constant size. Shifting all the elements of a chunk one element to the right is achieved by decreasing the starting position of the chunk by one element and updating the element at this position. The variable ptr is an array of starting positions of the chunks in alphabet.

The chunk q is updated by moving the elements from r-1 to 0 one element to the right (this is done by rotate_segment). The element at position 0 in the chunk q is replaced with the last element from the chunk q-1. For all the chunks from q-1 to 0 we decrease its starting index by one, and put the last element of the previous chunk into the position 0 of the chunk we currently update. The first symbol of the very first chunk is replaced with the c.

As each unMTF step potentially moves chunks to the left, eventually the first chunk will reach the first position in alphabet, in which case the alphabet-array has to be rearranged by putting all the chunks at the end of the array; this is done using rearrange_alphabet function.

In BZIP2 the length of the alphabet is 256 which after dividing into chunks gives us 16 chunks each of which is 16 characters long. Conveniently enough standard SIMD registers these days are 128-bit long which is exactly one chunk. This means that rearrange_alphabet can move chunks with two vector instructions rather than with 16 scalar ones. The fact that most of the SIMD architectures support permutations within a vector gives us a chance to implement a vectorised version of rotate_segment. Now, how can the desired vectorisation be expressed?

Auto-vectorisers we tried out (GCC, ICC) did not consider any of the functions suitable for the vectorisation. There are several reasons for that: first of all, the rotate_segment signature does not contain any information about the maximal values of idx, so a compiler can only deduce this information from the calling context. Secondly, a compiler needs to apply a cost model to show that the transformation is beneficial, but this is not an easy task as a potential vectorisation may increase the number of instructions which affects an instruction pipeline; or add conditions which affect branch prediction; or change the memory access; or similar transformations that may harm program performance. Without the knowledge that a particular function is a hot-spot, a compiler can take a decision not to vectorise a function even if it is possible in theory.

In order to express rotate_segment explicitly in a portable SIMD way we have to have an interface for vector permutation. In GCC it was impossible before we added this to version 4.7. Alternatively one can express a permutation using inline assembly, but disregard the fact it is non-portable, even for one architecture one may end-up creating several variants of the code. For example: Intel SSE3 has a PSHUFB instruction which does a byte-level permutation; any lower version of SSE supports 32-bit elements permutations only which require a programmer to come-up with vector shifting and masking scheme which is less efficient and in case the architecture uses AVX another version of the code is needed.

Vectorisation of `rearrange_alphabet` can be done in a portable way starting from GCC v3.2, declaring a variable of vector type and for every chunk loading it to the variable and storing it back into the memory. The code for the function looks as following:

```
#define vector(elcount, type)  \
__attribute__((vector_size((elcount)*sizeof(type)))) type
typedef char __attribute__ ((vector_size (16), aligned (4))) xchar;
#define unaligned(x)  ((xchar *)x)
void rearrange_alphabet ()
{
  int i;
  for (i = 15; i >= 0; i--)
    {
      vector (16, char) vec = *unaligned (&alphabet[ptr[i]]);
      short idx = N-256+16*i;

      *(vector (16, char) *)&alphabet[idx] = vec;
      ptr[i] = idx;
    }
 }
```

Some architectures, for example Intel, differentiate aligned and unaligned vector loads providing two separate instructions for this purpose. In the code above, we have to take care of the cases when a vector-assignment accesses unaligned memory. In order to inform the compiler, we mark potentially unaligned memory by converting it to the vector type with minimal alignment.

3.3 C vector extensions

Turns out that vector permutation is not the only missing feature which makes the vector programming framework incomplete. Analysing common operations in different SIMD accelerators and combining them with scalar operations available in C, we managed to identify the set of operations which is complete enough to cover a large part of the commonly available SIMD instructions. We base our framework on GCC, and the following features were missing:

1. Vector indexing in the same style as arrays indexed;
2. Vector element-wise and whole vector shifting. Element-wise shifting should take special care when a target supports a vector/scalar operand combination.
3. Scalar/vector and vector/scalar operations like `1 + {2, 3}`[1].
4. Vector comparison using standard comparison operations: `>`, `<`, `==`, `>=`, `<=`, `!=`;
5. Vector permutation;

The missing functionality has been implemented by the author of the thesis and is now an official part of GCC since version 4.7. All the changes have been implemented as a series of patches that went through a reviewing process via the

[1]Please note that this is not a conceptual problem, but rather syntactic sugar. One can always broadcast the scalar value to the elements of a vector.

gcc-patches@gcc.gnu.org mailing list and have been approved by the maintainers of the relevant part of GCC.

Further down we make an overview of GCC explicit SIMD framework in its current state and discuss the parts which have been added by the author of the thesis.

3.3.1 Vector types

The first step towards explicit SIMD programming is to declare a vector type which can be done using a notion of attributes. Consider the following variable declaration example:

```
int __attribute__ ((vector_size (16))) var;
```

The `int` type specifies the base type of the vector, and the attribute specifies the length of the vector type measured in bytes. In the declaration above, given that int is a 32-bit type we define a vector of 4 ints. The basic type of the vector can be both signed and unsigned integer types: `char`, `short`, `int`, `long` and `long long`. In addition float and double can be used to define a floating-point vector types. The size of the vector type can be any number which is a power of two. Vector types are treated in the same way as C base types, which means that one can create variables of vector types, create pointers to vector types and use `sizeof` operator, use vector type when declaring a function argument or function return type, make type-casts. In the latter case of casting from one vector type to another, one must make sure the types are of the same size.

3.3.2 Vector value

Defining a vector variable one can assign a constant value using an array notation. Consider the following example:

```
#define vector(elcount, type) \
__attribute__((vector_size((elcount)*sizeof(type)))) type

vector (4, float) pp = {3., .1, .4, .1};
```

Here we declare a variable `pp` of vector type of 4 floats and initialize the variable with values `3.0`, `0.1`, `0.4` and `0.1`. If the initialisation vector contains less elements than the type, the missing elements will be implicitly filled with zeroes without a warning being produced. For example:

```
vector (8, short) v = {1, 2, 3};
```

In this case the compiler would initialize `v` with `{1, 2, 3, 0, 0, 0, 0, 0}`.

Constant vector-values are defined in the same way as when initialising a variable, but with an explicit type-cast. For example:

```
vector (4, int) a, b;
a = b + (vector (4, int)){1, 2, 3, 4};
```

3.3.3 Vector operations

Vector shifts were implemented by A.Š.
optimising cases when vector/scalar shifts are
supported by the underlying hardware.

Vector types can be used within a subset of normal C operations. Currently GCC allows using the following operations on vector types: `+`, `-`, `*`, `/`, `%`, `unary minus`, `>>`, `<<`, `^`, `|`, `&`, `~`. All the binary operations perform an element-wise operation on vector elements. For example:

```
vector (4, int) a, b, c;
a = b + c;
```

This code for each of the four elements of `b` will add the corresponding four elements of `c` and store the result in `a`.

Assigning expression of vector types, it is allowed to use a short form of the binary operation like `+=`, `-=`, etc. The semantics of the operation is going to be the same as in the scalar case. For example:

```
vector (4, int) a, b, c;
a += b;          /* a = a + b;   */
b <<= c;         /* b = b << c;  */
```

3.3.4 Mixed vector/scalar operations

This was implemented by A.Š.

Following the OpenCL conventions we also allow both scalar/vector and vector/scalar variants of binary expression. In that case the scalar is transformed to the vector of corresponding type where all the elements are equal to the given scalar. Consider the following example:

```
vector (4, float) a, b, c;

a = b + 1.;      /* a = b + {1., 1., 1., 1.}  */
a = 1. + b;      /* a = {1., 1., 1., 1.} + b  */
```

Note that the transformation will happen only when the scalar can be safely transformed to the vector type. For example, the following code would produce an error, as converting `long` to `int` includes a truncation.

```
vector (4, int) a, b;
long l;

a = b + l;       /* Error, cannot convert long to int.  */
```

When shifting is used on the vector types, we do not follow OpenCL in the case when right-hand side of the shifting expression is greater than $\log_2 N$. Following the C standard of scalar shifting we leave this situation undefined. This choice was made deliberately to avoid runtime masking of the right-hand side.

3.3.5 Vector indexing

Vector indexing was implemented by A.Š.
together with bound checking flag.

Vectors can be indexed in the same way as if they were arrays with the same number of elements and base-type. Out of bound access invokes undefined behaviour at runtime, however, the warnings can be enabled using `-Warray-bounds`. Consider the following example:

```
vector (4, int) a = {1, 2, 3, 4};
int i = 3, sum = a[0] + a[1] + a[2] + a[i];
```

3.3.6 Vector comparisons

This was implemented by A.Š.
including code generation for
x86 architectures.

Vector comparison is supported within the standard comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`. Comparison operands can be either both integer type or both real type. Comparisons between integer-type vectors and real-type vectors are not supported. The result of vector comparison is a vector of the same width and number of elements as the comparison operands with a signed integer base type. Vectors are compared element-wise producing `0` when comparison is false and `-1` (constant of the appropriate type where all the bits are set) otherwise. Consider the following example:

```
vector (4, int) a = {1,2,3,4};
vector (4, int) b = {3,2,1,4};
vector (4, int) c;

c = a > b;      /* The result would be {0, 0,-1, 0}  */
c = a == b;     /* The result would be {0,-1, 0,-1}  */
```

3.3.7 Vector permutation

This was implemented by A.Š.
including code generation for
x86 architectures.

Vector shuffling is available using two and three argument `__builtin_shuffle` functions: `__builtin_shuffle (vec, mask)` and `__builtin_shuffle (vec0, vec1, mask)`. Both functions construct a permutation of elements from one or two vectors and return a vector of the same type as the input vector (s). The mask is an integral vector with the same width (W) and element count (N) as the output vector.

The elements of the input vectors are numbered in memory ordering of `vec0` beginning at 0 and `vec1` beginning at N. The elements of mask are considered modulo N in the single-operand case and modulo $2 * N$ in the two-operand case. Consider the following example:

```
vector (4, int) a = {1,2,3,4};
vector (4, int) b = {5,6,7,8};
vector (4, int) mask1 = {0,1,1,3};
vector (4, int) mask2 = {0,4,2,5};
vector (4, int) res;

res = __builtin_shuffle (a, mask1);     /* res is {1,2,2,4}  */
res = __builtin_shuffle (a, b, mask2);  /* res is {1,5,3,6}  */
```

3.3.8 Vector operation performance

This was implemented by A.Š.

As vector operations might be implemented with scalars or vectors of a smaller type, for the convenience of a programmer we introduce a flag called:

`-Wvector-operation-performance`

which emits warnings in case this happens. This is a useful tool to identify performance problems in the code or unimplemented features of GCC explicit vector extensions

3.3.9 Still missing

Currently GCC (starting from version 4.7) includes all the missing features except whole vector-shifting and does not support the following operations in vector mode: `++`, `--`, `!`, `&&`, `||`.

3.4 Case-study

Getting back to our running example, let us study the effects of vectorisation and evaluate performance. We are not going to make extensive performance measurements, because the figures mainly depend on the quality of a particular code-generator. The main purpose is to demonstrate that using GCC vector extensions we can address complicated patterns producing code which is portable across all the platforms supported by the compiler and which can efficiently use SIMD accelerators in case they are present.

In order to implement `rotate_segment` in a vectorised way we would load a 16-symbol chunk into a vector register, perform a permutation in-place and store it back into the memory. All the chunks are potentially unaligned, so we mark it in the same way as in `rearrange_alphabet`. The code looks as follows:

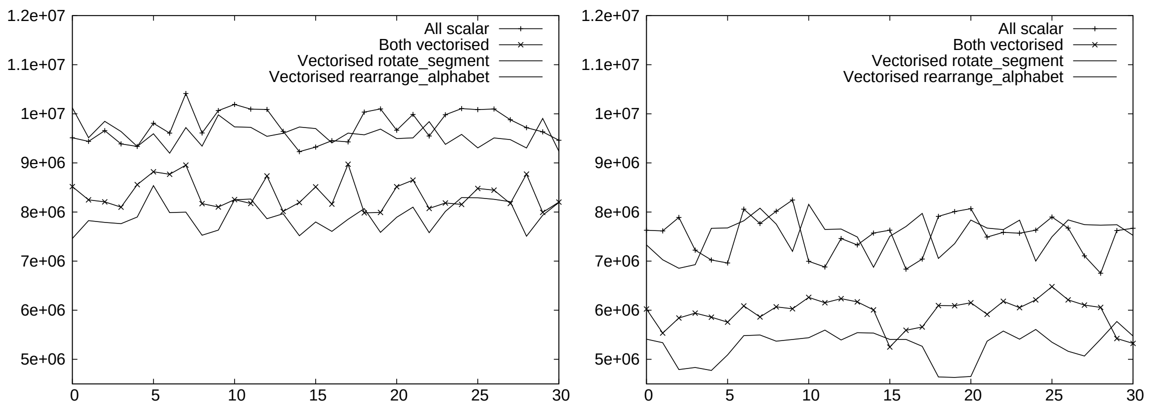
```
const vector (16, char) perms[16] = {
  (vector (16, char)){0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15},
  (vector (16, char)){1,0,2,3,4,5,6,7,8,9,10,11,12,13,14,15},
  (vector (16, char)){2,0,1,3,4,5,6,7,8,9,10,11,12,13,14,15},
  /* ... */
};

void rotate_segment (char *v, int idx)
{
  vector (16, char) t, vec;

  if (idx == 0)
    return;

  t = *unaligned (v);
  vec = __builtin_shuffle (t, perms[idx]);
  *unaligned (v) = vec;
}
```

As all the permutation masks are static, the compiler can perform an optimisation of each particular permutation. For example, it can replace the case when `idx` is one with something like: `swap (v[0], v[1])`. Note that in the OpenCL framework permutation is a library function call and the above optimisation cannot be achieved unless link time optimisation is used, in which case the library must be compiled with special flags.



(a) Intel Core2 duo, 2GB RAM, 4MB cache (b) Intel Core i5, 4GB RAM, 6MB cache

Figure 3.3: Vectorised and non-vectorised versions of unMTF measured in clock-cycles. All the measurements were repeated 30 times sequentially using an output of a previous run as an input for the next one. The first input is a file of 10^5 random characters.

Consider Fig. 3.3 which shows how the vectorised and non-vectorised versions of `rotate_segment` and `rearrange_alphabet` impact performance of unMTF. The base-line of the experiment is a fully scalar implementation. We observe the best performance improvement on both architectures when `rotate_segment` is vectorised and `rearrange_alphabet` is not: we have about 20% and 30% speed-ups on these architectures. We observe a negative impact from vectorising `rearrange_alphabet`. In order to explain this we have to realise that unaligned move on Intel is expensive and it also creates additional pressure on the memory. The operation itself happens quite rarely (once per length of the alphabet-array) so the initialisation overheads are bigger than the performance gain.

As another experiment we include the newly implemented function in the original implementation of BZIP2 in order to see the impact of this function on the overall decompression process. As input data we use an encoded 347MB video-file and as a measurement unix `time` command taking user's time. On a Intel Core2 duo we observed 18% speed-up (36.6s vs 30.2s) but on the Core i5, the run-time difference was in the order of the measurement error (26s both). As BZIP2 is a pipe-line we can see that the improvement of a single part may not affect the whole process; hence unMTF is not a hot-spot on Core i5 processor. Identifying a new hot-spot of the algorithm requires an analysis which is outside of the scope of this chapter.

We have demonstrated how easily one can experiment with the potential benefits of using SIMD extensions. All the results on both architectures were obtained from one and the same source code which would also work on any architecture supported by GCC.

3.5 Related work

In this section we will make an overview of existing concepts and approaches which allow one to exploit the SIMD extensions of a CPU. As a general remark, there exists a number of attempts to unify diverse SIMD instructions and come up with a portable level of abstraction. Our main claim here is that in order to maximise optimisation potential of programs and share a large portion of the code used in auto-vectorisers, the abstraction layer has to be implemented in a compiler as language extensions, not as a library or a novel programming language.

3.5.1 New languages e.g. ISPC

The language demonstrates quite good performance, however, we believe that it is not the best possible way to program SIMD extensions explicitly. The level of abstraction is too high which means that a lot of decisions is taken by the compiler without a programmer being able to control them. Expressing code as kernels, similarly to OpenCL or CUDA, opens a potential to reuse the same code not only on SIMD architectures, but at the same time, requires significant program rewrites. The set of supported SIMD architectures for the time being is limited to SSE2, SSE4, AVX, AVX2, and Xeon Phi.

3.5.2 Library-based solutions

The main drawback of any library approach is that dispatching from the API down to intrinsics or assembly has to be implemented at the level of the library. This means that parts of the compiler has to be reimplemented at the level of the library, and, for example, auto-vectorisers will not be able to benefit directly from the library. Or, if a compiler cannot optimise across intrinsics, then this functionality has to be implemented in the library. Ideally we would like to see a symbiosis between auto-vectorisers and explicit SIMD. One practical benefit of the library approach is an ability to make experiments very quickly, but in general it seems that library solutions do not scale.

3.5.3 Intrinsics to intrinsics mapping

That could have been a valid approach, however the variety of hardware changes quite rapidly, and if we will start to create mappings, we would need to create every-to-every instruction set mappings, or chose a canonical one. Both of the options are not desirable, not to mention, creating such a mapping requires a lot of work, for example, the mapping described in [152] is a 1MB file, and does not include AVX extensions.

3.5.4 C++ standard proposal

Independently from our work, A. Naumann et. al indicated the importance of having vectorisation as a language feature in [8]. The line of reasoning is very similar to ours — vectorisation is important, and it cannot realistically be automated or fitted into existing parallel models. They demonstrate a practical micro-kernel used in CERN and use the same arguments, i.e. more tightly integrated optimisations and implementation reusability, when discussing why they think vectorisation should be a part of the language.

3.5.5 Automatic vectorisation

The downside of automatic vectorisation is a lack of opportunities to influence the decision of the vectoriser. The number of supported patterns is always limited, and in the cases of non-trivial data-dependencies the vectoriser would give-up. In order to get the best performance from an auto-vectoriser in case of floating-point operations one has to specify flags that violate the IEEE implementation of floating point [104]. As an example we can consider the case of horizontal sums:

```
float *array, result;

for (i = 0; i < N; i++)
  result += array[i];
```

```
float *array, result;
float_vec reg;

/* Assume N % 4 == 0 */
for (i = 0; i < N; i += 4)
  reg += *(float_vec *)&array[i];

result = reg[0] + reg[1]
         + reg[2] + reg[3];
```

According to the IEEE floating point standard, the order of operations can change the result; hence the above optimisation is illegal. In order to legalise it in GCC, one needs to specify the `-ffast-math` flag when compiling and it is impossible to use it on a given loop only. It means that in order to make auto-vectoriser perform the optimisation, a programmer has to switch a flag potentially violating all the floating-point operations.

The auto-vectoriser cannot properly handle the loops with the control-flow, e.g. conditions, `goto`s, and uncountable loops, e.g. `while (*x != NULL)`.

3.5.6 Virtual instruction set

The LLVA approach provides a portable standard for SIMD operations. However, this approach raises several practical and theoretical questions. Practically, the architecture exists only as a prototype with implemented translators for several ISA-s. This means that in order to integrate LLVA in any existing compiler, we will have to provide a translation from the intermediate language of a compiler to the LLVA. Assuming that we did that, we will have to implement the translators for all the targets we want to support. Keep in mind that LLVA provides instructions only for vector operations, this means that all the non-vector operations have to be integrated into the representation as well. Assuming that we did that, we come to the point when we will have to instruct our auto-vectoriser and possibly other optimisations to generate the code using LLVA. How can we estimate the cost of the operation, if we do not know the target architecture?

As several architecture classes are supported within LLVA, and there are mechanisms allowing careful tuning for each processor class, how efficient would it be to run an LLVA code tuned for class A on class B?

3.5.7 OpenCL

In terms of portability our approach is very similar to OpenCL and we borrow the syntax of SIMD operations, however; there are a number of important distinctions. The intent of OpenCL is very different from ours. OpenCL operates with a large-scale problem trying to target many different diverse architectures, e.g. SPMDs, GPUs; where our approach solves a single issue. The OpenCL C programming language is based on ISO/IEC 9899:1999 C language standard (a.k.a C99) [63], but it also introduces a number of restrictions. Most importantly, OpenCL tries to cover all the undefined or ambiguous cases of the C99 standard. For example, basic types, like `int`, `char` and `long` get a fixed size; C99 in this cases fixes only the relation of the type-sizes i.e `char` ≤ `int` ≤ `long`. Defining bit-shift operations `e1 << e2`, OpenCL states that only the lower $\log_2 N$ bits of `e2` will be used during the operation; C99 in this case states that if `e2 >` $\log_2 N$, the result is undefined.

Arguably, the restrictions of OpenCL increase the portability of programs but at the same time they remove backward-compatibility with ISO C code. Practically it means that existing C code may not work within the OpenCL compiler.

Technically, OpenCL provides a set of libraries, and header files, but the actual compilation is done by a C compiler of the users choice. This is a key difference from the approach we are taking, as we implement SIMD operations as an integral part of the C language; hence as a part of the compiler. Decoupling a framework from the compiler gives you freedom when you choose a compiler, but in terms of SIMD operations we see the following problems with this approach:

1. In order to define a SIMD vector OpenCL provides the `type`*`n`* construction, where n can be 2,3,4,8 or 16 and the `type` is a basic scalar type, e.g. `char`,

`int`, `float`, etc. As there is no way to override selection operator `[]` in C, OpenCL introduces a new scheme for enumerating vector components introducing the notion of `lo, hi` components `x, y, z, w`, etc. The vector type is mapped to the hardware-specific SIMD vector type or static array in case SIMD accelerators are not present within the architecture. Such a design makes it complicated to support vectors of arbitrary length, as each `type`*n* is defined as a new structure and the chosen indexing scheme leads to a combinatorial explosion. Also, each time the length of vector register doubles, a standard correction is required. For example, currently, it is impossible to define a `char32` type, however, it is supported by Intel AVX.

   Our approach allows one to define a vector of the arbitrary length, where the length is a power of two. To index elements we use a standard selection `[]` operator and during compilation vector operations are compiled to the longest vectors supported by the architecture.

2. Basic vector operations like arithmetic, comparison, shuffling are, whether aliases to the intrinsic functions or external functions, defined in the library. From the performance point of view both cases are harmful as they decrease the chance for optimisations. Library function calls prohibit[2] even simple constant propagation; intrinsic functions normally do not participate in the optimisation cycle. If vector operations would be inlined, the compiler can generate better code with respect to the pipelining and register pressure.

3. OpenCL SDKs are mainly closed-source products which are released for some combination of hardware architecture and operating system. This means that there is a chance that all these products perform slightly differently. Our approach does not solve this problem fully, as code-generators are unique for every hardware architecture as well; however, most of the optimisations happen in the middle-end. Also, as GCC compiler is an open-sourced product, anyone has a chance to identify the reason of the undesired behaviour by means of code inspection or debugging.

3.6 Extended standard

In this section we sketch a proposal to extend the existing C standard to include the work we have implemented in terms of GCC. We start with the set of general assumptions we had in mind.

3.6.1 General principle

- Vectors can be of size 2^n bytes, $n > 0$. We want to make vector types extensible for future architectures. The 2^n property comes from observing sizes of the

[2]Unless link-time optimisations is considered.

SIMD registers and considering the sizes of the built-in types. Implementation-wise, operations on large vectors can be implemented using smaller vectors or scalars.

- We aim at supporting the intersection of the various instruction sets in the C language. For instructions outside the intersection, we explore if they can be expressed with a combination of the instructions from the intersection. For example, some of the architectures define multiply-add as a single instruction. We are not going to include multiply-add as a C primitive, as there is a way to express it with a combination of multiplication and addition. On the other hand it would be much more difficult to express the rotation operation without the designated primitive operation.

- We assume that compilers can perform complicated pattern matching to recognise the cases when several operations can be replaced by a single one on given hardware.

- Large vectors should be implemented with the largest hardware vector available on a given target or by scalar operations. Exact choices can be dictated by a cost model used for auto-vectorisation.

- A compiler shall provide a flag which would report the cases when a vector operation was implemented by scalar operations. Alternatively a user would have to analyse a final output, but she will not be able to reason if the generated code is optimal, or the fallback implementation has been used.

- The math library has to be extended by the vector variants of all the existing operations. Semantically, an operation on a vector type is an application of the scalar operation to all vector components. In this case we would be able to support hardware implemented mathematical functions for SIMD vectors.

- We should support estimated mathematical operations. Some of the instruction sets define estimated mathematical operations (like estimated square root) on vector types. These instructions are normally faster, but are less precise. In order to support them we propose to extend the math library by introducing estimated variants for all the operations. Semantically, if there is no implementation for estimated operation, we alias estimated operation to the normal one, or map it to hardware instruction otherwise.

Now we are going to describe extensions to the standard, identify the sections of ISO/IEC 9899:2011 [64] which would need altering and give a rationale regarding each operation.

3.6.2 Vector types

We introduce a notion of a new derived type [64, 6.2.5] which we are going to call a vector type. We do this by duplicating a passage about an array type with some added restrictions:

> A *vector type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the element type. The element type shall be complete whenever the vector type is specified. Element types must be evaluated either to integral type or a floating point type. Vector types are characterized by their element type and by the number of elements in the vector. The number of elements must be a compile time constant. The length of the vector type (the number of elements multiplied by the size of the element type) must be a power of two. A vector type is said to be derived from its element type, and if its element type is T, the number of elements is N, the vector type is sometimes called "vector of N T".

A syntax for vector types has to be chosen and should be described in the new subsection of [64, 6.7.2].

> vector-type:
>   **vector** **(** vector-size **,** type-specifier **)**
> vector-size:
>   constant-expression
>
> A vector-size multiplied by the size of type-specifier must evaluate to 2^k, where k is an integer and $k > 0$.
>
> An element type of the vector type shall be either integer or floating point.

The way we define a vector type is very similar to OpenCL, however we allow arbitrary long vectors and we are not restricting element types to the base types only allowing to use enums or user-defined types (assuming they can be evaluated to floats or integral types).

3.6.3 Declaration and initialisation

Now we need to define how the values of the new type can be constructed. We are going to borrow the syntax described in the Compound literals section [64, 6.5.2.5], allowing one to initialise an object of a vector type as if it were an array of known size. For example:

```
vector (4, int) x = {1, 2, 3, 4};
vector (4, int) y = {1};
f ((vector (4, int)){5,6,7,8});
```

According to [64, 6.5.2.5], [64, 6.7.9] this syntax should be supported automatically, as vector types are complete.

If we want to allow the following syntax for vectors:

```
vector (4, int) z = {[2]=42};
```

we would have to adjust the Initialization section [64, 6.7.9] paragraph 6 like this[3]:

> The type of the entity to be initialized shall be an array of unknown size or a complete object type that is not a variable length array type.
>
> If a designator has the form
>
> [constant-expression]
>
> then the current object (defined below) shall have array or vector type and the expression shall be an integer constant expression. If the array is of unknown size, any nonnegative value is valid.

3.6.4 Vector subscript

Vectors can be subscripted as if the vector were an array with the same number of elements and base type. Formal addition would require a new subsection in [64, 6.5.2]:

> Vector subscripting.
>
> Constraints.
>
> The first expression shall have type "vector of basetype *type*", the second expression (inside square brackets) shall have integer type and the result of type "*type*".
>
> Semantics
>
> A postfix expression followed by an expression in square brackets `[]` is a subscripted designation of an element of a vector object. The definition of the subscript operator `[]` is that `E1[E2]` designates the E2-th element of E1 (counting from zero).

3.6.5 Arithmetic operations

The vector operations we are going to support can be divided into 4 groups:

- Arithmetic operations (`+, -, *, /, %, unary +, unary -`);
- Bitwise operations (`&, |, ^, ~`);
- Comparison operations (`>, <, ==, <=, >= !=`);
- Vector shifts (`>>, <<`); and
- Vector shuffle (`shuffle (v1, v2, mask)`).

[3]Here and further we mark introduced changes in red.

All the binary operations mentioned above except `shuffle` are operating on vector of floating-point types or integral types of the same signedness, with the same number of elements. The semantics of the vector operation is component-wise application of the according operation on all the elements of the vector.

Any vector comparison operation returns a mask (vector of signed integer), where *false* is represented with value 0, and *true* with value -1 (all bits set) and the size of the mask element type is the same as the size of the operand's element.

Vector shuffle is defined similarly to the OpenCL shuffle and shuffle2 functions:

> Vector shuffling is available using functions `shuffle (vec, mask)` and `shuffle (vec0, vec1, mask)`. Both functions construct a permutation of elements from one or two vectors and return a vector of the same type as the input vector (s). The mask is an integral vector with the same width (W) and element count (N) as the output vector.
>
> The elements of the input vectors are numbered in memory ordering of `vec0` beginning at 0 and `vec1` beginning at N. The elements of the mask are considered modulo N in the single-operand case and modulo $2N$ in the two-operand case.
>
> Consider the following example,

```
typedef vector (4, int) v4si;

v4si a = {1,2,3,4};
v4si b = {5,6,7,8};
v4si mask1 = {0,1,1,3};
v4si mask2 = {0,4,2,5};
v4si res;

/* res is {1,2,2,4}  */
res = shuffle (a, mask1);

/* res is {1,5,3,6}  */
res = shuffle (a, b, mask2);
```

In order to include changes to `+,-` in the standard, the following fixes are needed in [64, 6.5.6]:

> For scalar addition, either both operands shall have arithmetic type, or one operand shall be a pointer to a complete object type and the other shall have integer type. (Incrementing is equivalent to adding 1.) For vector addition, either both operands shall have arithmetic vector type with equal number of elements and signedness, or one operand shall have pointer vector type and the other shall have integer vector type, assuming that the number of elements in both operands is the same.

Fixes for all the other operations are similar.

3.6.6 Future extensions

For conversions we do not need to add anything new. Initialisation by means of compound literals (6.5.2.5) would automatically propagate standard conversions to the vector components and cast operation (6.5.4) would take care of reinterpreting bytes of data as a given type.

3.6.7 Things to consider

Here is a list of features that are not in the proposed standard but might be added later:

- Horizontal reduction operations. A number of architectures e.g. SSE2, SSE3 support reductions over vectors (often called horizontal operations). The way we propose to deal with them is by recognising patterns in the program. That gives a good potential for using a target architecture more effectively, as the patterns can be found in ordinary programs and replaced with corresponding vector operations.
- Printf/Scanf extensions. AltiVec proposes convenient extensions for printing out vectors and reading in vectors. Currently one would need to implement vector IO by hand.
- Mixed scalar/vector operations. OpenCL supports arithmetic operations where one operand is a vector and another is a scalar, in which case the scalar operand is being promoted to a vector. That seems to be pure syntactic sugar, or is there a good usecase?
- Scatter/Gather operations. Some of the ISAs allow loads and stores from/to non-contiguous memory. Normally they are not very efficient, so it might be a good idea to discourage people from using them.
- Predication. Newest architectures e.g. MIC support masked execution of standard instructions. For example, we may add only those elements of two vectors that are marked as true in the corresponding mask. This is usually called "predicated instructions" and is used in the branches of vectorised conditionals.

3.7 Towards the existing instruction sets

In this section we study the coverage of the existing SIMD instructions. In other words, does the proposed standard give a handle on existing SIMD instructions, how it can be expressed and how much work compilers would have to do.

The number of SIMD-capable architectures is quite large, however most of them became rare and esoteric. In this chapter we to concentrate on those sets which are, to our opinion, to be found on modern CPUs. These are Intel's (MMX, SSE, SSEx, AVX), IBM's Altivec and ARM's NEON.

We are going to skip instructions that can be trivially obtained from the C operations. For example, all the instruction sets under consideration allow vector addition, so we will not pay extra attention to the addition on vectors. Instead we are going to consider those instructions that are available but cannot be directly mapped from the C operation. For those we propose a compilation scheme, if we can, or admit that they are going to stay unavailable to a programmer.

3.7.1 Intel

This overview is based on [60] volume 1, Chapters 8–14 and volume 2, Chapters 3,4. In general, for all the Intel extended instructions, we omit support for saturated arithmetics. There is no support for saturated scalar operations in C, so getting vector saturated operations only would make a standard incomplete.

MMX

MMX technology operates on 64-bit vectors and supports only integer types of size 1,2 or 4 bytes. It supports arithmetic and logical instructions, shifts, comparisons and data movement instructions. Here is a list of those instructions that cannot be directly mapped into the proposed operations:

— Complex operations —

- Fused instruction `PMADD` multiplies elements and adds adjacent pairs. It is important to mention that the resulting vector changes the size (and the number) of its components. In general we express those cases using vector construction. For example, the following pattern could be recognised:
```
vector (4, int16) a, b;
vector (2, int32) c = {a[3] * b[3] + a[2] * b[2],
                       a[1] * b[1] + a[0] * b[0]};
```

- `ANDN` is a bitwise `and` with a first operator negated. This can be pattern-matched from `~a & b`. Please note, that there is no support for bitwise `not`, so `~x` could be implemented as `x ^ broadcast(-1)` — `xor` with all bits set.

— Element reordering —

- Unpacking instructions interleave lower or higher parts of two operands like this: PUNPCKHW$([0,1,2,3],[4,5,6,7])$ = $[2,6,3,7]$. This can be expressed via `shuffle` operation.

— Inapplicable —

- Logical left and right shifts (`PSLLx`, `PSRLx`) are supported as well as arithmetic right shifts: (`PSRAx`). Arithmetic shifts are not directly expressible in case of scalar operations in C, so we will not support it in vector mode either.

- `PMULLW` keeps higher bits of integer multiplication. This instruction would be unavailable, as there is no universal way to do that in C for scalars. Normally one would cast operands to a wider type, perform a multiplication there, and take the higher bits. Unfortunately that will not work for the highest type.

SSE

Intel MMX technology introduced single-instruction multiple-data (SIMD) capability into the IA-32 architecture, with the 64-bit MMX registers, 64-bit packed integer data types, and instructions that allowed SIMD operations to be performed on packed integers. SSE extensions expand the SIMD execution model by adding facilities for handling packed and scalar single-precision floating-point values contained in 128-bit registers. Here is a list of non-trivial instructions:

— Operations on parts of a vector —

- Operations on the lowest value, preserving other values, like `ADDSS`, these can be recognised from the assignment to the lowest component. For example: `b[0] = b[0] + a[0]`. Other `SS`-prefixed instructions can be handled similarly.

— Mathematical operations —

- Reciprocal, square root, reciprocal of square root on floating point data, these should be available through the expansion of math library functions. Please note, that reciprocal and reciprocal of square root are approximate, where the square root is precise.

— Advanced arithmetic —

- Min and Max on packed data. Recognise the following pattern:
  ```
  m = (a > b);  m & a | ~m & b
  ```

- Average (`PAVGx`). Recognise the following pattern:
  ```
  res = (a + b) / broadcast (2);
  ```

— Element reordering —

- Shuffle and Unpack instructions are available via shuffle.

— Conversion operations —

- Conversions are available via typecasts, like
  ```
  res = (vector (4, float)){1,2,3,4};
  ```

— Complex operations —

- The PSADBW (compute sum of absolute differences). In principle, this instruction can be recognised via patterns, but the pattern becomes too big. So it might be worthwhile to avoid its recognition.

— Inapplicable —

- The `PMOVMSKB` (move byte mask) instruction creates an 8-bit mask from the packed byte integers in an MMX register and stores the result in the low byte of a general-purpose register. We consider this instruction to be too architecture-specific, hence it will not be available from the C level.

SSE2

SSE2 introduces operations on double-precision floating point values.

— Conversion operations —

- Conversions are available via recognising vector reconstructions. For example, for `CVTPD2PS` the following pattern can be used:

```
vector (2, double) d = ...
vector (4, float) x = {(float)d[0], (float)d[1]}
```

— Element reordering —

- Shuffles and unpacks are available via the shuffle operation.

— Inapplicable —

- The `MOVMSKPD` (move packed double-precision floating-point mask) instruction extracts the sign bit of each of the two packed double-precision floating-point numbers in an XMM register and saves them in a general-purpose register. This 2-bit value can then be used as a condition to perform branching. This is architecture-specific so will not be supported directly.

SSE3, SSSE3, SSE4

— Advanced arithmetic —

- `ADDSUBPx` mixed addition and subtraction, can be recognised via the vector construction:

```
{a[0]-b[0], a[1]+b[1], a[2]-b[2], a[3]+b[3]}
```

- `HADDPx` horizontal addition and subtraction can be recognised via the vector construction:

```
{a[0]+a[1], a[2]+a[3], b[0]+b[1], b[2]+b[3]}
```

  Other horizontal additions/subtractions can be expressed in a similar fashion.

— Complex operations —

- `PSIGNBx` negates each negative element. The instruction can be recognised via the following pattern:

```
m = a < broadcast (0);
 res = (a & m) * broadcast (-1) | a & ~m;
```

- Dot products allow one to chose which partial multiplications to add and where to put a result by using vector masks. The operation can be recognised via:

```
p = a * b; /* dot product of all components */
t = p[0] + p[3]; /* any combination here */
res = {t, 0, t, t}; /* any duplictation here */
```

— Element reordering —

- `MOVxDUP` instructions get the higher or lower part of a vector and duplicate its elements. For example, MOVSHDUP$(a, [1, 2, 3, 4]) = [3, 3, 4, 4]$. This can be expressed via `shuffle` operation. `MOVSHDUP` on vectors `a` and `b` can be expressed as `a = shuffle (b, (vint){1,1,3,3}`.
- `PALIGNR` (Packed Align Right) instruction concatenates two vectors and shifts the result right by the number specified via the third operand. Such a pattern is expressible via shuffle operation on char vectors.
- Blending instructions implement conditional selection of elements based on mask. These can be implemented via `a & m | b & ~m`

— Mathematical operations —

- Integer absolute values. The function `abs` that can be found in `stdlib.h` can be extended to recognise a vector argument.
- Round instructions can be accessed via an extended math library.

— Comparison operation —

- `PTEST` sets the zero flag of a CPU with a bitwise and of all vector elements. It can be recognised as a reduction of scalar `&&` operation over all the elements of the vector with further branching. For example:

```
if (m[0] && m[1] && ...)
  {
    ...
  }
```

— Inapplicable —

- `MPSADBW` and `PHMINPOSUW` are special instruction designed for some HD codecs. We will not recognise them due to the complexity of the pattern.
- `CRC32` is used to accumulate values while implementing Cyclic Redundancy Check (CRC) with a pre-defined polynomial. This is application-specific instruction, it will not be supported.
- `PMULHRSW` (Packed Multiply High with Round and Scale) — unavailable, as it is intended to operate on a specific kinds of fixed point numbers.
- String search routines will not be available. These instructions return the index of the element, where a certain condition holds — there is no simple way to express that in C.

AVX

— Data movement —

- Broadcast instructions can be recognised via vector constructions, for example: `(vint){x,x,x,x}`.
- Masked moves can be recognised via fusing mask and move operations, like: `a = a & ~m | b & m` where `m` choses which elements of `b` to be moved.
- Scatter/Gather instructions can be recognised via memory moves of vector components.

— Complex operations —

- Fused-multiply-add instructions can be recognised by fusing multiply and add instructions.

— Inapplicable —

- Advanced Encryption Standard (AES) instructions implement a set of specific actions required for implementing AES cryptographic algorithm. This is an application specific set of instructions so it will not be available.

3.7.2 Altivec

Altivec is an extension to the PowerPC architecture. It has a very rich API which is available not only via assembly, but also via C operations. For the overview we are using [47] and as previously, concentrate on the primitives that are not identical with the operations in the proposed standard.

General features

- Vector literals a la OpenCL, for example, `(vector int)(3)` would mean a broadcast of value 3 to all the positions of the vector. This is supported by recognising identical elements in a vector construction: `(vint){x,x,x,x}`.
- `vec_step` operator takes a vector type argument and returns an integer value representing the amount by which a pointer to a vector element should be incremented to move by 16 bytes. This can be trivially expressed without introducing a special function.
- `printf` and `scanf` extension are supported by introducing separators and new modifiers for vector types. That is a very convenient thing to do, but it will not be available via the proposed standard.

Operations

We do not consider any operation involving saturation on its evaluation step. We will not consider operations on fixed-point data types. Most of the patterns we are going to describe are going to be very similar to the Intel patterns, so sometimes we are going to avoid details.

- `vec_ld*`, `vec_st*` — specific load/store operation, should not be available to a programmer directly, but can be used during the implementation of vector assignments.

— Advanced arithmetic —

- `d = vec_avg(a,b)` (Vector Average) can be recognised via the
  ```
  res = (a + b + broadcast (1)) / broadcast (2)
  ```
  pattern.
- `vec_max`, `vec_min` — maximum/minimum of the corresponding elements can be recognised using pattern matching similarly to Intel's min/max instructions.
- `vec_mule`, `vec_mulo` — multiply even/odd elements of the operands. Please note that the resulting vector has double-sized element type, so the pattern we are looking for is:
  ```
  vector (4, int32) a, b;
  vector (2, int64) res = {(int64)a[0]*b[0], (int64)a[2]*b[2]};
  ```

— Mathematical operations —

- `vec_abs` is obtainable through the math library.
- `vec_ceil` (Vector Ceiling) is obtainable through the math library.
- `vec_expte` (Estimate of 2 raised to the corresponding vector element of the argument) is obtainable via the math library.
- `vec_floor` — obtainable via math library.
- `vec_loge` (Estimated logarithm base two) is obtainable via the math library.
- `vec_re` Vector Reciprocal Estimate can be accessed via the math library.
- `vec_round` vector round. Should be obtainable via math library.
- `vec_sqrte` estimate square root, should be obtainable via the math library.

— Complex instructions —

- `vec_andc(a,b)` (Vector Logical AND with Complement) can be recognised via `a & ~b`.

- `vec_madd`, `vec_mladd` — multiply and add, can be recognised via the

  ```
  a * b + c
  ```

  pattern

- `d = vec_nmsub(a,b,c)` — negative multiply subtract is recognised via the

  ```
  -RndToFPNearest(a * b - c)
  ```

  pattern.

- `vec_nor` — not or is found through pattern matching.

- `vec_sel` select bits of the operands depending on the value of mask should be pattern-matched with: `m & a | ~m & b`.

— Element reordering —

- `vec_merge`, `vec_mergeh` — interleave elements from two operands in a way: `d[2*i] = a[i]; d[2*i+1] = b[i]` can be recognised via pattern matching shuffle operation.

- `vec_pack` — extract even bytes of the first operand and pack them in the first half of the result; extract even bytes from the second operand, pack them in the second half of the result. This can be found on pattern matching on shuffle, probably with special masks being pre-defined.

- `vec_pack*` same as above.

- `vec_sld` (Vector Shift Left Double) is similar to Intel's `PALIGNR` and can be recognised via shuffle.

- `d = vec_slo (a, b)` Vector Shift Left by Octet. The contents of `a` are shifted left by the number of bytes specified by bits $\mathtt{b}_{15}$`[1:4]`; only these 4 bits in `b` are significant for the shift value. Bytes shifted out of byte 0 are lost. Zeros are supplied to the vacated bytes on the right. The result is placed into `d`. This is found by pattern-matching shuffle operations.

- `vec_splat*` — broadcast a value into all the elements of the vector. This is found by pattern-matching.

- `vec_trunc` — truncate floating point, should be available via the math library.

- `vec_unpackh`, `vec_unpackl` — sign-extend the first V/2 elements into V/2-element vector.

— Comparison operations —

- `vec_allx` — recognise the reduction of scalar `&&`.

- `vec_anyx` — recognise the reduction of scalar `||`.

— Inapplicable —

- `vec_sll` Whole vector shift is currently not expressible. The C way of doing it would be by performing an operation on casted value, but we normally do not have a scalar type of the according size. Introducing a new type for a single operation might be an overkill.

- `vec_rl` Vector Rotate Left. There is no such an operation on scalars in C, so probably we will not introduce it on vectors as well. Hopefully it can be pattern-matched as:

  ```
  (value << shift) | (value >> (sizeof(value)*8 - shift))
  ```

  Rotate right can be treated similarly.

- `vec_msum(a,b,c)` — multiply `a` by `b`, sum horizontally group of four elements of the multiplication. Add with `c`. This can be recognised, but the pattern would be rather complicated.

- `vec_mfvscr` is architecture specific and is not supported.

- `vec_cmpb` (Vector Compare Bounds Floating-Point) is not supported, as the result is not a standard mask.

- `vec_ds*` — interface for hints to the cache. This is architecture-specific and not directly related to vector operations.

- `vec_addc` — get the carry of the operand summation is not supported. There is no way in C to get carry information when doing standard addition, so we will not support it in vector mode either.

- `vec_lvsl`, `vec_lvsr` — generates a permutation useful for aligning data from an unaligned address. This can be useful for code generation, but it is too architecture specific to be accessible directly.

3.7.3 Arm NEON extensions

Here is the same overview of ARM NEON SIMD extensions, based on [5, Chapter 5]. Again, we are not going to include any instructions which involve saturated arithmetic, and we are going to exclude operations on half-precision types (float16). As most of the patterns were already mentioned in the Intel and Altivec sections, some of the descriptions will be very brief.

— Complex instructions —

- VABA (Vector Absolute Difference and Accumulate) subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

VABD (Vector Absolute Difference) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results into the elements of the destination vector.

Both of the instructions can be recognised via `abs(a - b)`, assuming that we have `abs` that can recognise vector types.

- VMLA, VMLS, VNMUL, VNMLA, and VNMLS — Floating-point multiply accumulate, with optional negation. These can be pattern-matched.
- VTST — (Vector Test Bits) can be recognised via the

  ```
  a & b != broadcast (0)
  ```

  pattern.
- VACGE and VACGT (Vector Absolute Compare) can be found via pattern matching.
- VMVN (Vector Move Not) can be recognised via `a = ~b` pattern.
- VBIF, VBIT, and VBSL — Bitwise select operations, can be recognised via the

  ```
  mask & a | ~mask & b
  ```

  pattern
- VORN (Bitwise Or Not) can be found by pattern match on `a | ~b`.

— Reordering elements —

- VEXT (Vector Extract) extracts 8-bit elements from the bottom end of the second operand vector and the top end of the first, concatenates them, and places the result in the destination vector. This is found by permutation pattern matching.
- VREV reverse the order of sub-elements within each element of the vector, for example, reverse bytes in 32-bit vector components. This can be recognised via the shuffle operation.
- VSWP (Vector Swap) exchanges the contents of two vectors. This can be found by pattern-match in the same way as scalar swaps are found.
- VTRN (Vector Transpose) treats the elements of its operand vectors as elements of 2×2 matrices, and transposes the matrices. This can be pattern-matched from shuffle.
- VPADD (Vector Pairwise Add) adds adjacent pairs of elements of two vectors, and places the results in the destination vector. Similarly to Intel's horizontal sums. Recognise the

  ```
  {a[0]+a[1], a[2]+a[3], b[0]+b[1], b[2]+b[3]}
  ```

pattern.

— Mathematical operations —

- VABS and VSQRT can be obtained through the math library.
- VRECPE and VRSQRTE Vector Reciprocal Estimate and Reciprocal Square Root Estimate. These can be obtainable via the math library.

— Data movement —

- VLD is a very powerful instruction, which comes with a form of scatter/gather support. These can be found by pattern-matching independent load/stores of vector components.
- VDUP Broadcasting an element. This can be found by pattern matching.

— Inapplicable —

- VTBL (Vector Table Lookup) uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0. This is really useful, but this instruction is very specific for the particular architecture, so it will not be directly available to a C programmer.
- Interleave instructions — not available. The pattern can be created to recognise independent scalar moves, but it will be rather complex.
- VCLS, VCLZ, and VCNT — Vector Count Leading Sign bits, Vector Count Leading Sign zeroes, Vector Count set bits. There seem to be no way one could map this instruction directly into C. There is no such a thing for scalar cases.

3.8 Conclusions

In this chapter we have demonstrated a framework which allows one to encode explicit vector computations in a portable fashion. The framework is implemented as a set of C language extensions within the GCC compiler preserving backward compatibility with ANSI C standards. This approach is beneficial because of the following reasons:

1. A programmer or a compiler which uses C as a target language gets a chance to express vector computations of any complexity at the level of C without involving any additional libraries or frameworks and without taking care about the specifics of any particular architecture, but being sure that the vector code would be executed within the SIMD accelerators in case they are present.
2. Backward compatibility makes it easy to change existing software by rewriting a certain function or code region using vector types and operators.

3. The internal representation of vector operations is shared with the auto-vectoriser which means that any architecture that supports auto-vectorisation also supports explicit vectorisation and vice-versa and any improvement affects both parts of the compiler.

4. Vector operations within GCC are fully-fledged members of the flow-graph; hence they participate in the optimisation cycle similarly to scalar operations.

The downsides of the proposed approach are the following:

1. Currently the framework is supported only within the GCC compiler, which means that one will have difficulties in case of moving a code-base to a different compiler. One of the solutions to this problem is to make the language extensions fit into a new C language standard, but unfortunately this is a long and complicated process.

2. The price for the abstraction layer is a potential inability to use the newest SIMD instructions. There is always a gap in time between the architecture becoming available on the market and the code generator being able to use it correctly. There is no good solution here; however the good thing is that a programmer cannot and does not have to do much about it. Whenever a given pattern would be incorporated in a code-generator, the code should start to work faster automatically.

3.9 Future work

In the near term we aim to implement support for the missing operations available in a scalar mode, but not available in a vector mode. These operations are: `++`, `--`, `&&`, `||`, `!`. After that it is necessary to make more careful research with respect to the common SIMD patterns supported by various architectures and provide according C constructions. For example, currently there are no facilities to use horizontal vector instructions like sum or min/max of the entire elements.

Another important issue that we aim to address is vector alignment. Some ISAs have different instructions to load/store vectors from aligned and unaligned memory. Normally unaligned variant of the instruction is less efficient. Currently in C we cannot annotate a chunk of memory being aligned/unaligned and hence generate an efficient load/store instruction. In auto-vectorisers a similar problem is partially solved [39, 121] by transforming the computation. As we are in explicit mode, our goal is to provide annotations and mechanisms to emit a suitable instruction. As C arrays degenerate into pointers, `aligned` attribute does not give the desired effect. When applied to a pointer, the pointer itself will be aligned (not the memory where it is pointing). In case the attribute is applied to the type `t` and we construct a one-dimensional "array" via `t*` syntax, we will align each element of the "array", but this is not what we want. A solution here is to introduce a new attribute for aligning a chunk of memory. After that pointer dereferences and pointer-arithmetics must be analysed in order to see if the alignment gets changed. Unfortunately in the general case such an analysis is impossible as variables and functions can be external. However, we hope to address this problem by including the analysis in the Link Time Optimisation (LTO) cycle when all the information is statically available.

In the long term we hope to include the described extensions into a new C language standard and we even sketched the way it could be done. An extended standard would allow one to use various compilers for one and the same code.

Chapter 4

Layout-type system

In the previous chapter we have identified the necessity to express vectorisation explicitly in a portable way. In this chapter, we focus on the challenge of identifying suitable layout combinations that enable auto-vectorisation. In Chapter 5 we demonstrate transformations that have to follow after the inference is done and prove that the semantics of transformed programs is being preserved. We study experimental results at the end of this chapter, and more extensively in Chapter 6.

4.1 Introduction

We propose a type inference that identifies data layouts suitable for vectorisation. A functional core language serves as the basis for our formalisation. It constitutes a stripped-down version of the programming language SaC [50] which we use as a vehicle for our experiments and to implement the inference. Essential features of a functional language that our system depends on are: pure functions, n-dimensional arrays as first class citizens, data parallel loop-nests expressed using an explicit syntactical construction, and memory management being fully implicit to allow adjustments of data layouts.

We use the N-body problem as a case study throughout this chapter. It nicely demonstrates the difficulties occurring when attempting the classical approach to vectorisation and it also shows the effectiveness of our proposed approach.

4.2 Core programming language

The inference presented in this thesis is based on a functional language called SaC-λ which is a stripped down version of SaC similar to what is described in [51]. The main reason we are not using the full version is to make our presentation and reasoning simpler. Most of the constructions of SaC can be seen either as syntactic sugar over SaC-λ, or they are irrelevant in terms of our discussion. For

This chapter is based on the HPCS-2013 paper [125] more details on the differences between SaC and SaC-λ please refer to Chapter 6. The implementation of the inference uses full-fledged SaC.

The language contains only the bare essentials of the SaC language adjusted to a λ-calculus style in order to facilitate a more concise description of our techniques. Fig. 4.1 shows the syntax of our core language.

$$
\begin{array}{lcl}
\langle \textit{program} \rangle & ::= & \textbf{letrec}\ \langle \textit{fundef} \rangle^*\ \textbf{in}\ \langle \textit{expr} \rangle \\
 & & \\
\langle \textit{fundef} \rangle & ::= & \langle \textit{id} \rangle\ (\langle \textit{args} \rangle?\ ) = \langle \textit{expr} \rangle \\
 & & \\
\langle \textit{expr} \rangle & ::= & \langle \textit{const} \rangle \\
 & | & \langle \textit{id} \rangle \\
 & | & \langle \textit{id} \rangle\ (\ \langle \textit{exprs} \rangle?\ ) \\
 & | & \langle \textit{prf} \rangle\ (\ \langle \textit{exprs} \rangle?\ ) \\
 & | & \textbf{if}\ \langle \textit{expr} \rangle\ \textbf{then}\ \langle \textit{expr} \rangle\ \textbf{else}\ \langle \textit{expr} \rangle \\
 & | & \textbf{let}\ \langle \textit{id} \rangle = \langle \textit{expr} \rangle\ \textbf{in}\ \langle \textit{expr} \rangle \\
 & | & \textbf{map}\ \langle \textit{id} \rangle < \langle \textit{expr} \rangle\ \langle \textit{expr} \rangle \\
 & | & \textbf{reduce}\ \langle \textit{id} \rangle < \langle \textit{expr} \rangle\ \textbf{(}\ \langle \textit{id} \rangle\ \textbf{)}\ \langle \textit{expr} \rangle \\
 & & \\
\langle \textit{const} \rangle & ::= & \mathrm{c} \mid \textbf{[}\ \langle \textit{consts} \rangle\ \textbf{]} \\
 & & \\
\langle \textit{consts} \rangle & ::= & \langle \textit{const} \rangle\ (,\ \langle \textit{const} \rangle)^* \\
 & & \\
\langle \textit{prf} \rangle & ::= & \textbf{sel} \mid + \mid \textbf{-} \mid \ldots \\
 & & \\
\langle \textit{args} \rangle & ::= & \langle \textit{id} \rangle\ (,\ \langle \textit{id} \rangle)^* \\
 & & \\
\langle \textit{exprs} \rangle & ::= & \langle \textit{expr} \rangle\ (,\ \langle \textit{expr} \rangle)^*
\end{array}
$$

Figure 4.1: The syntax of SaC-λ.

As in full-fledged SaC, programs in our stripped down version consists of a set of potentially mutually recursive function definitions ($\langle \textit{fundef} \rangle^*$ in the $\langle \textit{program} \rangle$ rule) and a dedicated goal expression ($\langle \textit{expr} \rangle$ in the same rule). Expressions are either constants, variables or function applications ($\langle \textit{expr} \rangle$ sub-rules 1,2 and 3 accordingly). Anonymous functions, i.e. lambda abstractions, are not supported — this makes type inference and transformation of conditions easier (for more details please refer to Subsections 4.5.1 and 5.2.6). Function applications are written in C style, i.e. arguments is a comma-separated list of expressions wrapped in parentheses. Local variable definitions are expressed as let-constructs ($\langle \textit{expr} \rangle$ sub-rule 6) and conditionals ($\langle expr \rangle$ sub-rule 5) exist in the form of if-then-else expressions. Primitive operations ($\langle prf \rangle$ rule) contain a set of standard arithmetic operations, e.g. $+$, `-`, etc., comparisons and mathematical functions like **sqrt**, **sin**, etc. and selections **sel**. For convenience, for selections we use `e[iv]` interchangeably with **sel**`(iv, e)`. Additionally, our core language contains two combinators, **map** and **reduce** (the last two sub-rules of $\langle expr \rangle$). They serve as vehicle for expressing data parallel operations — array comprehension combinators. See Subsections 4.2.1 and 4.2.2 for more details.

In SAC-λ every value, conceptually, is a multi-dimensional array which is being represented as a pair: $\langle s, d \rangle$, where s is a shape and d is data. Both shape and data are vectors, which are denoted in square brackets, e.g. $[1, 2, 3]$. For example, an n-dimensional array is represented by the following pair:

$$\langle [s_1, \ldots, s_n], [d_1, \ldots, d_m] \rangle$$

where $[s_1, \ldots, s_n]$ denotes the shape of the array, i.e. its extent with respect to n individual axes, and the vector $[d_1, \ldots, d_m]$ contains all elements of the array in a linearised form.

To describe the evaluation rules for SAC-λ we will use a natural operational semantics as described in [67, 36] together with the proposed notation. A single judgement of evaluation is denoted as:

$$\rho \vdash E \Downarrow \alpha$$

where E is a SAC-λ expression, ρ is an environment and α is a result of the evaluation of E in ρ. The value of an expression E depends on the identifiers that occur free in E. These values are recorded in the environment. An environment ρ is an ordered list of pairs $v \mapsto \alpha$ where v is a name and α is a value. The symbol $\cdot$ separates the elements of the list:

$$x \mapsto \langle [], [1] \rangle \cdot y \mapsto \langle [], [5] \rangle \cdot x \mapsto \langle [], [10] \rangle$$

To look-up the environment it is scanned from *left to right* using the rules presented in Fig. 4.2.

$$\frac{}{x \mapsto v \cdot \rho \vdash x \mapsto v} \qquad \frac{\rho \vdash x \mapsto v \qquad x \neq y}{y \mapsto v_1 \cdot \rho \vdash x \mapsto v}$$

Figure 4.2: Variable look-up rules.

In the example environment from above x is associated with 1 and y gets a value 5. The old x's value 10 is also present in the environment but is never accessed. Additionally to array values, the environment is used to store function definitions. More precisely we need to store a function definition and the environment in which this function has been defined. That makes the domain of values stored in the environment to be:

1. multi-dimensional arrays — $\langle s, d \rangle$, and
2. a closure of the form $[\![\lambda x_1, \ldots, x_n.e, \rho]\!]$ which denotes a pair of multi-argument nameless function definition with body e and environment ρ.

The typical equation we want to solve with the formal system is:

$$\rho_0 \vdash P \Downarrow v$$

where P is a program, v is unknown and ρ_0 is an initial environment containing evaluators for primitive functions like $+$, `-`, etc.

Now we present the rules of evaluation. Scalars are in a normal form so we cannot reduce them any further. Constants of higher dimensions can be constructed using nested lists of expressions. We evaluate it as shown in Fig. 4.3

$$\textsc{Scalar}\ \frac{\rho \vdash n \text{ is a scalar}}{\rho \vdash n \Downarrow \langle [], [n] \rangle} \qquad \textsc{Array}\ \frac{\overset{n}{\underset{i=1}{\forall}} \rho \vdash e_i \Downarrow \langle [s_1, \ldots, s_m], [d^i_1, \ldots, d^i_p] \rangle}{\rho \vdash [e_1, \ldots, e_n] \Downarrow \langle [n, s_1, \ldots, s_m], [d^1_1, \ldots, d^1_p, \ldots, d^n_1, \ldots, d^n_p] \rangle}$$

Figure 4.3: Semantics of Arrays and Scalars in SAC-λ.

As an example consider expressions 5 and $[[1,2],[3,4]]$. They will be evaluated as follows:

$$5 \Downarrow \langle [], [5] \rangle \qquad\qquad [[1,2],[3,4]] \Downarrow \langle [2,2], [1,2,3,4] \rangle$$

For non-scalar values there has to be a mapping between a multi-dimensional array and its flat representation. As a default mapping we use standard row-major order. This mapping is fixed for any SAC-λ program. In order to define the semantics of selections, we first formalise the mapping $\mathcal{R}m$ of index vectors within n-dimensional index spaces into offsets within the row-major representation of n-dimensional arrays:

$$\mathcal{R}m(\langle [n], [i_1, \ldots, i_n] \rangle, \langle [n], [s_1, \ldots, s_n] \rangle) = 1 + \sum_{k=1}^{n} \left(\prod_{j=k+1}^{n} s_j \right) i_k$$

Please note that we add one to the row-major order as our enumeration of array elements in tuples starts with one while our row-major mapping $\mathcal{R}m$ assumes the C convention for indices, i.e., 0 is considered the lowest legal index.

For all legal indices, i.e., $0 \le i_k < s_k$ for all $k \in \{1, \ldots, n\}$, $\mathcal{R}m$ is a bijective function for which we can define $\mathcal{R}m^{-1}$ using integer division operations denoted with `div` and `mod`, where `div` is a quotient and `mod` is a modulo. We have

$$\mathcal{R}m^{-1}(a, \langle [n], [s_1, \ldots, s_n] \rangle) = \langle [n], [a_1, \ldots, a_n] \rangle$$

where

$$\forall k \in \{1,\dots,n\} : a_k = ((a-1) \texttt{ mod } \prod_{i=k}^{n} s_i) \texttt{ div } \prod_{i=k+1}^{n} s_i.$$

Please also note that we subtract one from a as enumeration of elements in arrays start from one.

With the definition of $\mathcal{R}m$ at hand we obtain the semantics for selections presented in Fig. 4.4.

$$\textsc{Sel}\quad \frac{\rho \vdash iv \Downarrow \langle [n], [i_1,\dots,i_n] \rangle \qquad \rho \vdash e \Downarrow \langle [s_1,\dots,s_n], [d_1,\dots,d_m] \rangle}{\begin{array}{c} l \equiv \mathcal{R}m(iv, \langle [n], [s_1,\dots,s_n] \rangle) \\ \rho \vdash \mathbf{sel}(iv, e) \Downarrow \langle [], [d_l] \rangle \end{array}}$$

Figure 4.4: Semantics of selection in SAC-λ.

Binary scalar arithmetic functions like addition, multiplication, etc. are defined on all the numerical types for scalar values only. Semantically this can be seen as a function application of a function with a body which we cannot inspect, but with the evaluator that is coming from the environment. The rule for primitive functions is presented in Fig. 4.5. The $sem(\times)$ notation refers to the semantics of a primitive operation defined by SAC-λ.

$$\textsc{Prf}\quad \frac{\begin{array}{c} \times \in \{+, \texttt{-}, \dots\} \\ \rho \vdash e_1 \Downarrow \langle [], [d_1] \rangle \qquad \rho \vdash e_2 \Downarrow \langle [], [d_2] \rangle \end{array}}{\rho \vdash e_1 \times e_2 \Downarrow \langle [], [d_1 \; sem(\times) \; d_2] \rangle}$$

Figure 4.5: Semantics of primitive functions in SAC-λ.

The semantics of LET, APP and IF use standard rules as described in many text books. The semantics of LETREC has to deal with a set of mutually recursive functions. In order to present this formally we follows the idea presented in [67] and we construct an environment that contains closures containing self-references. It can be also seen as an infinite tree of closures. That guarantees that a closure of one function definition can reach all the other function definitions of the **letrec**. The rules are presented in Fig. 4.6.

The only constructs that require special attention are the **map** and **reduce** operators as well as vectorised versions of primitive operations. Jointly, these constructs play key roles in our inference.

The **map** and **reduce** operators constitute simplified versions of the with-loop constructs in fully-fledged SAC[1]. They are array versions of the well-known combinators. Both operators compute expressions over an N-dimensional index space

[1]For details on with-loops in SAC see [50].

IF-TRUE
$$\frac{\rho \vdash e_1 \Downarrow \langle [], [\mathit{true}] \rangle \qquad \rho \vdash e_2 \Downarrow v}{\rho \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v}$$

IF-FALSE
$$\frac{\rho \vdash e_1 \Downarrow \langle [], [\mathit{false}] \rangle \qquad \rho \vdash e_3 \Downarrow v}{\rho \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v}$$

LET
$$\frac{\rho \vdash e_1 \Downarrow v_1 \qquad x \mapsto v_1 \cdot \rho \vdash e_2 \Downarrow v}{\rho \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow v}$$

APP
$$\frac{\begin{array}{c} \rho \vdash e_1 \Downarrow v_1 \qquad \dots \qquad \rho \vdash e_n \Downarrow v_n \\ \rho \vdash f \Downarrow [\![\lambda x_1, \dots, x_n . e, \rho_1]\!] \qquad x_1 \mapsto v_1 \cdot \dots \cdot x_n \mapsto v_n \cdot \rho_1 \vdash e \Downarrow v \end{array}}{\rho \vdash f(e_1, \dots, e_n) \Downarrow v}$$

LETREC
$$\frac{\begin{array}{c} \rho' = f_1 \mapsto [\![\lambda x^1, \dots, x^1_{A_1} . e_1, \rho']\!] \cdot \ \cdots \ \cdot f_n \mapsto [\![\lambda x^n_1, \dots x^n_{A_n} . e_n, \rho']\!] \cdot \rho \\ \rho' \vdash e \Downarrow v \end{array}}{\rho \vdash \textbf{letrec } f_1(x^1_1, \dots, x^1_{A_1}) = e_1 \dots f_n(x^n_1, \dots, x^n_{A_n}) = e_n \textbf{ in } e \Downarrow v}$$

Figure 4.6: Semantics of conditions, applications, let expressions and letrec in SAC-λ.

and then either combine the results in an array (map variant) or fold them using a binary operator.

4.2.1 Map

MAP
$$\frac{\begin{array}{c} \rho \vdash e_u \Downarrow \langle [n], [u_1, \dots, u_n] \rangle \wedge \overset{n}{\underset{i=1}{\forall}} u_i > 0 \\[2ex] \overset{u_1-1}{\underset{i_1=1}{\forall}} \dots \overset{u_n-1}{\underset{i_n=1}{\forall}} iv \mapsto \langle [n], [i_1, \dots, i_n] \rangle \cdot \rho \vdash e_{op} \\ \Downarrow \langle [s_1, \dots, s_m], [d_1^{[i_1,\dots,i_n]}, \dots, d_p^{[i_1,\dots,i_n]}] \rangle \end{array}}{\begin{array}{c} s_{\text{map}} \equiv [u_1, \dots, u_n, s_1, \dots, s_m] \\ d_{\text{map}} \equiv [d_1^{[0,\dots,0]}, \dots, d_p^{[0,\dots,0]}, \dots, d_1^{[u_1-1,\dots,u_n-1]}, \dots, d_p^{[u_1-1,\dots,u_n-1]}] \\ \rho \vdash \textbf{map } iv < e_u \; e_{op} \Downarrow \langle s_{\text{map}}, d_{\text{map}} \rangle \end{array}}$$

Figure 4.7: Semantics of **map** in SAC-λ.

The semantics of **map** is presented in Fig. 4.7. The MAP-rule shows that the upper limit e_u has to reduce to an n-element vector which determines the outermost n dimensions of the overall result. For each index vector iv within the n-dimensional index range starting at $[0, \dots, 0]$ to e_u, the expression e_{op} needs to evaluate to an m-dimensional result of one fixed shape. These m-dimensional results are then composed to the overall result by concatenating their element values. Note that we explicitly prohibit empty index spaces by requiring all the components of e_u to be greater than zero. This restriction avoids generating empty arrays as a result of the **map** operation. Although it might be considered as a restriction, it is irrelevant in terms of layout types.

Here is a short example to illustrate how **map** works:

$$\textbf{map}\ i < [2,3]\ 4 \xrightarrow{\text{evaluates to}} [[4,4,4],[4,4,4]]$$

Note that here we use SAC-λ syntax, which means that the constant $[2,3]$ from the perspective of the semantic rules is $\langle [2],[2,3] \rangle$, the constant 3 is $\langle [],[3] \rangle$, etc. The iteration space always starts with the zero vector and ends with the upper bound of the operator. For the above example the iterations space would be:

$$\{[0,0],[0,1],[0,2],[1,0],[1,1],[1,2]\}$$

An iteration space forms an array of the shape identical to the upper bound ($[2,3]$ in our example). The goal expression of the map (denoted with e) is being evaluated at every iteration:

$$\{i=[0,0] \wedge e \Downarrow 4, \quad i=[0,1] \wedge e \Downarrow 4, \quad i=[0,2] \wedge e \Downarrow 4, \quad i=[1,0] \wedge e \Downarrow 4,$$
$$i=[1,1] \wedge e \Downarrow 4, \qquad i=[1,2] \wedge e \Downarrow 4\}$$

The order of the evaluation is non-deterministic. Finally the **map** operator produces an array $[[4,4,4],[4,4,4]]$ as a result. That is, according to Fig. 4.7 $\langle s_{\text{map}} = [2,3], d_{\text{map}} = [4,4,4,4,4,4] \rangle$.

4.2.2 Reduce

The semantics of the **reduce** operator is presented in Fig. 4.8.

Reduce

$$\frac{\begin{array}{c} \rho \vdash e_u \Downarrow \langle [n],[u_1,\dots,u_n] \rangle \qquad \rho \vdash e_{\text{neut}} \Downarrow \langle [s_1,\dots,s_m],[d'_1,\dots,d'_p] \rangle \\[2ex] \overset{u_1-1}{\underset{i_1=1}{\forall}} \dots \overset{u_n-1}{\underset{i_n=1}{\forall}} iv \mapsto \langle [n],[i_1,\dots,i_n] \rangle \cdot \rho \vdash e_{op} \\ \Downarrow \mathfrak{D}^{[i_1,\dots,i_n]} \equiv \langle [s_1,\dots,s_m],[d_1^{[i_1,\dots,i_n]},\dots,d_p^{[i_1,\dots,i_n]}] \rangle \\[2ex] \overbrace{f(f(\dots f}^{u_1\cdot\,\cdots\,\cdot u_n}(e_{\text{neut}},\mathfrak{D}^{[0,\dots,0]}),\dots),\mathfrak{D}^{[u_1-1,\dots,u_n-1]}) \Downarrow \langle [s_1,\dots,s_m],[r_1,\dots,r_p] \rangle \end{array}}{\rho \vdash \textbf{reduce}\ iv < e_u\ (f)\ e_{op} \Downarrow \langle [s_1,\dots,s_m],[r_1,\dots,r_p] \rangle}$$

Figure 4.8: Semantics of **reduce** operator in SAC-λ.

Similar to the Map-rule, the reduce rule computes identically shaped values e_{op} for all indices within the index space defined by the upper limit vector e_u. However, here the result is obtained by consecutive folding using the binary function f. As one can see, the semantics definition prescribes left to right folding with respect to a row-major unrolling in the index space. Despite this definition, we demand f to be associative and commutative in order to enable arbitrary folding orders. Also, we assume that the neutral element e_{neut} can be inferred from the context of the function f. If not we can require a user to provide one, as it is done in the original SaC language. The presence of neutral element serves the only purpose to support empty index ranges, which does not have any effect when inferring types, but it gets more important during the code transformation.

To illustrate how reduce work consider the following example:

$$\textbf{reduce } i < [2,3] \ (+) \ 4 \xrightarrow{\text{evaluates to}} 24$$

As before we are using the SaC-λ syntax where $[2,3]$ from the perspective of semantic rules is $\langle [2], [2,3] \rangle$. The iteration space is identical to the iteration space of **map** we have considered earlier: it starts with the zero vector and ends with the upper bound of the operator:

$$\{[0,0],[0,1],[0,2],[1,0],[1,1],[1,2]\}$$

Expressions are evaluated on every iteration:

$$\{i = [0,0] \wedge e \Downarrow 4, \quad i = [0,1] \wedge e \Downarrow 4, \quad i = [0,2] \wedge e \Downarrow 4, \quad i = [1,0] \wedge e \Downarrow 4,$$
$$i = [1,1] \wedge e \Downarrow 4, \quad i = [1,2] \wedge e \Downarrow 4\}$$

where e is the goal expression of the **reduce**. Finally evaluated expressions are combined using the binary function:

$$(i = [0,0] \wedge e \Downarrow 4) \quad + \quad (i = [0,1] \wedge e \Downarrow 4) \quad + \quad (i = [0,2] \wedge e \Downarrow 4) \quad +$$
$$(i = [1,0] \wedge e \Downarrow 4) \quad + \quad (i = [1,1] \wedge e \Downarrow 4) \quad + \quad (i = [1,2] \wedge e \Downarrow 4) \quad = \quad 24$$

That is, according to Fig. 4.8 $\langle [], [24] \rangle$. Evaluation order is again non-deterministic. A binary function for reduce must be associative and commutative and the shape of the reduced result would be identical to the shape of the evaluated goal expression.

Note that in both operations **map** and **reduce** the indexing variable can be referred to within the expression:

$$\textbf{map } i < [4] \ i \xrightarrow{\text{evaluates to}} [0,1,2,3] \qquad \textbf{reduce } i < [4] \ (+) \ i \xrightarrow{\text{evaluates to}} 6$$

4.2.3 Vector operations

We assume that every arithmetic primitive operation has its SIMD counterparts operating on V-element vectors, where V is a given target architecture specific constant. The semantics of the SIMD primitive functions can be defined as follows:

$$\vec{f}([a_1, \ldots, a_V]) = [f(a_1), \ldots, f(a_V)]$$

where f is a scalar operation and $\vec{f}$ is its SIMD counterpart.

The selection operation `sel(iv, a)` which selects the element at the index position `iv` of the array `a` also has a SIMD counterpart, but its semantics is not as straight-forward. For more details please refer to Chapter 5.

4.3 Running example

The main motivating example that we are going to use throughout the chapter is an implementation of the N-body problem. The problem is defined as an iterative approximation of the movement of N planets. During each step, accelerations, velocities and positions of all the planets are being recomputed. Acceleration of the i-th planet is computed from the relative positions of all the other planets. Then the velocity and, in turn, the position of the i-th planet is updated using the newly computed acceleration. For more details, please refer to [129] which discusses the N-body problem in more detail.

```
letrec
    # (float[3], float[3]) → float[3]
    vplus (x,y) = map i < [3]  (x[i] + y[i])

    # (float[3]) → float
    l2norm (x) =
        sqrt ((reduce i < [3]  (+) x[i]*x[i]) + 0.01)

    # (float[3], float[3], float) → float[3]
    acceleration (posx, posy, mass) =
        let diff = map i < [3] (posx[i] - posy[i]);
            norm = l2norm (diff);
            norm3 = norm * norm * norm
        in
            map i < [3]  (diff[i] * mass / norm3)

    # (float[3], float[N, 3], float[N]) → float[3]
    planet_acc (pos, positions, masses) =
        reduce i < [N]  (vplus)
               let
                   p = map j < [3] positions[i ++ j];
               in
                   acceleration (pos, p, masses[i])

    # (float[N,3], float[N,3], float[N], float)
    # → (float[N,3], float[N,3])
    advance (positions, velocities, masses, dt) =
        let
            dt_vec = [dt, dt, dt];
            acc = map i < [N]
                  let
                      p = map j < [3] positions[i ++ j]
                  in
                      planet_acc (p,positions,masses);
            vel = map iv < [N,3]
                  velocities[iv] + acc[iv] * dt_vec;
            pos = map iv < [N,3]
                  positions[iv] + vel[iv] * dt_vec
        in
            (pos, vel)
in
    ...
```

Listing 4.1: Implementation of the N-body.

We provide a core implementation of the benchmark using SAC-λ in Fig. 4.1. It has been slightly adjusted from the version discussed in [129] to be better suited for demonstrating our inference technique. We use the symbol `#` for inserting comments and the symbol `;` as a shortcut for nested *let* expressions.

The function `advance` is updating arrays of positions and velocities on each time step. To do so, it first computes the mutual accelerations between all planets. This computation is achieved by mapping the function `planet_acc` over all planets. The function `planet_acc` computes the sum of forces that all planets have on the given position `pos`. This reduction in turn makes use of the function `acceleration` which computes the acceleration between two planets `posx` and `posy`.

Note that the N-body implementation grants a number of vectorisation opportunities, most of which are not valid under classical auto-vectorisers. The main reason is the way acceleration is computed between two planets. Acceleration, velocities and positions are stored in arrays of shape $[N, 3]$. The most compute-intensive operation happens on the inner dimension of the position array. Theoretically, that would be an ideal scenario for vectorisation; however, the problem is that the size of this dimension is too small. For most of the architectures the length of the float vector would be at least four, which means that loading/storing within a given layout would require masking, and has an implication on the alignment of load/store which might introduce overheads on targets like Intel. As a consequence, at that point classical auto-vectorisers would typically give up.

As a programmer, one might predict such a behaviour, and extend the innermost dimension to match the vector length. That might bring some performance gains, but the danger is that the increased memory footprint of the array will slow down the overall performance, as the inner dimension increased from 3 elements to V, which overall increases the memory footprint by $(V - 3)/3$ times. This is an aspect that is easy to overlook. More importantly, an alternative solution is typically being missed.

Rather than considering a vectorisation over the triplets, one might consider a vectorisation of the array of accelerations over the components of triplets. In that case the memory overhead would be substantially lower and the number of elements processed per vector operation would be higher. The drawback is that such a transformed data layout has an impact on the whole program. It might be:

1. arbitrarily difficult to rewrite large programs manually (which means that automating this process is of practical interest); and

2. the transformation might not be beneficial because of potential overheads introduced by vectorisation.

4.3.1 The key ideas in a nutshell

We generalise the idea of vectorisation across non-innermost dimensions as follows: For any given array A with shape $[s_1, \dots, s_n]$ we consider vectorisations in all possible axes $1, \dots, n$. A vectorisation in axis k will lead to a layout remapping into an array

A^k of shape $[s_1, \ldots, s_{k-1}, \lceil s_k/V \rceil, s_{k+1}, \ldots, s_n, V]$ where the V elements that in A are adjacent in axis k are now adjacent in the innermost axis $n+1$ of A^k.

The key problem then lies in the necessity to find out which layouts can result in vectorisation. Vectorisation potential in general stems from applications of primitive operations like + to elements of an array within the context of an independent loop. In such a setting, any of the array's axes can be chosen for vectorisation whose corresponding index is traversed by an independent loop. However, in practice, the choices in most cases are more limited due to other selections that are present within such a nesting of independent loops. If selections into more than one array exist, we get correlations between the layout choices of the arrays involved; examples can be observed in the function `vplus` of our N-body application, where we receive correlations between the arrays `x` and `y`. If several elements within the same array are selected, axes with 2 or more different accesses require more involved code transformations and are, therefore, less favourable. Furthermore, we have to take into account that the same array can be used in several loop nests, all of which may provide vectorisation opportunities; an example is the use of the array `positions` within the body of the function `advance` in our running example. Finally, the nests of independent loops may not exist within a single function body. Instead, layout demands for vectorisations may need to be propagated through function calls; an example is the function `acceleration` of the N-body application which exposes the layout demands on its first two arguments to the calling context in `planet_acc`.

The layout type system presented in this chapter allows us to control all the aforementioned aspects: we can propagate layouts through function calls, and we can control tightly what happens to all loop indices involved. Furthermore, it takes into account multiple uses of the same data structure within an entire application and ensures consistent layout transformations throughout.

4.4 A type system for data layouts

Before we describe the layout types let us clarify the term *type* and put it into the perspective of the language. We use layout types to denote transformations of expressions. Additionally we have standard element types like: *int*, *float*, etc. and we have shapes as a part of the types which form a subtyping hierarchy and participate in function overloading. For more details refer to [50]. That means that every expression of the language is annotated with a type that consists of three orthogonal components: element type, shape and layout type. In further discussions we are going to consider only layout types assuming that the element types and shapes have been inferred and are sound.

As explained informally in the previous section, for a given n-dimensional array we consider n different layout transformations. We denote these by the natural numbers $i \in \{1, \ldots, n\}$, where i refers to the layout transformation when the shape of an array changes from $[s_1, \ldots, s_n]$ into $[s_1, \ldots, \lceil s_i/V \rceil, \ldots, s_n, V]$ and V neighbour elements at axis i are placed at the newly introduced $n+1$-th axis (see an example at Subsection 4.4.1). In addition, we use 0 to denote the shape identity and we use $\triangle$ to denote a shape extension from $[s_1, \dots, s_n]$ into $[s_1, \dots, s_n, V]$. The transformed values are either replicated in case of constants or a represent V-fold selection. The latter is needed for the vectorisation of expressions that happen inside our map or reduce constructs. Finally, we add a layout type for index vectors. They play a crucial role in the layout inference as they introduce constraints between layouts of different arrays whenever they are used for selections from more than one array. Index vectors can have types $idx(m)$, $m \in \mathbb{Z}_+$ which denote that the m-th component of the index vector is considered for vectorisation. Now, we can define the set or layout types as

$$L = \mathbb{N} \cup \{\triangle\} \cup idx(m), m \in \mathbb{Z}_+ \qquad \text{(Layout types[2])}$$

We also introduce layout-type-signatures to denote the different possible layout transformations an individual function can be applied to. Formally, an n-argument function is described by a $(\tau_1, \dots, \tau_n) \to \tau_{n+1}$ type where all τ_i are layout types as defined above. We denote the union of L with all layout-type-signatures over L by LT.

4.4.1 Example

Here is a simple example to develop an intuition for the layout types and corresponding transformations. Let us consider a function that adds two matrices:

```
# (float[N,N], float[N,N]) → float[N,N]
matplus (a, b) =
    map i < [N, N]
        a[i] + b[i]
```

Both input matrices as well as the result are two dimensional arrays of shape $[N, N]$ with the element type *float*. Our goal is to find a valid layout transformation for the arrays in the program which would lead to replacement of scalar primitive operations with vector ones. In this work we restrict layout transformations of the arrays and only consider tiling with tiles of size $1 \times V$ across one of the array's axes and moving those tiles into newly created dimension. For our example we have the following cases (assuming that $N = 4$ and $V = 2$):

$$M^0 = \begin{pmatrix} [01, 02, 03, 04] \\ [05, 06, 07, 08] \\ [09, 10, 11, 12] \\ [13, 14, 15, 16] \end{pmatrix} \qquad M^1 = \begin{pmatrix} [[01, 05], [02, 06], [03, 07], [04, 08]] \\ [[09, 13], [10, 14], [11, 15], [12, 16]] \end{pmatrix} \qquad M^2 = \begin{pmatrix} [[01, 02], [03, 04]] \\ [[05, 06], [07, 08]] \\ [[09, 10], [11, 12]] \\ [[13, 14], [15, 16]] \end{pmatrix}$$

Here M^i denotes a transformed array M with respect to the layout type i. Here we assume that M^i is flattened using row-major order. We expect from our inference to identify two vectorisation possibilities for *matplus*: when both of the arguments

[2]Note here that despite of the infinite nature of the definition of L, for any given program L is finite as the natural numbers are bound by the maximum number of array axes present.

are of layout type 1 and both of the arguments are of layout type 2. Formally we would expect to see the following typings:

```
# Corresponds to $M^1$
# matplus :: $(1,1) \to 1$
matplus (a, b) =
    map i :: $idx(1)$ < [N, N]
        a[i] :: △ + b[i] :: △
```

```
# Corresponds to $M^2$
# matplus :: $(2,2) \to 2$
matplus (a, b) =
    map i :: $idx(2)$ < [N, N]
        a[i] :: △ + b[i] :: △
```

Here and further we use $e :: \tau$ notation to denote that e is of layout type τ to avoid confusion with standard typing for which we are using $e : float[N]$ notation. The above typings will correspond to the following transformations:

```
# (float[N/V,N,V], float[N/V,N,V])
# → float[N/V,N,V]
matplus (a1, b1) =
    map i < [N/V, N]
        vplus (vsel (i, a1),
               vsel (i, b1))
```

```
# (float[N,N/V,V], float[N,N/V,V])
# → float[N,N/V,V]
matplus (a2, b2) =
    map i < [N, N/V]
        vplus (vsel (i, a2),
               vsel (i, b2))
```

where `vplus` and `vsel` correspond to vector variants of $+$ and `sel`.

4.4.2 Environments

The purpose of the layout type system is to infer which combinations of layout choices for all data structures will enable some code vectorisation. To describe n such combinations within a single environment, we formalise environments as mappings from identifiers to n-element vectors of types, i.e., we have environments $\mathcal{E} \subset Id \times LT^n$. We denote the lookup of a variable v in $\mathcal{E}$ by $\mathcal{E}(v)$ and $\mathcal{E} \oplus (v, \langle\tau_1, \dots, \tau_n\rangle)$ denotes an environment that returns the vector type $\langle\tau_1, \dots, \tau_n\rangle$ for the variable v. Empty environment is denoted with $\{\}$.

Unless specified otherwise, we use small Greek letters to denote vectors of layout types, and indexes to get individual components. For example: $\tau \equiv \langle\tau_1, \dots, \tau_n\rangle, \tau_i \in LT$. We use $|\tau|$ to denote the number of components of a vector type τ, i.e. $|\langle\tau_1, \dots, \tau_n\rangle| = n$.

For a more succinct presentation of the type system, we use separate environments for all functions. We denote the collection of all these environments by a "function environment" $\mathcal{F} \subset Id \times \mathcal{E}$. Lookup of a function identifier f and presence of function identifiers are denoted in the same way as its done for standard environments, i.e., we use $\mathcal{F}(f)$ and $\mathcal{F} \oplus \mathcal{E}$, respectively.

In order to access the resulting type of the function f we introduce a meta-variable f in the relevant environment. We can look-up a function type using $\mathcal{F}(f)(f)$ notation. As we require all entries of one variable environment to have the same length, an environment $\mathcal{E}$ of a function f takes the general form:

$$\mathcal{E} = \{v_1 :: \langle\tau_1^1, \dots, \tau_n^1\rangle, \dots v_m :: \langle\tau_1^m, \dots, \tau_n^m\rangle, f :: \langle\tau_1^f, \dots, \tau_n^f\rangle\}$$

It can be seen as a matrix of size $(m+1) \times n$, where m is the number of local variables and arguments in the function this environment captures, +1 comes from the meta variable denoting the type of the function and n is the number of valid layout combinations for that function.

Each column in the matrix represents a layout combination for all variables and arguments of the function including its signature. We refer to the i-th column of an environment $\mathcal{E}$ by $\mathcal{E}^{[i]}$.

4.4.3 Ownership of $idx(k)$ and $\triangle$ types

The way we have presented $idx(k)$ and $\triangle$ types so far may lead to undesirable typings. This can be understood from the following example:

```
map i < [N]
    f (map j < [N] a[i] + b[j])
```

In the scalar case the function f gets an array b where each element is increased with $a[i]$. Now, the problem happens when both maps are vectorised. That is:

```
map i :: idx(1) < [N]
    f (map j :: idx(1) < [N] a[i] :: △ + b[j] :: △)
```

Such a typing will lead to a wrong result. At the first iteration, we will add

$$[[a_1, a_2, a_3, a_4], [a_1, a_2, a_3, a_4], \dots] + [[b_1, b_2, b_3, b_4], [b_5, b_6, b_7, b_8], \dots]$$

assuming that $V = 4$. However, it does not mean that such a map cannot be vectorised. Correct vectorisations would be:

$$[[a_1, a_1, a_1, a_1], [a_1, a_1, a_1, a_1], \dots] + [[b_1, b_2, b_3, b_4], [b_5, b_6, b_7, b_8], \dots]$$

in the case $j :: idx(1) \wedge i :: 0$ or

$$[[a_1, a_2, a_3, a_4], [a_5, a_6, a_7, a_8], \dots] + [[b_1, b_1, b_1, b_1], [b_1, b_1, b_1, b_1], \dots]$$

in the case of $i :: idx(1) \wedge j :: 0$.

The error happens because we lose a number of iterations in the first example as compared to the last two. In the scalar case, when $i :: 0 \wedge j :: 0$ the number of iterations is N^2. In the last two cases, we cut one of the map iteration spaces by V, and we add V elements at a time: $N/V \cdot N \cdot V = N^2$. However, in the first example, we cut both iteration spaces by V which results in $N/V \cdot N/V \cdot V = N^2/V \neq N^2$.

To solve this problem we have to track where the $\triangle$ type is coming from. There are two options:

1. it is either a replicated value, e.g. a constant; or

2. it is a selection into an array of layout type k using the index vector of layout type $idx(k)$.

For the first case, we are going to use $\triangle^0$ notation. For the second case, we need to track which map/reduce "owns" this $\triangle$ layout type. For that purpose, we enumerate map/reduces in a function and annotate index variable with this index. We will use `#map` and `#reduce` to denote the map/reduce indexes. As a result $idx(k)$ types get an additional parameter which we express as: $idx(k, \alpha)$, where α is an index of some map/reduce. Selections on the layout-type $idx(k, \alpha)$ result in $\triangle^\alpha$. By doing so, we will be able to verify that goal expressions of map/reduces will carry the right index, in case the body is of layout-type $\triangle$ and the index is of layout type $idx(k)$.

For our example we will have:

```
map i :: idx(1,α) < [N]
    f (map j :: idx(1,β) < [N] a[i] :: △^α + b[j] :: △^β)
```

Such a typing will be rejected, because plus imposes an equality constraint on $\triangle$ argument indexes.

To summarise: idx and $\triangle$ types are annotated with a parameter. For $\triangle$ types there are two kinds of concrete types: $\triangle^0$ which denotes a replicated value and $\triangle^{\#\text{map}}$ which means that the value was obtained via selection at the type $idx(k, \#\text{map})$ — the only possible family of concrete idx types. Note that for $\triangle^0$ there is no dual idx type. Also, function signatures may contain polymorphic idx and $\triangle$ types: $idx(k, \alpha)$ and $\triangle^\alpha$ respectively. During the application of a polymorphic type we run unification over the polymorphic parameters (α-s).

4.4.4 Type rules

With these definitions at hand, we can define a deduction system to characterise the validity of layout-transformations. The judgements of this deduction system are of the form $\mathcal{F}, \mathcal{E} \vdash expr :: \langle \tau_1, \dots, \tau_m \rangle$ where

- $\mathcal{F}$ is a function environment; it contains separate environments for all functions,
- $\mathcal{E}$ is an environment containing valid layout transformations for the identifiers in the current context,
- $expr$ is an expression,
- m is the number of valid layout transformations for the function under consideration, and
- τ_i are the m layout-transformations that $expr$ can undergo within the current function.

The type rules for the non-array-specific core of the language can be seen on Fig. 4.9.

Note that in the rest of the thesis we use $\mathcal{D}(e)$ to denote a number of axes in e and the length of the vector $\mathcal{S}_0(e)$ to denote the length of the first dimension.

$$\frac{e \Downarrow \langle [s_1, \dots, s_n], [d_1, \dots, d_p] \rangle}{\mathcal{D}(e) = n} \qquad \frac{e \Downarrow \langle [n], [d_1, \dots, d_p] \rangle}{\mathcal{S}_0(e) = n}$$

Most of these rules are multi-type versions of the standard rules for typing a first order applied λ-calculus: they only differ from their standard counterparts by dealing

Const
$$\frac{\underset{i=1}{\overset{n}{\forall}}\ \tau_i \in \{0,\dots,\mathcal{D}(c)\} \cup \{\triangle^0\}}{\mathcal{F},\mathcal{E} \vdash c :: \langle\tau_1,\dots,\tau_n\rangle}$$

Var
$$\frac{}{\mathcal{F},\mathcal{E} \vdash x :: \mathcal{E}(x)}$$

App
$$\frac{\begin{array}{c}\underset{i=1}{\overset{t}{\forall}}\ \mathcal{F},\mathcal{E} \vdash e_i :: \langle\tau_1^i,\dots,\tau_n^i\rangle \\ \mathcal{F},\mathcal{F}(f) \vdash f :: \langle(\sigma_1^1,\dots,\sigma_1^t) \rightarrow \sigma_1,\dots,(\sigma_m^1,\dots,\sigma_m^t) \rightarrow \sigma_m\rangle \\ \underset{i=1}{\overset{n}{\forall}} \quad \exists j \in \{1,\dots,m\} \quad \underset{k=1}{\overset{t}{\forall}}\ \sigma_j^k = \tau_i^k \wedge \sigma_j = \tau_i\end{array}}{\mathcal{F},\mathcal{E} \vdash f(e_1,\dots,e_t) :: \langle\tau_1,\dots,\tau_n\rangle}$$

Let
$$\frac{\mathcal{F},\mathcal{E} \vdash e_1 :: \tau \qquad \mathcal{F},\mathcal{E} \oplus (x,\tau) \vdash e_2 :: \sigma}{\mathcal{F},\mathcal{E} \vdash \textbf{let } x = e_1 \textbf{ in } e_2 :: \sigma}$$

Letrec
$$\frac{\begin{array}{c}\mathcal{F}' = \mathcal{F} \bigoplus_{i=1}^{n} \Big(f_i,\mathcal{F}(f_i) \oplus \big(f_i,\langle(\tau_1^1,\dots,\tau_{A_i}^1) \rightarrow \sigma_1,\dots,(\tau_1^{V_i},\dots,\tau_{A_i}^{V_i}) \rightarrow \sigma_{V_i}\rangle\big)\Big) \\ \underset{i=1}{\overset{n}{\forall}}\ \mathcal{F}',\mathcal{F}'(f_i) \bigoplus_{j=1}^{A_i} \big(a_j^i,\langle\tau_j^1,\dots,\tau_j^{V_i}\rangle\big) \vdash e_i :: \langle\sigma_1,\dots,\sigma_{V_i}\rangle \\ \mathcal{F}',\mathcal{E} \vdash e :: \rho\end{array}}{\mathcal{F},\mathcal{E} \vdash\ \textbf{letrec } f_1(a_1^1,\dots,a_{A_1}^1) = e_1,\dots,f_n(a_1^n,\dots,a_{A_n}^n) = e_n \textbf{ in } e :: \rho}$$

Figure 4.9: Non array-specific layout rules.

with vectors of n types for each identifier rather than a single type. Two rules are of special interest here: the Const rule for typing constants and the App rule for typing function applications.

The Const rule allows us to attribute any layout transformation type as long as we stay within the dimensionality of the constant or we choose to extend the shape. As a consequence, any possible type inference will have to imply type constraints from the context in order to constrain the types for constants.

The App rule correlates n potential layout combinations within the calling context with m potential layout combinations of the called context. This ensures that only those layout combinations are present for which suitable function layout transformations exist, which effectively ensures consistency throughout the entire program.

The rules that give rise to layout transformations are those for primitive operations and those for the map and reduce constructs are shown in Fig. 4.10.

As explained informally, in the previous section, we look for patterns where a primitive operation with a vector counterpart (Prf[$\triangle$] rule) is applied to element selections (Sel[$\triangle$] rule) into arrays that are located within a data parallel context (Map[$\triangle$] rule or Red[$\triangle$] rule). As we have seen in `matplus`, a vector version of

$$\textsc{Prf}[\triangle]\quad
\frac{\begin{array}{c}
\mathcal{F},\mathcal{E} \vdash e_1 :: \langle \tau_1,\dots,\tau_n \rangle \\
\mathcal{F},\mathcal{E} \vdash e_2 :: \langle \sigma_1,\dots,\sigma_n \rangle \\
\underset{1\le i\le n}{\rho_i} = \begin{cases}
0 & \tau_i = 0 \wedge \sigma_i = 0 \\
\triangle^{\alpha} & \tau_i \in \{\triangle^0, 0\} \wedge \sigma_i = \triangle^{\alpha} \\
\triangle^{\alpha} & \tau_i = \triangle^{\alpha} \wedge \sigma_i \in \{\triangle^0, 0\} \\
\triangle^{\alpha} & \tau_i = \triangle^{\alpha} \wedge \sigma_i = \triangle^{\alpha}
\end{cases}
\end{array}}
{\mathcal{F},\mathcal{E} \vdash +(e_1,e_2) :: \langle \rho_1,\dots,\rho_n \rangle}$$

$$\textsc{Map}[\triangle]\quad
\frac{\begin{array}{c}
\mathcal{F},\mathcal{E} \oplus (j, \langle \tau_1,\dots,\tau_n \rangle) \vdash e :: \langle \sigma_1,\dots,\sigma_n \rangle \\
\mathcal{F},\mathcal{E} \vdash u :: \langle \phi_1,\dots,\phi_n \rangle \wedge \overset{n}{\underset{i=1}{\forall}} \phi_i = 0 \\
\underset{1\le i\le n}{\rho_i} = \begin{cases}
k & \begin{array}{c}\tau_i = \mathit{idx}(k,\alpha) \wedge \sigma_i \in \{\triangle^{\alpha}, \triangle^0\} \\ k \in \mathbb{Z}_+, \alpha \overset{\text{def}}{=} \#\text{map}\end{array} \\
\triangle^{\alpha} & \tau_i = 0 \wedge \sigma_i = \triangle^{\alpha} \\
0 & \tau_i = 0 \wedge \sigma_i = 0 \\
\mathcal{S}_0(j) + k & \tau_i = 0 \wedge \sigma_i = k \in \mathbb{Z}_+
\end{cases}
\end{array}}
{\mathcal{F},\mathcal{E} \vdash \textbf{map } j < u \ e :: \langle \rho_1,\dots,\rho_n \rangle}$$

$$\textsc{Red}[\triangle]\quad
\frac{\begin{array}{c}
\mathcal{F},\mathcal{F}(f) \vdash f :: \langle (\upsilon_1,\upsilon_1) \to \upsilon_1,\dots,(\upsilon_m,\upsilon_m) \to \upsilon_m \rangle \\
\mathcal{F},\mathcal{E} \oplus (j, \langle \tau_1,\dots,\tau_n \rangle) \vdash e :: \langle \sigma_1,\dots,\sigma_n \rangle \\
\mathcal{F},\mathcal{E} \vdash u :: \langle \phi_1,\dots,\phi_n \rangle \wedge \overset{n}{\underset{i=1}{\forall}} \phi_i = 0 \\
\overset{n}{\underset{i=1}{\forall}} \exists j \in \{1,\dots,m\}\ \sigma_i = \upsilon_j \\
\underset{1\le i\le n}{\rho_i} = \begin{cases}
0 & \begin{array}{c}\tau_i = \mathit{idx}(k,\alpha) \wedge \sigma_i \in \{\triangle^{\alpha}, \triangle^0\} \\ k \in \mathbb{Z}_+, \alpha \overset{\text{def}}{=} \#\text{reduce}\end{array} \\
\triangle^{\alpha} & \tau_i = 0 \wedge \sigma_i = \triangle^{\alpha} \\
\sigma_i & \tau_i = 0 \wedge \sigma_i \in \mathbb{N}
\end{cases}
\end{array}}
{\mathcal{F},\mathcal{E} \vdash \textbf{reduce } j < u\ (f)\ e :: \langle \rho_1,\dots,\rho_n \rangle}$$

$$\textsc{If}[\triangle]\quad
\frac{\begin{array}{c}
\mathcal{F},\mathcal{E} \vdash p :: \langle \tau_1,\dots,\tau_n \rangle \\
\mathcal{F},\mathcal{E} \vdash t :: \langle \sigma_1,\dots,\sigma_n \rangle \\
\mathcal{F},\mathcal{E} \vdash f :: \langle \phi_1,\dots,\phi_n \rangle \\
\underset{1\le i\le n}{\rho_i} = \begin{cases}
\phi_i & \tau_i = 0 \wedge \sigma_i = \phi_i \wedge \sigma_i, \phi_i \neq \triangle \\
\triangle^{\alpha} & \tau_i \in \{\triangle^{\alpha}, \triangle^0, 0\} \wedge \sigma_i \in \{\triangle^0, 0\} \wedge \phi_i = \triangle^{\alpha} \\
\triangle^{\alpha} & \tau_i \in \{\triangle^{\alpha}, \triangle^0, 0\} \wedge \sigma_i = \triangle^{\alpha} \wedge \phi_i \in \{\triangle^0, 0\} \\
\triangle^{\alpha} & \tau_i \in \{\triangle^{\alpha}, \triangle^0, 0\} \wedge \sigma_i = \triangle^{\alpha} \wedge \phi_i = \triangle^{\alpha}
\end{cases}
\end{array}}
{\mathcal{F},\mathcal{E} \vdash \textbf{if } (p) \textbf{ then } t \textbf{ else } f :: \langle \rho_1,\dots,\rho_n \rangle}$$

$$\textsc{Idx}[\triangle]\quad
\frac{\begin{array}{c}
\mathcal{F},\mathcal{E} \vdash j :: \langle \tau_1,\dots,\tau_n \rangle \\
\mathcal{F},\mathcal{E} \vdash h :: \langle \sigma_1,\dots,\sigma_n \rangle \\
\underset{1\le i\le n}{\rho_i} = \begin{cases}
\mathit{idx}(k,\alpha) & \tau_i = \mathit{idx}(k,\alpha) \wedge \sigma_i = 0 \\
\mathit{idx}(\mathcal{S}_0(j) + k,\alpha) & \tau_i = 0 \wedge \sigma_i = \mathit{idx}(k,\alpha) \\
0 & \tau_i = 0 \wedge \sigma_i = 0
\end{cases}
\end{array}}
{\mathcal{F},\mathcal{E} \vdash j +\!\!+ h :: \langle \rho_1,\dots,\rho_n \rangle}$$

$$\textsc{Sel}[\triangle]\quad
\frac{\begin{array}{c}
\mathcal{F},\mathcal{E} \vdash j :: \langle \tau_1,\dots,\tau_n \rangle \\
\mathcal{F},\mathcal{E} \vdash a :: \langle \sigma_1,\dots,\sigma_n \rangle \\
\underset{1\le i\le n}{\rho_i} = \begin{cases}
\triangle^{\alpha} & \tau_i = \mathit{idx}(k,\alpha) \wedge \sigma_i = k \wedge k \in \mathbb{Z}_+ \\
\triangle^{\alpha} & \tau_i = 0 \wedge \sigma_i = \triangle^{\alpha} \\
0 & \tau_i = 0 \wedge \sigma_i \in \mathbb{N}
\end{cases}
\end{array}}
{\mathcal{F},\mathcal{E} \vdash \textbf{sel } (j,a) :: \langle \rho_1,\dots,\rho_n \rangle}$$

Figure 4.10: Layout rules enabling layout transformations.

plus is applied to selections to the arrays inside the map.

Depending on the nesting of map constructs, the $\textsc{Map}[\triangle]$ rule propagates type relations in a different way. We have to distinguish four different cases:

1. The map construct may control a layout transformation, i.e., it may be responsible for the data-parallel loop that is due to be vectorised. In this case, the corresponding axis k is attributed as type $\mathit{idx}(k,\alpha)$ for the index variable and the expression e needs to be of expansion type ($\triangle^{\alpha}$).

2. The map-construct can be syntactically located between the controlling map construct and the expression that is to be vectorised. In this case, the type for

the index j has to be of type 0 and the expression is of expansion type.

3. If we do not have a vectorisation at all, the types for the index, expression and the entire map construct are all 0.

4. Finally, the map construct may surround a map construct that controls a vectorisation. In that case, the expression is of some type k already and the result type of the map-construct has to reflect that we have a layout transformation on an inner dimensionality. This is done by adding the number of axes of the surrounding map to k.

The PRF[$\triangle$] rule captures all possible vectorisation cases: vectorisation is possible (indicated by the expansion type $\triangle$), whenever at least one argument has an expansion type. Finally, the only rule that gives rise to such an expansion type is the SEL[$\triangle$] rule for array selections. Similar to the MAP[$\triangle$] rule, the SEL[$\triangle$] rule has to deal with potential nesting of array selections:

1. The case that gives rise to vectorisation is where the index has type $idx(k, \alpha)$ and the array to select from has a matching layout transformation k.

2. If a selection is applied to an array that has given rise to vectorisation already (it is of type $\triangle^{\alpha}$) but the selection is still located inside the controlling map construct, the index needs to be of type 0 and the expansion type is propagated on.

3. Finally, the selection can be located outside of a controlling map construct, in which case the array is of type k and the result type as well as the index type are both of type 0.

The IDX[$\triangle$] rule allows for nested map/reduce constructs to be typeable. The main use case for that is a function application on non-scalar selections from an array.

4.5 Layout inference

Layout inference can be directly deduced from the layout rules similarly to monomorphic type systems. However, the main challenges come from:

1. The lengths of vector types in the environment are not known at the time we start the inference;

2. CONST defines the valid layout-types for components of the vector type, but we do not know which variant we should use for a certain component;

3. Recursive functions require a fixed point iteration; and

4. Parametrised idx and $\triangle$ types require unification.

The overall intuition behind the algorithm is that we start by adding all the valid type combinations and we cross out those combinations that have proven to be untypeable at each step of the inference. Every time we see a constant, we expand the existing environment by assuming that every column in the environment is compatible with any valid layout type for the given constant. Finally, as for recursive functions, we introduce a $\perp$ type when the type of the function is not yet known and we use a fixed-point iteration to eliminate $\perp$ types. As for unification, we follow the ideas of Hindley-Milner type inference [57, 90]. That is: we introduce a type-variable whenever the argument type is of type *idx* or $\triangle$; at function applications we instantiate polymorphic variables of the function signature and unify them with the existing variables; for all the other cases we unify the *idx* and $\triangle$ parameters according to the type rules.

The algorithm can be seen as a top-down traversal over the program, where for every term the layout rule corresponding to the type of the term is applied. We start the inference with an empty function $\mathcal{F}$ which is extended whenever a function is being processed.

4.5.1 The inference algorithm

We formulate the inference using $\mathcal{T}_{\text{inf}}$ schemata. The algorithm is a top-down traversal and the type of the overall program can be inferred by applying $\mathcal{T}_{\text{inf}}$ to the **letrec** expression. Further in this section we define $\mathcal{T}_{\text{inf}}$ application for all the expression kinds according to Fig. 4.1 and explain the details. That allows us to infer types for all the programs in our language.

Formally a $\mathcal{T}_{\text{inf}}$ application has the following form:

$$\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, e)$$

where $\mathcal{F}$ is a function environment, $\mathcal{E}$ is an environment related to an expression e we are inferring a type for. The inference step evaluates to a triplet:

$$(\mathcal{F}', \mathcal{E}', \tau) = \mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, e)$$

with potentially modified function environment $\mathcal{F}'$, potentially modified environment $\mathcal{E}'$ and a type (in the sense of environments, i.e. a tuple of layout types) τ. The meaning of this application is the following:

$$\frac{(\mathcal{F}', \mathcal{E}', \tau) = \mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, e)}{\mathcal{F}', \mathcal{E}' \vdash e : \tau}$$

Before we formalise the algorithm we have to introduce a couple of new meta-operators and meta-types. First of all, we introduce a bottom type, denoted with $\perp$. This type is used only in terms of the inference algorithm to drive a fixed point iteration and does not represent any valid layout. Intuitively it denotes the lowest type in a subtyping hierarchy: $\forall \tau \in L \quad \perp <: \tau$, if we have had sub-typing. The second meta-type is called *nil* and is denoted with $\square$. Semantically it means that in a given environment at the position where $\square$ appears, an expression is proven to be untypeable. The main purpose of $\square$ is to mark a column in the environment that has to be deleted.

The variable typing rule VAR says that the type of a variable is found by a lookup in the environment. However, the variables have to get there somehow. If we look at the inference rules, an environment is extended every time we use the $\oplus$ operator. Let expressions directly translates into the algorithm step. However, for function arguments, map and reduce index variables and constants the type has to be guessed. To solve this we are going to consider all the valid typings and eliminate those that are not sound during the inference. We introduce the T meta-operator to generate all the valid typings for an object x:

$$T(x) = \begin{cases} \langle 0, 1, \dots, \mathcal{D}(x), \triangle^0 \rangle & x \text{ is a constant} \\ \langle 0, 1, \dots, \mathcal{D}(x), idx(1, \alpha_1), \dots, idx(\mathcal{S}_0(x), \alpha_k), \triangle^0, \triangle^{\alpha_{k+1}} \rangle & x \text{ is an argument} \end{cases}$$

where α_i is obtained by calling a globally stateful function `newvar()` which returns a unique identifier to parametrise *idx* and $\triangle$ types. Note that we have decided to include $\triangle^0$ and $\triangle^\alpha$ explicitly, although $\triangle^0$ is an instance of a $\triangle^\alpha$. The alternative would be to start with $\triangle^\alpha$ and expand the environment whenever there is a special case for $\triangle^0$, however, we believe that this is easier for presentation purposes. Also, we cannot get rid of $\triangle^0$ cases entirely leaving polymorphic versions only, as $\triangle^0$ may remove an equality constraint on parameters. For example, think of a function `f (a, b) = a + b`. If we type it $(\triangle^\alpha, \triangle^\alpha) \to \triangle^\alpha$, the unification of α and zero would prohibit a typing $(\triangle^\alpha, \triangle^0) \to \triangle^\alpha$, at the same time we cannot type the overall function $(\triangle^\alpha, \triangle^\beta) \to \triangle^\alpha$ as it has an additional constraint that $\beta = 0$. Note that the *idx* types can be omitted entirely if the argument base type is not integral or the dimension of the argument is not one.

At every step we are going to reconstruct the environment by either expanding it with a new layout or shrinking it in case a certain layout combination is proven to be untypeable. In order to express this process formally, we introduce three more meta-operators on types: n-times type replication $R(\tau, n)$, n-times type component replication $C(\tau, n)$ and a helper meta-operator for tuple concatenation $+\!\!+$. We defined those as follows:

$$\begin{aligned} \tau +\!\!+\, \sigma &= \langle \tau_1, \dots, \tau_{|\tau|}, \sigma_1, \dots, \sigma_{|\sigma|} \rangle \\ \tau +\!\!+ \langle\rangle &= \langle\rangle +\!\!+\, \tau = \tau \\ C\left(\langle \tau_1, \dots, \tau_{|\tau|} \rangle, n\right) &= +\!\!+_{i=1}^{|\tau|} \left(+\!\!+_{j=1}^{n} \langle \tau_i \rangle \right) \\ R\left(\langle \tau_1, \dots, \tau_{|\tau|} \rangle, n\right) &= +\!\!+_{i=1}^{n}\, \tau \end{aligned}$$

For example $R(\langle 1,2,3\rangle, 2) = \langle 1,2,3,1,2,3\rangle$ and $C(\langle 1,2,3\rangle, 2) = \langle 1,1,2,2,3,3\rangle$.

Now, we define two operations on environments: environment extension $\oplus$ and environment shrinking $\ominus$. Please note, that here $\oplus$ has a different semantics than in the inference rules. However, they are related in the following sense: before we can add a new variable, which relates to the inference rules $\oplus$, an environment has to be extended using the $\oplus$ and the *idx* and $\triangle$ parameters in new columns have to be renamed. Environment extension is defined as:

$$\mathcal{E} \oplus \tau = \{(v, R(\sigma, |\tau|)) \quad | \quad (v, \sigma) \in \mathcal{E}\}$$

For every parameter variable in the environment column we generate a new name and replace old names with new names. Such a procedure preserves original constraints, but renames every variable in a column. We do this as unification happens in terms of a single column.

For example, if

$$\mathcal{E} = \{(v, \langle 0, 1, \triangle^{\alpha}\rangle)\}$$

then

$$\mathcal{E} \oplus \langle 2, 3\rangle = \{(v, \langle 0, 1, \triangle^{\beta}, 0, 1, \triangle^{\gamma}\rangle)\}.$$

Note that the number of components of every type in the environment is an invariant denoted with $l(\mathcal{E})$.

Environment shrinking is defined using a helper *rm* meta-operator as follows:

$$\begin{aligned} \mathcal{E} \ominus \tau &= \mathcal{E}' \quad \texttt{where} \\ \mathcal{E}' &= \{(v, rm(\sigma, \tau) \quad | \quad (v, \sigma) \in \mathcal{E}\} \quad \texttt{where} \\ rm(\sigma, \tau) &= \mathbin{+\!\!+}_{i=1}^{|\tau|} \text{if}\, \tau_i \neq \square\, \text{then}\, \langle \sigma_i \rangle\, \text{else}\, \langle\rangle \end{aligned}$$

For example, assuming that $\mathcal{E} = \{(v_1, \langle 1, 0\rangle), (v_2, \langle \triangle^{\alpha}, 1\rangle)\}$, we delete the columns of $\mathcal{E}$ at the position where we have $\square$ in the right-hand side operator. $\mathcal{E} \ominus \langle \square, 1\rangle = \{(v_1, \langle 0\rangle), (v_2, \langle 1\rangle)\}$. This is being used to remove columns of the environment which are proven to be untypeable.

With these definitions at hand we can now formally describe individual cases of the $\mathcal{T}_{\text{inf}}$ application.

[Alg-Letrec] — letrec expression

Letrec inference consists of two steps — inferring types for the functions and inferring type for the goal expression. However, as functions might call each other in their bodies and functions can be recursive, we populate $\mathcal{F}^0$ with function types that return $\perp$ for any possible layout combination that arguments can take. That ensures that a look-up in the function environment $\mathcal{F}^0$ is always successful. After that we infer types for all the functions again, considering their goal expressions. Formally we say that the inference of `letrec` is an inference of its goal expression assuming that functional environment $\mathcal{F}^n$ contains function types.

$$\mathcal{T}_{\text{inf}}\big(\{\},\{\},\text{letrec}\, f_1(a_1^1,\ldots,a_{m_1}^1) = e_1,\ldots,f_n(a_1^n,\ldots,a_{m_n}^n) = e_n \,\text{in}\, e\big) = \mathcal{T}_{\text{inf}}(\mathcal{F}^n,\{\},e)$$

To construct the environment $\mathcal{F}^n$ we start by creating environments $\mathcal{E}_i$ for every function f_i. The environment $\mathcal{E}_i$ contains a Cartesian product of all the possible argument types plus the type of the function. Components of this function type will have a form $(\tau_1,\ldots,\tau_n) \to \bot$ for every possible layout combination of all the arguments. A Cartesian product of the argument types is created by adding every argument x to $\mathcal{E}_i$ using the following procedure: expand all the existing types of $\mathcal{E}_i$ by applying $\mathcal{E}_i \oplus T(x)$ and add the entry $x : C(T(x), l(\mathcal{E}))$. This can be formalised as:

$$\begin{aligned}
\mathcal{E}_i^1 &= \{(a_1^i, T(a_1^i))\}\\
\mathcal{E}_i^2 &= \mathcal{E}_i^1 \oplus T(a_2^i) \cup \{(a_2^i, C(T(a_2^i), l(\mathcal{E}_i^1))\}\\
&\cdots\\
\mathcal{E}_i^{m_i} &= \mathcal{E}_i^{m_i-1} \oplus T(a_{m_i}^i) \cup \{(a_{m_i}^i, C(T(a_{m_i}^i), l(\mathcal{E}_i^{m_i-1}))\}\\
\forall_{j=1}^{l(\mathcal{E}_i^{m_i})} \rho_j &= (\mathcal{E}_i^{m_i}(a_1)_j,\ldots,\mathcal{E}_i^{m_i}(a_n)_j) \to \bot\\
\mathcal{E}_i^{m_i+1} &= \mathcal{E}_i^{m_i} \cup \{(f_i,\rho)\}
\end{aligned}$$

Note that the ρ_j is a function type for the layout combination of the arguments in the j-th column of $\mathcal{E}_i^{m_i}$. The notation $\mathcal{E}_i^{m_i}(a_k)_j$ denotes j-th component of the type that the argument a_k has in the environment $\mathcal{E}_i^{m_i}$. For example, for the following function:

```
# (float[N]) → 1
foo (a) = ...
```

we expect the following environment:

$$\begin{array}{rcccc}
\text{a}: & 0 & 1 & \triangle^0 & \triangle^\alpha\\
\text{foo}: & (0)\to\bot & (1)\to\bot & (\triangle^0)\to\bot & (\triangle^\alpha)\to\bot
\end{array}$$

We can construct a functional environment $\mathcal{F}^0$ that captures all the functions returning $\bot$ for any valid layout combination of the arguments as follows:

$$\mathcal{F}^0 = \bigcup_{i=1}^{n}\{(f_i, \mathcal{E}_i^{m_i+1})\}$$

Now, using $\mathcal{F}^0$ we can precise function types by inferring the types of the goal expressions. Formally we denote it as follows:

$$\begin{aligned}
(\mathcal{F}^1,\_,\_) &= \mathcal{T}_{\text{inf}}(\mathcal{F}^0, \mathcal{F}^0(f_1), f_1(a_1^1,\ldots,a_{m_1}^1) = e_1)\\
&\cdots\\
(\mathcal{F}^n,\_,\_) &= \mathcal{T}_{\text{inf}}(\mathcal{F}^{n-1}, \mathcal{F}^{n-1}(f_n), f_n(a_1^n,\ldots,a_{m_n}^n) = e_n)
\end{aligned}$$

Fixed point iterator

We use a fixed point iterator as we can have recursive functions. The main principle is that we introduce a bottom type if a function application hits a yet unfinished function. The bottom types can be absorbed by the condition, which gives rise to the fixed point. The step of a fixed point is an application of $\mathcal{T}_{\text{inf}}$ to the letrec expression.

The fixed point iteration stops when none of the types have been changed. The fixed point terminates because the size of any environment is bounded by the product of dimensionalities of the arguments, constants and map/reduce index variables. That is because environments can grow only on application of $\oplus$ which happens in the case of constants, arguments and map/reduce index variables. The let case does not count, as environment expansion happens via the expression we substitute with. This means that even if an environment would grow after elimination of a bottom type, it would not grow bigger than the bound. And as bottom types never replace non-bottom types, we can either precise a type (i.e. replace a bottom type with a concrete type) or leave it unchanged.

[Alg-Fundef] — function definitions

When we start the inference of a function definition, the functional environment will contain entries for all the functions from the letrec. Environments of individual functions will at least contain the arguments and the function type. The only thing that we need to do is to infer the type for the body of a function and replace the functional type. Formally we denote it as follows:

$$\begin{aligned}
\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{F}(f), f(a_1, \ldots, a_n) = e) = (\mathcal{F}'', \mathcal{E}', \mathcal{E}'(f)) \quad & \texttt{where} \\
(\mathcal{F}', \mathcal{E}, \sigma) &= \mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{F}(f), e) \\
\forall_{i=1}^{l(\mathcal{E})} \rho_i &= (\mathcal{E}(a_1)_i, \ldots, \mathcal{E}(a_n)_i) \to \sigma_i \\
\mathcal{E}' &= \{(v, \tau) \quad | \quad (v, \tau) \in \mathcal{E} \wedge v \neq f\} \cup \{(f, \rho)\} \\
\mathcal{F}'' &= \{(g, \mathcal{E}_g) \quad | \quad (g, \mathcal{E}_g) \in \mathcal{F}' \wedge g \neq f\} \cup \{(f, \mathcal{E}')\}
\end{aligned}$$

In the last two steps we reconstruct environment $\mathcal{E}$ by removing potentially imprecise type for f and adding a newly inferred one and we have updated functional environment $\mathcal{F}$ replacing environment of f with $\mathcal{E}'$.

[Alg-VarConst] — variables and constants

As the Var rule suggests, the type of a variable can be obtained by looking-up the environment:

$$\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, v) = (\mathcal{F}, \mathcal{E}, \mathcal{E}(v))$$

As we said earlier, for constants we need to guess the type, as they might have a number of typings and we cannot say which of them are sound. We extend the environment with $T(c)$:

$$\mathcal{T}_{\text{inf}}(\mathcal{F},\mathcal{E},c) = (\mathcal{F},\mathcal{E}\oplus T(c), C(T(c), l(\mathcal{E})))$$

For example, assuming that we have an environment $\mathcal{E}$:

$$a:\ \ 1\ \ 2$$

and we apply the inference to the constant 42 — $\mathcal{T}_{\text{inf}}(\mathcal{F},\mathcal{E},42)$, we expect an environment to become:

$$a:\ \ 1\ \ 2\ \ 1\ \ 2$$

as $T(42) = \langle 0, \triangle^0\rangle$.

[Alg-App] — function applications

Function applications require a bit more work. First we acquire layout types for all the arguments. After that we generate valid types for the results and finally we shrink the environment and add the resulting type.

The problem we potentially have is that the environment changes on every $\mathcal{T}_{\text{inf}}$ reduction step. This means that if we have an application of f to $e_1,\dots,e_n$ and we infer a type for e_1 with $(\mathcal{F},\mathcal{E}^1,\tau) = \mathcal{T}_{\text{inf}}(\mathcal{F},\mathcal{E},e_1)$, it might be invalidated in environment $\mathcal{E}^1$. For example, if we infer the type for the second argument: $(\mathcal{F},\mathcal{E}^2,\sigma) = \mathcal{T}_{\text{inf}}(\mathcal{F},\mathcal{E}^1,e_2)$, then the type of e_1 in $\mathcal{E}^2$ might differ from τ. In other words for a function application we need to find types of $e_1,\dots,e_n$ in environment $\mathcal{E}^n$. The easiest way to achieve that would be to keep expressions in the environment, in which case the argument types would be automatically updated on every reconstruction. For technical reasons we assume that all the argument expressions of a function application are variables. That would allow us to add the type of the argument expressions in the environment making sure that it is being updated properly; as adding e_2 has a potential effect on the type of e_1.

The transformation itself is straight-forward: instead of $f(e_1,\dots,e_n)$ we consider expression **let** $v_1 = e_1$ **in let** $v_2 = e_2 \dots$ **in**$f(v_1,\dots,v_n)$. We start with acquiring types for the arguments.

$$\mathcal{T}_{\text{inf}}(\mathcal{F},\mathcal{E},f(v_1,\dots,v_n)) = (\mathcal{F},\mathcal{E}^2, rm(\psi',\psi')$$
$$\forall_{i=1}^{l(\mathcal{E}^n)}\phi^i = \mathcal{E}(v_i)$$

Every argument v_i has type ϕ^i in environment $\mathcal{E}$. The next step would be to get a type of function f. We do that by enquiring functional environment $\mathcal{F}$ and then the environment that is bound to f.

$$\langle(\tau_1^1,\dots,\tau_n^1)\to\sigma_1,\dots,(\tau_1^m,\dots,\tau_n^m)\to\sigma_m\rangle = \mathcal{F}(f)(f)$$

At this point the $\mathcal{F}(f)(f)$ vector may contain repetitions which we have to remove, as otherwise the fix-point iteration will never finish. This can be understood from the following example:

```
foo (a, ...) =
    if (p (a)) then
        foo (a, ...)
    else
        a
```

This function has several arguments, so the first argument will be replicated several times. Now, at the first fixed-point cycle we will discover a recursive call and will use the types from the initial environment $\mathcal{F}^n$, which may contain identical types $(0, \dots) \to \bot$ more than once. If we are going to consider all such typings during the inference then every column of the environment where $a :: 0$ (and other types match) will be replicated. On the next fixed-point cycle it will be extended again, and so on. Conversion to a set consists of two steps:

1. remove identical types (up to the names of the *idx* and $\triangle$ polymorphic parameters); and

2. remove the instances of polymorphic types (typically coming from including $\triangle^0$ explicitly).

Consider an example:

```
f (a) = a + 1
```

The types for the expressions of this function will look like:

$$\begin{array}{rcccccc} a: & 0 & 0 & \triangle^0 & \triangle^0 & \triangle^\alpha & \triangle^\beta \\ 1: & 0 & \triangle^0 & 0 & \triangle^0 & 0 & \triangle^0 \\ a+1: & 0 & \triangle^0 & \triangle^0 & \triangle^0 & \triangle^\alpha & \triangle^\beta \end{array}$$

which results in the following vector-type:

$$\langle (0) \to 0, \;\; (0) \to \triangle^0, \;\; (\triangle^0) \to \triangle^0, \;\; (\triangle^0) \to \triangle^0, \;\; (\triangle^\alpha) \to \triangle^\alpha, \;\; (\triangle^\beta) \to \triangle^\beta \rangle$$

The first step of converting this vector into a set will find out that $(\triangle^0) \to \triangle^0$ is repeated two times, and that $(\triangle^\alpha) \to \triangle^\alpha, \;\; (\triangle^\beta) \to \triangle^\beta$ is the same type. By merging those typings we get the following set:

$$\{(0) \to 0, \;\; (0) \to \triangle^0, \;\; (\triangle^0) \to \triangle^0, \;\; (\triangle^\alpha) \to \triangle^\alpha\}$$

The second step will find out that $(\triangle^0) \to \triangle^0$ is an instance of $(\triangle^\alpha) \to \triangle^\alpha$ and the final set will look like:

$$\{(0) \to 0, \;\; (0) \to \triangle^0, \;\; (\triangle^\alpha) \to \triangle^\alpha\}$$

Coming back to our example — let us assume that while converting from a vector type to a set we managed to merge some typings and now instead of m types in a vector:

$$\langle (\tau_1^1, \dots, \tau_n^1) \to \sigma_1, \dots, (\tau_1^m, \dots, \tau_n^m) \to \sigma_m \rangle$$

we have M types in a set:

$$\{(\tau_1^1,\dots,\tau_n^1)\to\sigma_1,\dots,(\tau_1^M,\dots,\tau_n^M)\to\sigma_M\}$$

Now we have to match the $\phi_k^1,\dots,\phi_k^n$ argument types with the arguments of a function type $\tau_1^i,\dots,\tau_n^i$. To avoid global unification of τ_*^i arguments with local idx and $\triangle$ parameters we are going to instantiate polymorphic variables of the function signature and run unification on the new instances similarly to the way it is done in the Hindley-Milner system. Let us denote this process by introducing a new set of function signatures where polymorphic variables are instantiated (per environment column) as:

$$\{(\hat\tau_1^1,\dots,\hat\tau_n^1)\to\hat\sigma_1,\dots,(\hat\tau_1^M,\dots,\hat\tau_n^M)\to\hat\sigma_M\}$$

Note that function type might be not unique, i.e. for a chosen $\phi_k^1,\dots,\phi_k^n$ we might get several valid return types. For example, in function `f` above we have two return types for the argument of type 0. To deal with this, the result of the match would be a $l(\mathcal{E})$-element tuple of tuples, where every inner tuple consists of valid return types. Keep in mind that this does not affect termination because we are restricted with the dimensionality of arguments, and constructions like:

```
f (a) = map i < [N] f (a)
```

where the shape of the result increases at each application, are prohibited by the normal type system, and will not be typeable by the layout type system as well.

Also, some of the arguments might have $\perp$ as a part of its types. They will not match any function type, as the arguments are always consist of non-$\perp$ types. However, it does not mean that we have to consider this layout combination untypeable. We say that if any of $\phi_k^1,\dots,\phi_k^n$ are $\perp$, then the result of the application is $\perp$. Here is how we express it:

$$\forall_{i=1}^{l(\mathcal{E})}\psi_i=\begin{cases}\langle\sigma_1',\dots,\sigma_x'\rangle & \forall_{j=1}^{x}\sigma_j'=\hat\sigma_k \implies \exists k\le M\ (\forall_{w=1}^{n}\phi_w^i=\hat\tau_w^k\wedge\phi_w^i\ne\perp)\\ \langle\perp\rangle & \exists k\le n\ \left(\phi_k^i=\perp\right)\\ \langle\square\rangle & \text{otherwise}\end{cases}$$

In case $|\psi_i|>1$ we need to replicate the i-th column of $\mathcal{E}$ $|\psi_i|$ times. Then we flatten ψ by concatenating its components and finally we remove environment columns where ψ has $\square$ and we add updated ψ type for the function application.

$$\begin{aligned}&\mathcal{E}^1=\{(e,R(\tau_i,|\psi_i|))\,|\,(e,\tau)\in\mathcal{E}\}\\&\psi'=\mathbin{+\!\!+}_{i=1}^{m}\ \psi_i\\&\mathcal{E}^2=\mathcal{E}^1\ominus\psi'\end{aligned}$$

ψ' is a flattened ψ and $rm(\psi',\psi')$ is a ψ' type with $\square$ components being removed.

[Alg-Let] — let expressions

The let expression is processed by inferring a type for the expression we are substituting with, adding the variable of this type to the environment and inferring a type for the goal expression within the new environment.

$$\begin{aligned}\mathcal{T}_{\text{inf}}\left(\mathcal{F},\mathcal{E},\text{let}\,x = e_1\,\text{in}\,e_2\right) &= \left(\mathcal{F}^2,\mathcal{E}^3,\tau''\right) \quad \texttt{where}\\ (\mathcal{F}^1,\mathcal{E}^1,\tau) &= \mathcal{T}_{\text{inf}}(\mathcal{F},\mathcal{E},e_1)\\ \mathcal{E}^2 &= \mathcal{E}^1 \cup \{(x,\tau)\}\\ (\mathcal{F}^2,\mathcal{E}^3,\tau') &= \mathcal{T}_{\text{inf}}(\mathcal{F}^1,\mathcal{E}^2,e_2)\end{aligned}$$

[Alg-Prf] — primitive functions

Primitive functions can be handled via constructing a function type for the primitive function and adding it into the function environment:

$$\begin{aligned}\tau_+ = \langle &(0,0)\to 0,\quad (0,\triangle^\alpha)\to\triangle^\alpha,\quad (\triangle^\alpha,0)\to\triangle^\alpha,\\ &(\triangle^\alpha,\ \triangle^0)\to\triangle^\alpha\quad (\triangle^0,\ \triangle^\alpha)\to\triangle^\alpha\quad (\triangle^\alpha,\ \triangle^\alpha)\to\triangle^\alpha\rangle\\ \mathcal{T}_{\text{inf}}(\mathcal{F},\mathcal{E},a+b) &= \mathcal{T}_{\text{inf}}(\mathcal{F}\cup\{(+,\{(+,\tau_+)\})\},\mathcal{E},+(a,b))\end{aligned}$$

Please note, that by constructing the inference of primitive functions this way we guarantee that the function will be still typeable even in case one of the arguments is of type $\perp$. It would be desirable to use such an approach in other rules, however some of the constraints are easily reconstructable as function types.

[Alg-MapRed] — map and reduce expressions

Similarly to application, we abstract the upper bound expression u into a variable v_u, we expand the environment with all the valid types for the index variable j and infer a type for the goal expression e.

$$
\begin{aligned}
\mathcal{T}_{\text{inf}}(\mathcal{F},\mathcal{E},\text{map}\, j < v_u\, e) = \big(\mathcal{F},\mathcal{E}^3, rm(\psi,\psi)\big) \quad & \texttt{where} \\
\tau = \langle 0, idx(1,\#\text{map}),\dots, idx(\mathcal{S}_0(j),\#\text{map})\rangle & \\
\mathcal{E}^1 = \mathcal{E} \oplus \tau \cup \{(j, C(\tau, l(\mathcal{E}))\} & \\
(\mathcal{F},\mathcal{E}^2,\sigma) = \mathcal{T}_{\text{inf}}(\mathcal{F},\mathcal{E}^1, e) & \\
\upsilon = \mathcal{E}^2(v_u), \tau = \mathcal{E}^2(j) & \\
\forall_{i=1}^{l(\mathcal{E}^2)} \psi_i = \begin{cases} k & \tau_i = idx(k,\alpha) \wedge \sigma_i \in \triangle^\alpha, \triangle^0 \wedge \upsilon_i = 0 \\ & \qquad \alpha = \#\text{map} \\ \triangle^\alpha & \tau_i = 0 \wedge \sigma_i = \triangle^\alpha \wedge \upsilon_i = 0 \\ 0 & \tau_i = 0 \wedge \sigma_i = 0 \wedge \upsilon_i = 0 \\ \mathcal{S}_0(j) + k & \tau_i = 0 \wedge \sigma_i = k \in \mathbb{Z}_+ \wedge \upsilon_i = 0 \\ \perp & \tau_i = \perp \vee \sigma_i = \perp \vee \upsilon_i = \perp \\ \square & \text{otherwise} \end{cases} & \\
\mathcal{E}^3 = \mathcal{E}^2 \ominus \psi &
\end{aligned}
$$

The inference for a reduce expression is made in a similar fashion up to the generation of the ψ type which directly follows from the conditions on type ρ in the Red[$\triangle$] typing rule.

[Alg-Sel] — selections and index concatenations

Similarly to application, both arguments of the selection are abstracted into variables v_i and v_e, and the resulting type is generated using conditions on the ρ type in the Sel[$\triangle$] rule.

$$
\begin{aligned}
\mathcal{T}_{\text{inf}}(\mathcal{F},\mathcal{E},\text{sel}(v_i,v_e)) = (\mathcal{F},\mathcal{E}', rm(\psi,\psi)) \quad & \texttt{where} \\
\tau = \mathcal{E}(v_i), \sigma = \mathcal{E}(v_e) & \\
\forall_{i=1}^{l(\mathcal{E})} \psi_i = \begin{cases} \triangle^\alpha & \tau_i = idx(k,\alpha) \wedge \sigma_i = k \wedge k \in \mathbb{Z}_+ \\ \triangle^\alpha & \tau_i = 0 \wedge \sigma_i = \triangle^\alpha \\ 0 & \tau_i = 0 \wedge \sigma_i \in \mathbb{N} \\ \perp & \tau_i = \perp \vee \sigma_i = \perp \\ \square & \text{otherwise} \end{cases} & \\
\mathcal{E}' = \mathcal{E} \ominus \psi &
\end{aligned}
$$

The inference for the index concatenation is constructed similarly.

[Alg-If] — condition

Condition deserve special attention here as it is the only kind of expression that is able to absorb $\perp$ types in a branch, and propagate non-$\perp$ layout-types. This is a basic mechanism of the fix point iterator — in case there is a recursive call to a not yet inferred function in one of the branches, the return type of this function call would be $\perp$ and it would be propagated up to the branch expression. That would allow us to infer the type of the function and on the next iteration of the fixed point to precise it in case some of the layout combinations in the branch are untypeable. Similarly to the case of application we abstract predicate, then branch and else branch expressions into the variables v_p, v_t and v_f accordingly. Here is the rule:

$$
\begin{aligned}
\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, \text{if}\, v_p\, \text{then}\, v_t\, \text{else}\, v_f) = (\mathcal{F}, \mathcal{E}', rm(\psi, \psi)) \quad \texttt{where}& \\
\tau = \mathcal{E}(v_p), \sigma = \mathcal{E}(v_t), \phi = \mathcal{E}(v_f)& \\
\forall_{i=1}^{l(\mathcal{E})} \psi_i = \begin{cases}
\phi_i & \tau_i = 0 \wedge \phi_i = \sigma_i \wedge \sigma_i, \phi_i \neq \triangle \\
 & \tau_i \in \{0, \triangle^0, \triangle^\alpha\} \\
\triangle & \quad \wedge\, (\tau_i, \sigma_i) \in \{(0, \triangle^\alpha), (\triangle^0, \triangle^\alpha), \\
 & \qquad\qquad (\triangle^\alpha, 0), (\triangle^\alpha, \triangle^0), (\triangle^\alpha, \triangle^\alpha)\} \\
\perp & \tau_i = \perp \vee (\phi_i = \perp \wedge \sigma_i = \perp) \\
\phi_i & \tau_i = 0 \wedge \phi_i \neq \perp \wedge \sigma_i = \perp \\
\sigma_i & \tau_i = 0 \wedge \phi_i = \perp \wedge \sigma_i \neq \perp \\
\triangle^0 & \tau_i = \triangle^0 \wedge \phi_i \in \{0, \triangle^0\} \wedge \sigma_i = \perp \\
\triangle^\alpha & \tau_i = \triangle^\alpha \wedge \phi_i \in \{0, \triangle^0, \triangle^\alpha\} \wedge \sigma_i = \perp \\
\triangle^0 & \phi_i = \perp \wedge \sigma_i \in \{0, \triangle^0\} \\
\triangle^\alpha & \tau_i = \triangle_\alpha \wedge \phi_i = \perp \wedge \sigma_i \in \{0, \triangle^0, \triangle^\alpha\} \\
\square & \text{otherwise}
\end{cases}& \\
\mathcal{E}' = \mathcal{E} \ominus \psi&
\end{aligned}
$$

4.5.2 Sample layout inference

We are going to consider an application of the inference algorithm to the `vplus` function, which is a part of the N-body code. The function is defined as:

```
# (float[3], float[3]) → float[3]
vplus (x,y) = map i < [3]  (x[i] + y[i])
```

It adds two three-element vectors component-wise.

For presentation purposes we are going to split the inference process into two steps. First we are going to infer types ignoring the *idx* and $\triangle$ parameters and then, we will annotate the columns of the environment that contain the *idx* and $\triangle$ types with parameters and check if the typing still holds. We can always do this because, by not considering parameters, we implicitly are able to unify arbitrary parameters. That means that we can only get a larger (or equal) set of typings, which can be further restricted. None of the types will be lost.

When the inference of the function definition starts, the [Alg-Letrec] rule has created an environment that consist of all the possible argument layout type combinations returning bottom type. Here is how the environment $\mathcal{F}(\texttt{vplus})$ looks:

$x:$	0	1	$idx(1)$	$\triangle$	0	1	$idx(1)$	$\triangle$	0	1	$idx(1)$	$\triangle$	0	1	$idx(1)$	$\triangle$
$y:$	0	0	0	0	1	1	1	1	$idx(1)$	$idx(1)$	$idx(1)$	$idx(1)$	$\triangle$	$\triangle$	$\triangle$	$\triangle$
$*:$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

We use $*$ to denote the return type of the function body to save space on the page.

Now, according to the [Alg-MapRed] inference rule, we expand $\mathcal{E}$ further by adding a type for i and a type for the upper expression. The valid types for i would be $\sigma = \langle 0, idx(1)\rangle$. For presentation purposes we are going to add σ without expanding the environment. As for the upper expression, we could have expanded the environment with all the valid types for [3], but as we are inferring the type inside the *map* expression, we know that all the types other than 0 would be cancelled out, so we might just add a vector of zeroes. Here is an updated environment:

$x:$	0	1	$idx(1)$	$\triangle$	0	1	$idx(1)$	$\triangle$	0	1	$idx(1)$	$\triangle$	0	1	$idx(1)$	$\triangle$
$y:$	0	0	0	0	1	1	1	1	$idx(1)$	$idx(1)$	$idx(1)$	$idx(1)$	$\triangle$	$\triangle$	$\triangle$	$\triangle$
$i:$	σ	σ	σ	σ	σ	σ	σ	σ	σ	σ	σ	σ	σ	σ	σ	σ
$[3]:$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$*:$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Note that when we use [3] in the environment we assume the matching variable for the upper expression in the map. The body of the map is a primitive operation. Following the primitive function rule [Alg-Prf], we assume that we have surrounding variables for subexpression so we infer types for those first. The left-hand side expression is a selection $\text{sel}(i, x)$ which in this particular case would produce a type for combinations: $(0,0), (0,1), (idx(1), 1), (0, \triangle)$ and cross out all the rest columns. After application of [Alg-Sel] inference rule, the environment looks like:

$i:$	0	0	$idx(1)$	0	0	0	$idx(1)$	0	0	0	$idx(1)$	0	0	0	$idx(1)$	0
$x:$	0	1	1	$\triangle$	0	1	1	$\triangle$	0	1	1	$\triangle$	0	1	1	$\triangle$
$y:$	0	0	0	0	1	1	1	1	$idx(1)$	$idx(1)$	$idx(1)$	$idx(1)$	$\triangle$	$\triangle$	$\triangle$	$\triangle$
$x[i]:$	0	0	$\triangle$	$\triangle$	0	0	$\triangle$	$\triangle$	0	0	$\triangle$	$\triangle$	0	0	$\triangle$	$\triangle$
$[3]:$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$*:$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Note that we got rid of all the σ-s at this step. The right-hand side of the plus operation is also selection: $\texttt{sel}(i, y)$, so we apply the [Alg-Sel] inference step again and the new environment looks like:

$i:$	0	0	0	0	0	$idx(1)$	0	0	0	0
$x:$	0	1	$\triangle$	0	1	1	$\triangle$	0	1	$\triangle$
$y:$	0	0	0	1	1	1	1	$\triangle$	$\triangle$	$\triangle$
$x[i]:$	0	0	$\triangle$	0	0	$\triangle$	$\triangle$	0	0	$\triangle$
$y[i]:$	0	0	0	0	0	$\triangle$	0	$\triangle$	$\triangle$	$\triangle$
$[3]:$	0	0	0	0	0	0	0	0	0	0
$*:$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Finally, we apply the [Alg-Prf] inference rule to $x[i]$ and $y[i]$ which is an inner body of the map. This allows us to infer the type for map which would also be a type for the body of the function (denoted with $*$ in the environment). After the application the environment would look like:

$i:$	0	0	0	0	0	$idx(1)$	0	0	0	0
$x[i]:$	0	0	$\triangle$	0	0	$\triangle$	$\triangle$	0	0	$\triangle$
$y[i]:$	0	0	0	0	0	$\triangle$	0	$\triangle$	$\triangle$	$\triangle$
$x[i] + y[i]:$	0	0	$\triangle$	0	0	$\triangle$	$\triangle$	$\triangle$	$\triangle$	$\triangle$
$[3]:$	0	0	0	0	0	0	0	0	0	0
$x:$	0	1	$\triangle$	0	1	1	$\triangle$	0	1	$\triangle$
$y:$	0	0	0	1	1	1	1	$\triangle$	$\triangle$	$\triangle$
$*:$	0	0	$\triangle$	0	0	1	$\triangle$	$\triangle$	$\triangle$	$\triangle$

That finalises the first step of the inference for `vplus`. The function type can be seen from the last three lines of the environment. Now let us annotate the idx and $\triangle$ types with parameters and perform unification. In this function we have one **map** construct. Let us assume, that it has an index M and gives rise to $idx(k, M)$ and $\triangle^M$ layout types. The new environment will look like:

$$\begin{array}{rcccccccccccc}
i: & 0 & 0 & 0 & 0 & 0 & idx(1,M) & 0 & 0 & 0 & 0 & 0 & 0 \\
x[i]: & 0 & 0 & \triangle^\alpha & 0 & 0 & \triangle^M & \triangle^\alpha & 0 & 0 & \triangle^\alpha & \triangle^0 & \triangle^\alpha \\
y[i]: & 0 & 0 & 0 & 0 & 0 & \triangle^M & 0 & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha & \triangle^0 \\
x[i]+y[i]: & 0 & 0 & \triangle^\alpha & 0 & 0 & \triangle^M & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha \\
[3]: & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
x: & 0 & 1 & \triangle^\alpha & 0 & 1 & 1 & \triangle^\alpha & 0 & 1 & \triangle^\alpha & \triangle^0 & \triangle^\alpha \\
y: & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha & \triangle^0 \\
*: & 0 & 0 & \triangle^\alpha & 0 & 0 & 1 & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha & \triangle^\alpha
\end{array}$$

As we can see, none of the typings of the first step got cancelled, the typing $(1, 1) \to 1$ produces a concrete $idx(1, M)$ type, after that both selections become of type $\triangle^M$ and finally it is consumed by the **map** that has initiated it. The third column from the right has unified α-s in the arguments because of the plus primitive operation. The last two columns are initiated by the $(\triangle^\alpha, \triangle^0) \to \triangle^\alpha$ and $(\triangle^0, \triangle^\alpha) \to \triangle^\alpha$ typings of the plus.

The resulting type of a simple function like vector addition might be quite surprising or counterintuitive, but let us try to develop some intuition regarding this matter. The variety of possibilities comes from two facts:

1. We are allowed to perform a primitive operation on mixed scalar/vector arguments. Despite that, it is not necessary for constants, as we can always promote a scalar to a vector by assigning a $\triangle^0$ type to it; we cannot do that in case of expressions with dependencies. For example, in the expression $a[i] :: \triangle^\alpha + b[j] :: 0$, there is no way to promote $b[j]$ to vector. However, we can still apply a vector plus to it.

2. Scalar selection can be performed on arrays of layout-type $k \in \mathbb{Z}_+$. If an array has a type k, then we group its elements across dimension k into vectors. But we can still get a scalar component from the vectorised array by selecting a vector and then selecting the component from the vector.

Intuitively, the `vprod` types $(0, 0) \to 0, (1, 1) \to 1$ and $(\triangle^\alpha, \triangle^\alpha) \to \triangle^\alpha$ should be valid for vector additions, and so they are. Now, in the $(0, 0) \to 0$ type we can replace the first or second or both arguments with 1 and we should still get a 0 type as a result. Also, we can have $(0, \triangle^\alpha) \to \triangle^\alpha$ as we can promote the scalar selection via vector plus. We can swap the arguments and get $(\triangle^\alpha, 0)$ in the arguments, because plus is commutative. Finally, if we make a scalar selection from 0 argument, we should be able to select from the argument of type 1, which gives us types $(1, \triangle^\alpha)$ and $(\triangle^\alpha, 1)$ argument types.

4.5.3 Inference complexity

From the inference algorithm we know that a variable environment can change its size during inference, so let us estimate the upper bound. The size of the environment depends on the dimensionality of the arguments, on the number of maps/reduces in a function and on the number of constants per function. Putting these all together:

$$
\begin{aligned}
O(\mathcal{F}(f)) = \prod_{a \in \mathrm{Args}(f)} \mathcal{D}(a) \quad &\cdot \prod_{m \in \mathrm{Maps}(f)} \mathcal{S}_0(\mathrm{index}(m)) \\
\cdot \prod_{r \in \mathrm{Reduces}(f)} \mathcal{S}_0(\mathrm{index}(r)) \quad &\cdot \prod_{c \in \mathrm{Consts}(f)} \mathcal{D}(c)
\end{aligned}
$$

where *index* yields the index variable of the map/reduce.

The variables of lets are local and never expand the environment. Although function applications can have multiple instances of the same type, those instances usually come from the ability to promote type 0 into $\triangle^0$. This might create an expansion of the environment a constant number of times, but in terms of big O the complexity would stay the same.

It is important to note that the upper bound we have presented is reached only when evaluated expressions are not combined together. For example, we compute n maps, and drop $n-1$ of them. In all the other cases the combinators of evaluated expressions will generate constraints that will force us to cross out some columns of the environment. Let us consider the evolution of the environment for the following example:

```
f (a) =
    let b = map i < [N,N] a[i] in
        let c = map j < [N,N] b[j] in
            let d = map k < [N,N] c[k] in
            d
```

In map/reduce constructs, the main source of environment expansion is a scalar selection on a k type. For the first map the environment will look like:

$$
\begin{array}{r|cccccc}
a: & 0 & 1 & 1 & 2 & 2 & \triangle \\
i: & 0 & 0 & idx(1) & 0 & idx(2) & 0 \\
b: & 0 & 0 & 1 & 0 & 2 & \triangle
\end{array}
$$

The next map will not multiply the size of the environment by $\mathcal{S}_0(\mathrm{index}(j)) = 2$. Instead it will add two more columns, coming from scalar selections on k types:

$$
\begin{array}{r|cccccccc}
a: & 0 & 1 & 1 & 1 & 2 & 2 & 2 & \triangle \\
i: & 0 & 0 & idx(1) & idx(1) & 0 & idx(2) & idx(2) & 0 \\
b: & 0 & 0 & 1 & 1 & 0 & 2 & 2 & \triangle \\
j: & 0 & 0 & 0 & idx(1) & 0 & 0 & idx(2) & 0 \\
c: & 0 & 0 & 0 & 1 & 0 & 0 & 2 & \triangle
\end{array}
$$

So as the next map will add only two columns.

$$
\begin{array}{r|cccccccccc}
a: & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & \triangle \\
i: & 0 & 0 & idx(1) & idx(1) & idx(1) & 0 & idx(2) & idx(2) & idx(2) & 0 \\
b: & 0 & 0 & 1 & 1 & 1 & 0 & 2 & 2 & 2 & \triangle \\
j: & 0 & 0 & 0 & idx(1) & idx(1) & 0 & 0 & idx(2) & idx(2) & 0 \\
c: & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 2 & 2 & \triangle \\
k: & 0 & 0 & 0 & 0 & idx(1) & 0 & 0 & 0 & idx(2) & 0 \\
d: & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 2 & \triangle
\end{array}
$$

Here, the number of different typings did not grow exponentially with the number of maps. That is because maps have interdependencies, and a large number of variants got cancelled. Environment extension in this case is determined by the ability to make scalar selections on vectorised types.

As for constants, for most of practical cases, the number in a function is small. Also, every constant can be treated as if it were coming from an argument. That suggests that for practical purposes the environment size is mainly bound by the number of arguments.

4.5.4 Parameters of *idx* and $\triangle$ types

As we have said already the reason we introduce polymorphic parameters on $\triangle$ and *idx* layout types is to avoid iteration shadowing. However, there are further important properties. First of all, the $\triangle^0$ type helps us to differentiate between a replicated value and a "partial result" that has to be accumulated by some map/reduce. Although there is a known duality between types 0 and $\triangle^0$, we cannot get rid of the $\triangle^0$ type because of the following example:

```
# float[N] → float[N]
foo (a) =
    let b = map i < [N] 3 in
        map j < [N] a[j] + b[j]
```

If we want this function to be of type $(1) \to 1$ we have to allow selection `b[j]` with $j :: idx(1)$. In principle we could allow type-casting expressions of type 0 into some type k (given that k is within the dimensionality of the expression), but such a type-cast introduces memory copying which potentially is expensive.

Also, parameters allow us to avoid the following patterns:

```
foo (a, x) =
    ...
    map i < [N]
        if p (a[i]) then
            foo (a, a[i])
        else
            a
    ...
```

Such a pattern initiates transformations that our system is not yet ready to deal with, i.e. recursive map/reduce. However, the typing of the foo (in case $x :: \triangle$) will contain a concrete parameter in the signature. That means that whenever we apply `foo` in the context of map/reduce, we will have to unify two concrete $\triangle$ types resulting from two different map/reduces. As a consequence during the inference of a function definition, we can immediately reject non-polymorphic *idx* or $\triangle$ types, except $\triangle^0$.

4.6 Initial evaluation

In this section we are going to present some preliminary experiments to demonstrate the potential performance improvements that we can expect from the proposed framework. Extensive measurements can be found in Chapter 7.

The measurements we are going to present consist of two benchmarks: the N-body problem, our running example described in Fig 4.1, and the Mandelbrot problem.

The properties of those benchmarks are rather different. The N-body is both memory intensive and compute intensive involving an iterative update of a reasonably sized multi-dimensional array. Such a pattern can be found in many scientific applications such as solving partial differential equations numerically or other approximation problems. The Mandelbrot benchmark represents a class of applications where most of the execution time is spent on computations while the number of memory operations is very small. Also, the Mandelbrot benchmark requires a more complex pattern of vectorisation, as a computation of an individual element is expressed as a tail-recursive function which has to be vectorised in order to match the new layout. This requires additional effort when it comes to masking elements of individual vectors, and this pattern is not present in the N-body benchmark.

Name	**Description**
m-i3	"Intel (R) core (TM) i3-2310m CPU @ 2.10Ghz" processor, 2 cores with hyperthreading, equipped with AVX instruction set (8 floats per SIMD register), running Gentoo Linux with kernel version "3.14.4-gentoo" with GCC compiler version "Gentoo 4.8.2 p1.3r1, pie-0.5.8r1" and ICC version "14.0.3 20140422".
m-i7	"Intel (R) Core (TM) i7-2600 CPU @ 3.40GHz" processor, 4 cores with hyperthreading, equipped with AVX2 instruction set (8 floats per SIMD register), running Linux with kernel version "2.6.32-431.17.1.el6.x86_64" and GCC version 4.9.0.
m-xeon	Intel box with "Intel (R) Xeon (R) CPU X5650 @2.67Ghz", 12 cores with hyperthreading, equipped with SSE4.2 instruction set (4 floats per SIMD vector), running Linux with kernel version "2.6.32-431.17.1.el6.x86_64" and GCC version 4.9.0.

Figure 4.11: Machines used to produce the measurements

In the current set of experiments we want to verify two things:

1. The proposed inference does improve vectorisation, and
2. The effects of vectorisation are orthogonal to multi-threaded execution.

To verify the first statement the only thing that matters is the availability of SIMD instructions set on a CPU. For the second statement we require a CPU to have multiple cores. We use three different machines whose descriptions are presented in Fig 4.11. "m-i7" and "m-xeon" can be seen as typical nodes of a cluster which have four and 12 cores with hyperthreading accordingly. "m-i3" is a low-profile machine, but with a strong vectorisation capabilities. As a consequence our testbed allows us to observe behaviour on a server-type hardware and verify both the vectorisation and scalability, while experiments on "m-i3" put our vectorisation system in a more restrictive setup. This restrictiveness is very important for embedded or low-profile devices like smartphones, where vectorisation plays a crucial role in image or video processing applications. The SIMD instruction sets are different for all the three machines: AVX, AVX2 and SSE4.2.

All the machines used in this set of measurements are Intel-based. We use GNU GCC and Intel ICC compilers, and the perf tool to count a number of instructions via performance counters of a CPU. The runtime is measured by putting a timer around the core regions which excludes initialisation and output times. For every data point a minimum of 5 runs is being taken. We compile all of the benchmarks with the following compilation flags:

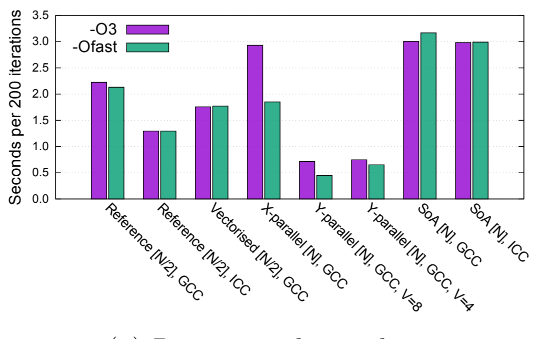
GCC `-Ofast -Wall -Wextra -mtune=native -march=native -std=gnu99 -fomit-frame-pointer -fopenmp -lm -lrt`

ICC `-gcc -Ofast -Wall -Wextra -mtune=native -march=native -std=gnu99 -fomit-frame-pointer -openmp -lrt`

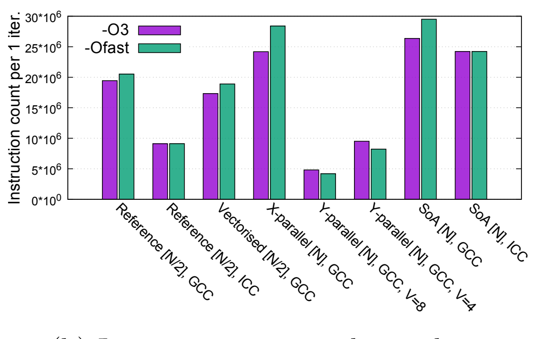
The transformation of the program is happening to a very high-level language which eventually has to be compiled down to some target language. The target language of SAC is C. In this set of experiments we are going to hand-code C programs which mimic the SAC output that we expect to get automatically by the SAC compiler. This should give us an idea of what runtimes we can expect excluding any potential overheads that SAC can bring. In order to mimic multi-threaded execution we are using OpenMP annotations.

4.6.1 N-body

We start with observing runtime and instruction count relationships of different implementations of the N-body on a single core in Fig. 4.12.



(a) Runtime relationship.



(b) Instruction count relationship.

Figure 4.12: Runtime and instruction count relationships for the N-body 1024 planets, 200 iterations on a "m-i3" machine.

On the left graph we can see runtime figures of the N-body with 1024 planets and 200 iterations altering the level of optimisation between `-O3` — the highest level of safe optimisations and `-Ofast` — the highest level of optimisations with unsafe math enabled. Unsafe math mainly allows us to reduce floating-point values in an arbitrary order. The right graph shows, for every measurement from the left graph, the number of instructions obtained by the "perf" tool. The unit of measure is instruction count per one N-body iteration which we obtain by: $\frac{Inst_{300} - Inst_0}{300}$, where $Inst_i$ is the number of instructions per i iterations of the N-body. The names of the benchmarks on Fig. 4.12 mean the following:

Reference [N/2] is a reference C implementation of the N-body benchmark as it can be found at the Debian Shootout[3]. Note that this benchmark uses the fact that the absolute value of the acceleration for (i, j) planets is the same as the acceleration for (j, i) planets, but has a different sign. We are going to mark such a solution with [N/2] postfix and the program that computes accelerations for all the pairs with [N] postfix. The version which is doing half of the computations makes a lot of sense in a single-threaded environment but makes it less favourable in a multi-threaded context, as the outer level parallelisation introduces large overheads because of scheduling complexity. See [129] for more details.

Vectorised [N/2] is the reference implementation vectorised across the inner axis.

X-parallel [N] is vectorised across the inner (x) axis, with SIMD vectors of length 4. It pads an array by adding dummy elements to the velocity triplet and position triplet.

Y-parallel [N] is a vectorised implementation across the outer (y) axis. It does not add dummy elements, so the amount of memory that planets take is the same as in the reference implementation. One important property of this benchmark is that it can efficiently use long SIMD vectors, as every vector stores individual components of different planets, where the X-parallel version requires one to pack individual triplets in the vector which is more expensive. That is why we prefix this benchmark with the length of the vector we use: either four elements (V=4) or eight elements (V=8).

SoA [N] is a reference implementation that computes all the pairs but which transforms arrays of structures into structures of arrays.

The key observation from Fig. 4.12 is that vectorisation across the outer axis performs the best and the main reason for that is the substantially smaller instruction count. Note that instruction count is not the most precise metric, as it does not directly correlate with runtime. In the case of Reference and X-parallel with `-Ofast` we have more instructions but better runtime. However, for Y-parallel the difference is too large to be ignored. As for the effects from turning on `-Ofast`, we can see that

[3]The Computer Language Benchmarks Game, see http://benchmarksgame.alioth.debian.org/ for more details.

GCC can do a better job, applying some vectorisation techniques and being able to change the order of reductions. As in the case of Y-parallel we change the order of the reduction anyway, we consider that an honest comparison would be in the case of `-Ofast`. We would assume that main effects of `-Ofast` are in the application of the vector variant of the square root (we do that explicitly in case of Y-parallel) and from doing vector reduction in a more efficient way, as we express it as a sum of components. Finally we can see that transforming an array of structures into a structure of arrays (SoA benchmark) does not have the same effect as our layout transformations (X-parallel and Y-parallel). Either none of the compilers recognised a potential for vectorisation or the locality effect resulted in disappointing performance. We are not going to consider SoA in further measurements.

Now, we would like to investigate how the implementations are going to behave in the presence of multithreading. First we are going to run the experiment on the Intel-based CPU with 4 hyperthreaded cores. We do not have the Intel compiler installed as well as “perf” software on this machine so we are going to start with single-core runtime relationships — see Fig. 4.13.

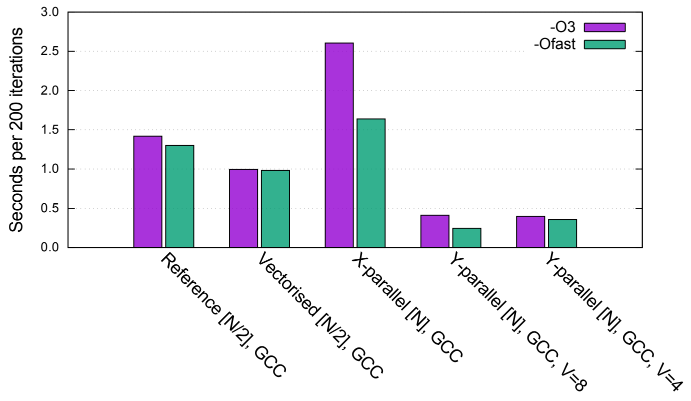
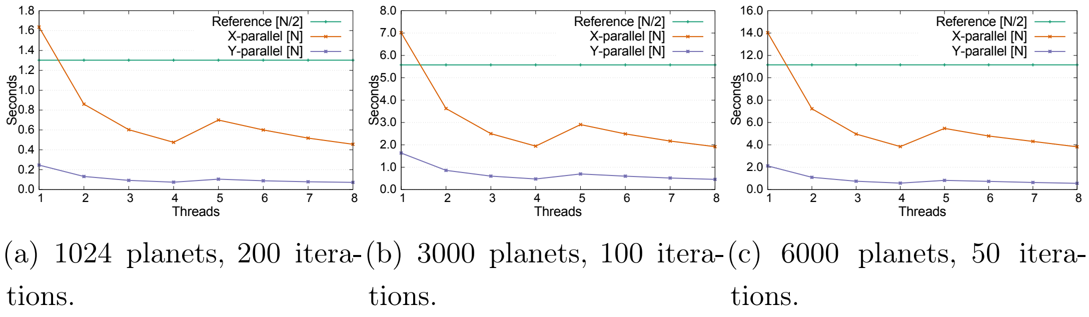


Figure 4.13: Runtime relations for the N-body 1024 planets, 200 iterations on the “m-i7” host.

We have a similar relationships as in Fig. 4.12. However, the difference between X-parallel and Y-parallel is higher: 4.5 on “m-i3” and 6.5 here. The reason could be either a newer instruction set or a newer compiler or faster memory or a combination of these.

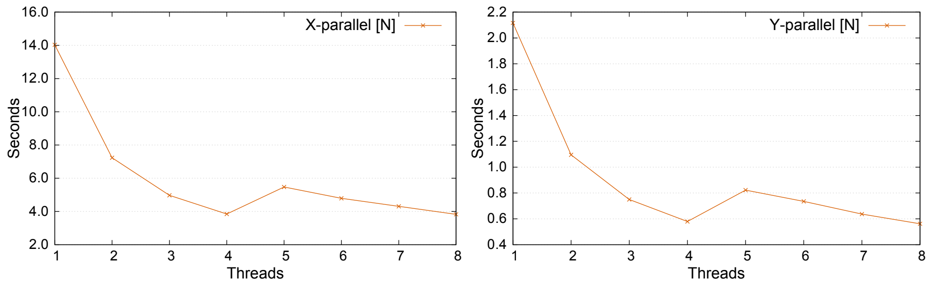


(a) 1024 planets, 200 iterations. (b) 3000 planets, 100 iterations. (c) 6000 planets, 50 iterations.

Figure 4.14: Scaling of the N-body on the machine described at Fig. 4.13

To observe the effects of multi-core presence, we are running three series of experiments increasing the size of the input data to observe if larger memory footprints influences the performance. The results are presented in Fig. 4.14.

The scaling across all the three figures is similar. The difference between X-parallel and Y-parallel decreases when the structure does not fit in the cache, but then it increases again in case of 6000 planets. From the graphs it might seem that the Y-parallel scales worse and in case of larger amount of cores the runtimes of the X-parallel and Y-parallel can merge, but this it is just the scale of the graph. In order to demonstrate that, we are going to look at individual graphs of the N-body with 6000 planets for X-parallel and Y-parallel versions. The runtimes are presented at Fig. 4.15.



(a) 6000 planets, 50 iteration, X-parallel.

(b) 6000 planets, 50 iteration, Y-parallel.

Figure 4.15: Individual scaling of X-parallel and Y-parallel N-body versions.

The scaling graphs are very similar, and the Y-parallel version scales better. Both benchmarks have a “jump” after 4 cores, which happens as there are only 4 physical cores, which means that in the case of 5 and more threads two threads will share one cache.

Finally, we measure the N-body scaling on the 24 core Intel machine (12 hyper-threaded CPUs) and see how it scales. See Fig. 4.16.

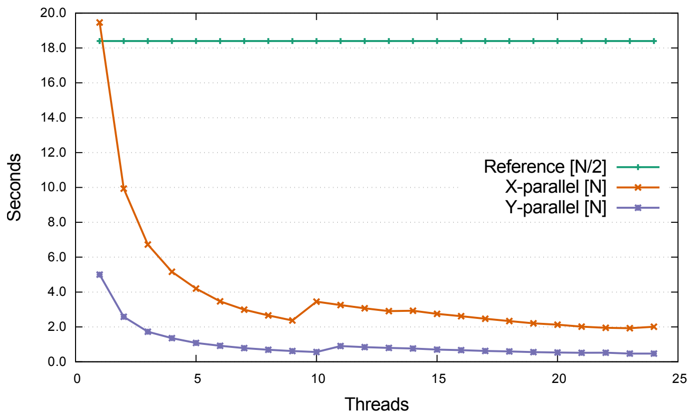


Figure 4.16: Scaling of the N-body implementations on the “m-xeon” machine.

This experiment confirms the scaling claim — having more CPUs does not merge the runtimes of X-parallel and Y-parallel. The Y-parallel is four times faster than the X-parallel on a single core and on 12 hyperthreaded cores.

4.6.2 Mandelbrot

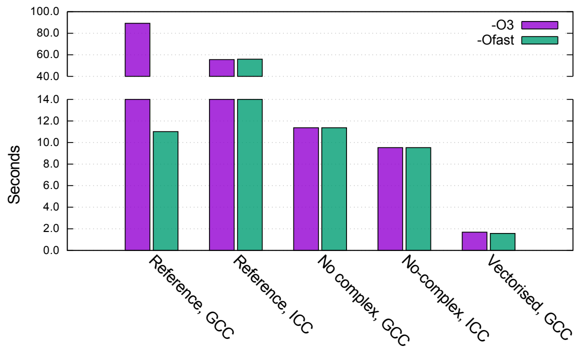
As a second example we look at a computation of Mandelbrot sets. The formulation of the Mandelbrot algorithm in SAC-λ, assuming that complex numbers are built-in and `<a, b>` denotes a complex constant $a + bi$, can be found in Fig. 4.17 on the left. We assume that DEPTH, HEIGHT, WIDTH, X1, Y1, DX and DY are compile-time constants.

```
letrec
  iter (i, z, a) =
    if i < DEPTH
       and cabsf (a) < 2
    then
      iter (i + 1, z * z + a, a)
    else
      i
in
  map i < HEIGHT
    map j < WIDTH
      iter (0,
            <0, 0>,
            <X1 + DX*j, Y1 + DY*i>)
```

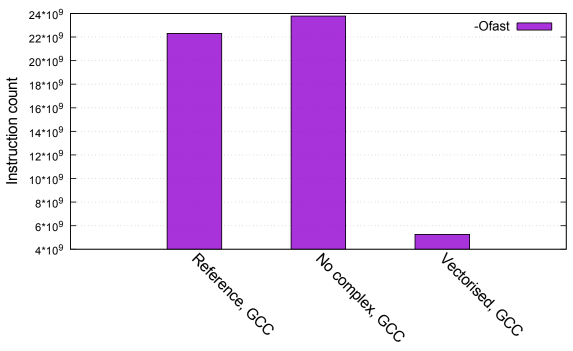
```
letrec
  iter (i, z, a) =
    if i < DEPTH and
       sqrt (z[0]*z[0]
             + z[1]*z[1]) < 2
    then
      iter (i + 1,
            [z[0]*z[0] - z[1]*z[1] + a[0],
             z[0]*z[1] + z[1]*z[0] + a[1]],
            a)
    else
      i
in
  map i < HEIGHT
    map j < WIDTH
      iter (0, [0, 0],
            [X1 + DX*j, Y1 + DY*i])
```

Figure 4.17: Formulation of the Mandelbrot problem in SAC-λ using built-in complex numbers on the left and using array-based representation of complex numbers on the right.

No matter if complex numbers are built-in or not, at some point they are going to be represented as two-element structures and scalar operations on complex numbers will be expressed via normal scalar operations. We demonstrate this in Fig. 4.17 on the right.



(a) Runtime relationship — *please mind the break in the graph.*



(b) Instruction count relationship.

Figure 4.18: Runtime relations for the Mandelbrot 2048×2048 floats with depth=4096 on the “m-i3” machine.

As in the case of the N-body we start with sequential runtime and instruction count relationships obtained on “m-i3” — see Fig. 4.18. The names of the benchmarks on Fig. 4.18 mean the following:

Reference is a literal translation of the formulation found on the left hand side of fig. 4.17 in C using float-based complex numbers.

No-complex is a literal translation of the formulation found on the right hand side of Fig. 4.17 in C using floats. We also get avoid the square root computation by translating $\sqrt{z_0^2 + z_1^2} < 2$ into $z_0^2 + z_1^2 < 4$.

Vectorised is a vectorisation of the No-complex function using a $(\triangle, \triangle, \triangle) \to \triangle$ layout type for the function `iter`, which means that we compute V iterations simultaneously. The function itself is simple, however the required code transformation is non-trivial — it requires creating data-flow masks with further predication and it all happens in the recursive context. The transformed code will look like this:

```
# (float[V], float[2,V], float[2,V]) -> float[V]
iter ($\vec{i}$, $\vec{z}$, $\vec{a}$) =
    let
        $\vec{m}$ = ($\vec{i}$ < $\vec{d}$) and ($\vec{z}$ * $\vec{\vec{z}}$ < $\vec{4}$);
    in
        if $\vec{m}$ == $\vec{false}$ then
            $\vec{i}$;
        else
            let
                $\vec{t_1}$ = iter ($\vec{i}+\vec{1}$, $\vec{z}$ * $\vec{z}$ + $\vec{a}$, $\vec{a}$);
            in
                select ($\vec{m}$, $\vec{t_1}$, $\vec{i}$)
```

where $\vec{x}$ corresponds to the variable x with its shape being replicated V times; $\vec{z} * \vec{\vec{z}}$ corresponds to vectorised version of $z_0^2 + z_1^2$; boolean operations are computed component-wise; multiplication and addition are overloaded for complex numbers and `select(m,a,b)` corresponds to `if m[i] then a[i] else b[i]` applied to all V vector components.

This is similar to N-body with respect to runtime and instruction relationships — the vectorised version runs much faster and has significantly less instructions. Note that here we have measured instruction counts in the overall program. This does include I/O operations, but their contribution to the overall instruction count is negligible. As for the Reference version, we can see that GCC compiler can bring it down to No-complex at `-Ofast`, where ICC fails to do so. We exclude this version from further measurements.

The scaling runtimes on "m-i7" can be found in Fig. 4.19. Note that we are using the default *dynamic* scheduler in OpenMP, as computations are non-uniform over the grid.

Scaling is much smoother than in the case of the N-body and we do not have a "jump" after 4 threads. That is because the Mandelbrot problem is more compute-bound than memory bound.

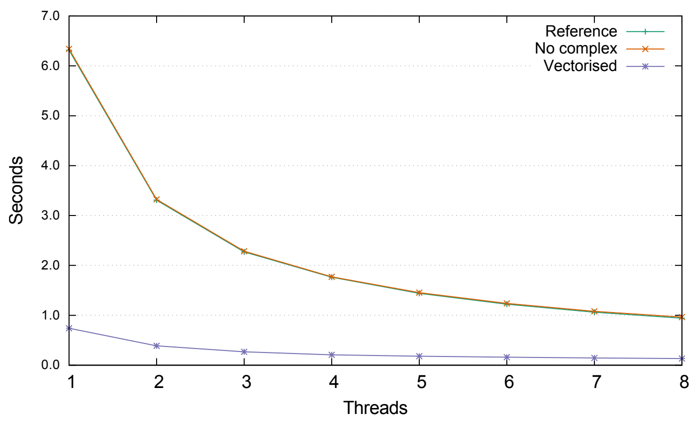


Figure 4.19: Scaling for the Mandelbrot implementations with 2048 × 2048 floats and depth=4096 on the "m-i7" machine.

Finally we present scaling figures on "m-xeon" to ensure that adding more cores does not make the runtimes merge. The results are presented in Fig. 4.20

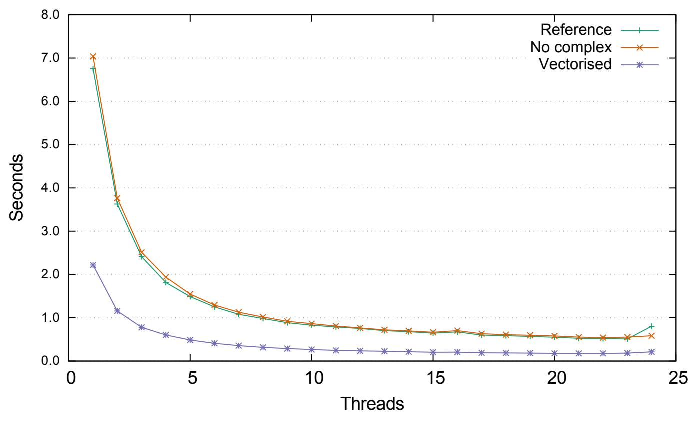


Figure 4.20: Scaling for the Mandelbrot implementations with 2048 × 2048 floats and depth=4096 on the "m-xeon" machine.

4.6.3 Summary

Both benchmarks have demonstrated very good speed-ups relative to the reference versions when compiling with GCC: 4.7x for the N-body and 7x for the Mandelbrot. Also, when comparing hand-coded vectorisations with the versions obtained from the Intel compiler with full optimisations we get: 2.8x for the N-body and 6.1x for the Mandelbrot. The speed-up is preserved in presence of multi-threaded execution which once again confirms that SIMD speed-ups are orthogonal to other parallelisation techniques and can be used together.

4.7 Related work

The idea to modify data layouts by means of compiler transformations is not new. There has been considerable work in the context of optimisations for improved cache behaviour [87, 69] and, more recently, for improved streaming through GPUs [85, 133]. In that work, improvements of spatial and temporal locality are the key goals. While this may seem to be a goal very similar to what we propose here, spatial locality is not sufficient for an efficient vectorisation, as we experienced in the N-body example — vectorisation across the inner dimension has better spatial locality than the variant we have inferred.

A number of works use the polyhedral model to optimise data layouts for a given loop nest. On the one hand, polyhedral transformation, being formulated in a mathematically rigorous way, takes away most of the considerations regarding transformation correctness due to the known correctness of polyhedra itself. On the other hand, it stays unclear how to apply such a transformation to the whole program, where different loop nests potentially grant different layouts. The information has to be propagated outside the loop-nests and the overall program has to be adjusted. Alternatively, layout changes can be done for the variables used in the given loop-nest: before entering the loop nest first, and restoring restoring original layouts on exit. Some just-in-time compilation techniques might be helpful here. However, in this case it is not clear how to justify performance penalty for memory copying operations.

In principle the work that we describe in this chapter can be formulated in terms of the polyhedral model. The map/reduce constructs have similarities to loop-nests and by trying to vectorise every loop in a loop nest using the "Direct outer loop vectorisation" technique from [97] with further checks if the vectorisation succeeded, we can generate layout types for the arrays referenced in this loop nest. Generated constraints have to be resolved and the polyhedral model would not help here. After that, the code has to be transformed which can be solved by the polyhedral model. However, the main weakness of such a technique in our set-up is the lack of support for function applications inside loop nests, and not all the functions can be inlined. That would have a serious impact when it comes to inferring the layout types for the overall program.

We plan to use this work as a basis for more complex vectorisations like grid and stencil computations by introducing new kinds of layout types, in which case we believe that our pure functional setup would allow more aggressive optimisations.

G. Chen et al. in [27] describe an approach which is very similar to ours. They also propose to infer data layouts of the arrays in a whole program. The main focus of their work is to formulate potential layouts for arrays as a constraint network and solve it. The layouts are defined as vectors in an N-dimensional space. The work is theoretical and the application is not described. There is no discussion about the selection of the layout-setting for the whole program assuming that constraint resolution returned a number of alternatives.

U. Bondhungula et al. in [16] present a polyhedral model based transformation for tiling loop nests for further parallelisation by means of OpenMP. The polyhedral framework described in their paper is able to handle sophisticated loop nests. However the transformation changes only the order of the iterations which might not be sufficient for efficient vectorisation. The transition from the inferred iteration order to the new layout is non-trivial, as the layouts have to match for the variables that are being reused. Also, the locality considerations used in this paper are not always sufficient for efficient vectorisation. At the example of the N-body we have seen that the solution with the best spatial locality grants less vectorisation opportunities than tiling across the first axis of a 2-d array, with subsequent reorganisation of the data structure.

K. Trifunovic et al. in [139] present a polyhedral based transformation for automatic vectorisation. Similar to [16], this work assumes that layouts are fixed and the transformation is substituting identical arithmetic operations with vector ones.

T. Hanretty et al. in [55] present a framework to optimise alignment conflicts caused by stencil computations. The key idea of the transformation is very similar to what we do — interchanging dimensions with further transposition. The main difference of the approaches is that in our work we are concerned that layout transformations for the sake of optimisation of a certain operation may have a negative effect on the overall performance. So we are concerned with generating all the potential program vectorisations and choosing the best overall. On the other hand, the transformation described in [55] would not currently be applicable in our setup as it uses operations in selection functions (i.e. `a[i - 1]`) that are not allowed by our type inference. However, by applying a preprocessing step on the stencil-like computations we can express it in the acceptable form for our inference system.

Roland Leißa et al. in [84] demonstrate a language which is an extension of C, that allows one to annotate data types which later are used by the type inference to infer and propagate vector operations using scalar code. The main use-case demonstrated in the paper is very similar to the N-body where vectors of triplets are vectorised over the individual component axis rather than over the whole structure. The main difference of the approaches is that we concentrate on an automatic inference of the layout without providing any annotations. Another difference is that we use multidimensional arrays instead of vectors of records.

P. Clauss and B. Meister in [31] present a framework to optimise the data locality of the loop-nest by rearranging data layouts of arrays. The transformation proposed in the paper, for a given loop-nest, generates new indexing functions for the dependent arrays such that iterations would access arrays sequentially in terms of the loop nest. This looks like an ideal solution from the theoretical point of view. However it is not clear how to solve the same problem for multiple loop-nests.

M. Kandemir and I. Kadayif in [69] propose to change memory layouts dynamically to achieve better locality in loop nests. This is an interesting approach which can be considered as a next step for our framework, as currently we explicitly avoid layout changes of any array at runtime. On the other hand, one can construct a situation where two expressions require one and the same array to have contradicting layouts. The main idea the technique proposes is to estimate during execution if changing a layout improves the cost of the loop-nest about to be executed, and if it does, perform a dynamic adjustment. In our case we would have to adjust the cost function, as we are not concerned with locality, but with runtime, which is harder to estimate. Also, the approach assumes that a program is a series of loop-nests joined by control-flow, which is not directly applicable in our setup.

4.8 Conclusions

In this chapter we advocate a novel systematic approach towards data layout transformation that enables vectorisation. This approach is motivated by the observation that many scientific codes have vectorisation potential that cannot be utilised due to an algorithm driven choice of data-layouts that is at odds with an effective vectorisation. We have demonstrated one such example, namely the naive N-body code and discussed why a straight-forward formulation leads to an unfavourable data layout.

Building on the N-body example, we have developed our approach towards a systematic inference of layout transformations. By means of a type system we abstract from all program details not relevant for a choice of data layouts. Using this abstraction facilitates not only the inference of layouts themselves but also guarantees the consistency of all inferred layouts.

We describe the type system as well as an inference algorithm in detail and we show how this identifies possible layout variations for our running example, the N-body code.

We have also provided some initial performance figures. Manually modified codes that reflect the inferred layout transformations show that substantial runtime improvements close to the vector-width of the chosen architecture are achieved over competitive C implementations of the N-body problem. They also show that these improvements are orthogonal to non-vector-based parallelisations that stem from the use of multi-core CPUs.

The orthogonality between vectorisation and multi-threaded parallel execution renders this work particularly powerful in the context of code generation for high-performance execution. The complexity of the program transformations that might be necessary to achieve the inferred layouts suggests that the full potential of this approach would be ideally realised through a fully automated, compiler driven process.

Chapter 5

Program transformation and proof of correctness

5.1 Introduction

We now present a formal layout-transformation scheme that takes a program annotated with layout types and transforms the program on the source level. This approach allows layout transformations to be implemented as a high-level AST to AST translation without requiring any adjustments to the low-level code generation.

Besides the type-driven code translation scheme we also provide a correctness proof of our transformation. This is based on a correctness notion that captures the essence of data-layout independence by looking at equality as being factored by index-space transformations of function arguments and function results.

5.2 Program transformation

After the type inference is done we get a set of vectorisation possibilities for a given program as a typing environment containing types which are n-element tuples. In order to perform a transformation one has to chose one column from an environment. Such a choice can be done either manually or by means of a cost model. For the time being let us assume that the choice is made — we will demonstrate a simple cost model in Section 6.2.1 in detail.

Even in terms of a single column of the environment, user-defined functions may be applied multiple times on arguments of different layout types, which may result into calls to different instances of the same function, i.e. to the different columns of the function environment. In other words, we are dealing with function overloading with respect to layout types. As all such overloadings can be resolved statically we start our transformation with a preprocessing step where we:

This chapter is based on the FHPC-2013 paper [127]

1. disambiguate function names for functions of different layout types and resolve the names in applications; and

2. annotate every sub-expression with its layout type

We demonstrate this process by example. Consider the following code:

```
P = letrec
      f (x) = e1
      g (x) = e2
    in
      f (g (c))
```

where c is some constant. Let us assume that the types for the functions of P are:

$$f :: \langle(\alpha_1) \to \beta_1, \dots, (\alpha_n) \to \beta_n\rangle$$
$$g :: \langle(\gamma_1) \to \delta_1, \dots, (\gamma_m) \to \delta_m\rangle$$

and the types for the body are:

$$c :: \langle\hat{\gamma}_1, \dots, \hat{\gamma}_p\rangle$$
$$f(g(c)) :: \langle\hat{\beta}_1, \dots, \hat{\beta}_p\rangle$$

where $\hat{\gamma}_i \in \{\gamma_1, \dots, \gamma_m\}$, and $\hat{\beta}_i \in \{\beta_1, \dots, \beta_n\}$. Choosing a vectorisation of P means to chose a column of the goal expression in P, i.e. a number $l \in \{1, \dots, p\}$. As a result, all the function applications in P will impose an instance of a function that is required at a particular application. Therefore, we can rewrite P into P_l as follows:

```
P_l = letrec
        f_1 (x :: α_1) = e_1 :: β_1
        ...
        f_n (x :: α_n) = e_1 :: β_n
        g_1 (x :: γ_1) = e_2 :: δ_1
        ...
        g_m (x :: γ_m) = e_2 :: δ_m
      in
        f_i (g_j (c :: γ_j)) :: β_i
```

After such preprocessing, Letrec and App can be used without further transformations, assuming that inner bodies of function definitions and the arguments of function applications are transformed. A Let expression requires its subexpressions to be transformed. That makes Const, Sel, Prf, If, Map and Reduce essential parts of the transformation.

Before we can formalise the transformation we introduce the new operators and notation we are going to use. First of all, we relate an array of layout type 0 and its transformation under layout type $k \in \mathbb{Z}_+$. We do this by introducing an index-translating function I_k which for every index in the array of layout type 0 yields an index in the k-vectorised array[1].

$$I_k(\langle[n],[i_1,\dots,i_n]\rangle) = \langle[n+1],[i_1,\dots,i_k \texttt{ div } V,\dots,i_n, i_k \texttt{ mod } V]\rangle$$

and the inverse variant of the same function:

[1]For our proofs we only require I_k to be bijective. As a consequence, our results could be generalised beyond the particular mappings that our layout type system considers.

$$I_k^{-1}(\langle [n], [i_1, \ldots, i_n] \rangle) = \langle [n-1], [i_1, \ldots, V i_k + i_n, \ldots i_{n-1}] \rangle$$

Secondly, we define the semantics of the helper operators. The $+\!\!+$ operation *concatenates* two arrays on the outer level, i.e. extending the first component of the shape. $+\!\!+$ works as follows:

$$[[1,2]] +\!\!+ [[3,4],[5,6]] = [[1,2],[3,4],[5,6]]$$

The following semantic rule defines the operation:

$$\frac{\overset{+\!\!+}{} \quad e_1 \Downarrow \langle [s_1, s_2, \ldots, s_n], [d_1, \ldots, d_p] \rangle \qquad e_2 \Downarrow \langle [s'_1, s_2, \ldots, s_n], [d'_1, \ldots, d'_q] \rangle}{s \equiv [s_1 + s'_1, s_2, \ldots, s_n] \qquad e_1 +\!\!+ e_2 \Downarrow \langle s, [d_1, \ldots, d_p, d'_1, \ldots, d'_q] \rangle}$$

Another operation is called *array stacking*, denoted with $\oplus$. In contrast to $+\!\!+$ it concatenates n arrays on the inner level, i.e. adding a new dimension at the end of shape vector. The application of $\oplus$ works as follows:

$$\oplus([1,2,3],[4,5,6]) = [[1,4],[2,5],[3,6]]$$

This is an n-ary operation defined using the following rule:

$$\frac{\overset{\oplus}{} \quad e_1 \Downarrow \langle [s_1, s_2, \ldots, s_m], [d^1_1, \ldots, d^1_p] \rangle \qquad \ldots \qquad e_n \Downarrow \langle [s_1, s_2, \ldots, s_m], [d^n_1, \ldots, d^n_p] \rangle}{s \equiv [s_1, \ldots, s_m, n] \qquad \oplus(e_1, \ldots, e_n) \Downarrow \langle s, [d^1_1, \ldots, d^n_1, \ldots, d^1_p, \ldots, d^m_p] \rangle}$$

This works similarly to concatenation, but adds an extra dimension and does concatenation on the last shape component rather than on the first one. In the rest of the paper we are going to use the following notation: $\oplus_{i=0}^{n} f(i)$ which is a shortcut for $\oplus(f(0), \ldots, f(n-1))$.

The slicing of an array e with the last index component fixed at j is denoted as $e[*, j]$. For example, $([[1,2,3],[4,5,6]])[*,1] = [2,5]$. In essence this is a reverse operation for $\oplus$ which allows to grab j-th component. Semantically it looks like this:

$$\frac{\text{SLICE} \quad e \Downarrow \langle [s_1, \ldots, s_{n-1}, s_n], [d_1, \ldots, d_{p \cdot s_n}] \rangle \qquad iv \Downarrow \langle [], [j] \rangle \wedge \ j < s_n}{s \equiv [s_1, \ldots, s_{n-1}] \qquad e[*, iv] \Downarrow \langle s, [d_{j+1}, d_{s_n+j+1}, \ldots, d_{(p-1)\cdot s_n + j + 1}] \rangle}$$

Intuitively stacking helps us to create an array of the layout type $\triangle$ out of V arrays of layout type 0. Slicing is a dual operation that allows one to retrieve one of the V components.

Now we define a program transformation formally using the $\mathcal{T}$ operator. It is applied to a typed program P after the preprocessing step defined above is done:

$$\mathcal{T}(P) = P'$$

where P' is a transformed program in SAC-λ. Note that in order to express vector operations, P' will use a set of built-in vector operations that are not allowed in P.

As function definitions are not first class expressions in SAC-λ, we introduce a version of $\mathcal{T}$ that can be applied to function definitions. We denote it with $\mathcal{T}_f$ and $\mathcal{T}_f($`foo`$)$ refers to the transformed function `foo`.

$\mathcal{T}$ is a one-shot top-down transformation defined recursively on a syntax structure of a program. We next present the definition of $\mathcal{T}$ for every term in SAC-λ.

5.2.1 Const

The base case of $\mathcal{T}$ is the transformation of constants. A constant c can get the following layout types:

0 in which case the constant stays the same as in the original program;

$1, \ldots, \mathcal{D}(c)$ in which case the data is being vectorised over the axis encoded by the layout type; and

$\triangle^0$ in which case the constant is being stacked V times.

Let us consider transformations for all three cases separately. Constant transformation under the layout type 0 is an identity function:

$$\mathcal{T}(c) = c \qquad c :: 0$$

Constant vectorisation under the layout type $\tau = k \in \mathbb{Z}_+$ is a cutting of an array c over its axis τ using tiles of size $1 \times V$ and putting V-sized chunks in such a way that the elements become adjacent in flattened representation (in our cases under $\mathcal{R}m$). Note that by doing so, we potentially increase the number of elements in the vectorised array, as the size of the τ axis might be not divisible by V. This would be a problem if we would ever make a reverse array vectorisation with further selection at newly introduced indexes. We never do that, and we never shuffle elements of the vector, so the new "phantom" elements of the array never participate in the original computations. The only thing that we should show is that index translations within the program stay sound.

To transform constants of the layout type $k \in \mathbb{Z}_+$ we use:

$$\frac{\begin{array}{c} s_c = [s_1, \ldots, s_n] \qquad s'_c = [s_1, \ldots, \lceil s_\tau / V \rceil, \ldots, s_n, V] \\ p' = (s_1 \cdot \cdots \cdot \lceil s_\tau / V \rceil \cdot \cdots \cdot s_n \cdot V) \\ c = \langle s_c, [d_1, \ldots, d_p] \rangle \qquad c' = \langle s'_c, [d'_1, \ldots, d'_{p'}] \rangle \\ \\ \forall k \in \{1, \ldots, p'\} \quad \left\{ \begin{array}{l} cidx' = \mathcal{R}m^{-1}(k, s'_c) \\ zero' = [\underbrace{1, \ldots, 1}_{n}, 0] \\ d'_k = \begin{cases} d_{\mathcal{R}m(I_\tau^{-1}(cidx'), s_c)} & I_\tau^{-1}(cidx') < s_c \\ d_{\mathcal{R}m(I_\tau^{-1}(cidx' \cdot zero'), s_c)} & \text{otherwise} \end{cases} \end{array} \right. \end{array}}{\mathcal{T}(c) = c' \qquad c :: \tau \in \{1, \ldots, \mathcal{D}(c)\}}$$

In this case multiplication and less than operators in

$$cidx' \cdot zero' \quad \text{and} \quad I_\tau^{-1}(cidx') < s_c$$

expressions are used per vector component. Note the way we treat "phantom" elements of the vectorised array — we fill the empty cells with the first element of each vector, which should be always present, as zero axes sizes are not supported. As a result, any vectorised operation on such a vector succeeds, if the corresponding scalar operation succeeds in the original program. However, if the original program terminates with a hardware exception due to invalid arguments of a certain operation, this behaviour is going to be preserved.

Finally we consider the case when c is of layout-type $\triangle^0$ which means that the array has to be stacked V times.

$$\mathcal{T}(c) = \oplus_{i=0}^{V} c \qquad c :: \triangle^0$$

5.2.2 SEL

Selections can manifest themselves in three different ways: scalar selection, vector selection and scalar selection on a vectorised array. To define the selection transformation formally, we introduce a primitive operation for *vector selection* called **vsel**. Semantically, vector selection is a concatenation of V scalar selections on the last dimension of a vectorised array. Keep in mind that the length of an index vector used in vector selection is one element shorter than the dimensionality of the vectorised array.

$$\textsc{vsel}\;\frac{iv \Downarrow \langle [n-1], [i_1, \ldots, i_{n-1}] \rangle \qquad e \Downarrow \langle \langle [n], [s_1, \ldots, s_{n-1}, V] \rangle, [d_1, \ldots, d_m] \rangle}{\begin{array}{c}\mathbf{vsel}(iv, e) \Downarrow \langle [V], [d'_1, \ldots, d'_i] \rangle \\ \text{where } \forall i \in \{0, \ldots, V-1\} : \langle [], [d'_i] \rangle = \mathbf{sel}(iv +\!\!+ [i], e)\end{array}}$$

Selection transformation is defined as:

$$\mathcal{T}(\mathbf{sel}(iv, a)) = \begin{cases} \mathbf{sel}(\mathcal{T}(iv), \mathcal{T}(a)) & iv :: 0 \wedge a :: 0 \\ \mathbf{vsel}(\mathcal{T}(iv), \mathcal{T}(a)) & iv :: idx(k) \wedge a :: k \\ \mathbf{vsel}(\mathcal{T}(iv), \mathcal{T}(a)) & iv :: 0 \wedge a :: \triangle \\ \mathbf{sel}(I_k(\mathcal{T}(iv)), \mathcal{T}(a)) & iv :: 0 \wedge a :: k \end{cases}$$

As it alway is, for layout types 0 the transformation is an identity function. Vector selections can happen in two cases:

1. $a :: k \wedge iv :: idx(k)$, which means that this selection is invoked within a map/reduce operation, and the index space is vectorised over the k-th axis. As an example consider a map over a 1-d array where the inner operation uses selection:

```
map i < [N] f (a[i])
```

in which case if $i :: idx(1) \wedge a :: 1$, the selection has to return an expression of vector layout type rather than scalar;

2. $a :: \triangle \wedge iv :: 0$, which can be found in nested maps, when selection happens on the axis which was not vectorised. For example, assume we have a 2-d array a of shape $[M, N]$:

```
map i < [M]
  let
      line = map j < [N] a[i ++ j]
  in
      f (line[iv])
```

in this case, assuming that $i :: idx(1)$, j can be only of type 0, and the `line` has to become a 1-d array of vectors rather than of scalars, which means that `line` gets a layout type $\triangle$. Any further selections on `line` implies vector selections with an index of layout type 0.

Finally, when selections on a vectorised array are performed outside the map/reduce operation, the result of the selection has to be scalar. To illustrate that assume a data-parallel operation on the array a after which one scalar element is being accessed:

```
let
    r = map i < u f(a[i])
in
    g (r[e])
```

Assuming that $a :: k \wedge r :: k \wedge e :: 0$, we adopt index e by applying the I_k index transformation.

5.2.3 Primitive functions PRF

All the binary primitive scalar operations have built-in vector variants, so the only thing the transformation should take care of is vectorisation of the scalar component, which is allowed by the layout-type system. Without loss of generality we consider the plus operator as being a representative of all binary primitive functions in SAC-λ.

$$\mathcal{T}(a+b) = \begin{cases} \mathcal{T}(a) + \mathcal{T}(b) & a :: 0 \wedge b :: 0 \\ (\oplus_{i=0}^{V} \mathcal{T}(a)) \vec{+} \mathcal{T}(b) & a :: 0 \wedge b :: \triangle \\ \mathcal{T}(a) \vec{+} (\oplus_{i=0}^{V} \mathcal{T}(b)) & a :: \triangle \wedge b :: 0 \\ \mathcal{T}(a) \vec{+} \mathcal{T}(b) & a :: \triangle \wedge b :: \triangle \end{cases}$$

where $\vec{+}$ is a vector variant of $+$. Note that here the indexes of $\triangle$ types are irrelevant, as their correctness has been established by the layout type inference. As a consequence, if any subexpression is of a $\triangle$ layout type, a scalar plus is transformed into a vector plus.

5.2.4 MAP

To formalise map/reduce transformations we introduce a function U_k that updates an upper-bound when the index of map/reduce is of layout-type $idx(k)$.

$$U_k(\langle [n], [u_1, \dots, u_n] \rangle) = \langle [n], [u_1, \dots, \lceil u_k/V \rceil, \dots, u_n] \rangle$$

The application of U_k to some e is very similar to the result of $I_k(e)$. The only difference is that we omit the last vector component. We do this as vectorised selection operates on n-element indexes, selecting from arrays of $(n+1)$-element shapes.

We apply U_k when the index vector of the map is of layout type $idx(k)$. In all the other cases we only propagate the transformation into subexpressions.

$$\mathcal{T}(\textbf{map}\ j < u\ e) = \begin{cases} \textbf{map}\ j < U_k(\mathcal{T}(u))\ \mathcal{T}(e) & j :: idx(k) \wedge e :: \triangle \\ \textbf{map}\ j < \mathcal{T}(u)\ \mathcal{T}(e) & \text{otherwise} \end{cases}$$

5.2.5 REDUCE

The only case when **reduce** requires a transformation is, similarly to **map**, when the layout type of its index vector is $idx(k)$ and the inner expression is of layout type $\triangle$. The upper bound expression has to be transformed in the same way as in **map**. However, in contrast to **map**, the resulting layout type of **reduce** has to be 0. This does not happen automatically as a return layout type of a reduction function is $\triangle$, which implies an extra step of reduction. First we consider the case when the axis of vectorisation is divisible by V i.e. there are no "phantom" elements:

```
T(reduce i < u (f) e) =
    let
        r = reduce i < U_k(T(u)) (T_f(f)) T(e)
    in
        reduce i < [V] (f) r[*,i]
```

In order to see this transformation in action, consider a summation (with $+$) of a 1-dimensional array a of shape $[N \cdot V]$:

```
reduce i < [N·V] (+) a[i]
```

Assuming that a has layout type 1 and i is of layout type $idx(1)$, the following vectorisation

```
r⃗ ≡ reduce i < [N] (+⃗) a[i]
```

will be of shape $[V]$ which has to be reduced with a scalar plus:

```
reduce i < [V] (+) r⃗[i]
```

Now let us consider the case when the size of the vectorisation axis is not divisible by V. We cannot perform a vector operation on the last vector (an element of vectorised array which has $\lceil s_k/V \rceil$ at its k-th position in the index), as it will contain "phantom" elements. For example, if $V \equiv 4$ then the vectorised array $[1,2,3,4,5,6,7,8,9]$ will look like $[[1,2,3,4],[5,6,7,8],[9,x,x,x]], x = 9$ — we cannot add the last vector, but we can still add the first two using $\vec{+}$.

Using this observation we split the index space of vectorised reduce into two parts:

$$[0,\ldots,0] \le i < [u_1,\ldots,\lfloor u_k/V \rfloor,\ldots,u_n]$$

and

$$[0,\ldots,\lfloor u_k/V \rfloor,\ldots,0] \le i < [u_1,\ldots,\lceil u_k/V \rceil,\ldots,u_n]$$

Formally we denote that as follows:

```
$\mathcal{T}$(reduce i < $u$ ($f$) $e$) =
    let
        $\vec{r}$ = reduce i < [$u_1^T,\ldots,u_k^T$ div $V,\ldots,u_n^T$] ($\mathcal{T}_f(f)$) $\mathcal{T}(e)$;
        $\vec{r'}$ = reduce i' < [$u_1^T,\ldots,$(if $u_k$ mod $V == 0$ then 1 else 0)$,\ldots,u_n^T$] ($\mathcal{T}_f(f)$)
                  let
                      i = [$i'_1,\ldots,i'_k + (u_k^T$ div $V),\ldots,i'_n$]
                  in
                      $\mathcal{T}(e)$
    in
        let
            $r_1$ = reduce i < [$V$] ($f$) $\vec{r}$[*, i];
            $r_2$ = reduce i < [$u_k^T$ mod $V$] ($f$) $\vec{r'}$[*, i];
        in
            f($r_1$, $r_2$)
```

where $u^T = \mathcal{T}(u)$ and u_i^T refers to the components of the vector u^T.

In order to illustrate the final formula, consider a reduction with plus over an array of shape $[10, N]$ being of layout-type 1. In this case $\vec{r}$ would hold a result of vectorised addition of vectors $[0,*,*]$ and $[1,*,*]$. Then $\vec{r'}$ would sum-up vectors $[2,*,*]$, using vector plus. Note that we can do this, as an operation on "phantom" elements would not affect the result. Finally, r_1 would sum-up the elements inside the vector and r_2 would sum-up non-phantom elements of $\vec{r'}$. Also, when u_k is a multiple of V, the index range of $\vec{r'}$ will be empty, and $\vec{r}$ will get a vectorised neutral element, which later will be reduced to the scalar neutral element.

Finally, when reduce is of layout type $\triangle$ or k the transformation is defined as:

```
$\mathcal{T}$(reduce $i < u$ ($f$) $e$) = reduce $i < u$ ($\mathcal{T}_f(f)$) $\mathcal{T}(e)$
```

5.2.6 IF

The transformation of a condition is the most complicated part of all the transformations we present in this work. There are two reasons for that:

1. vectorised conditions require control-flow to data-flow transformations; and

2. the branches of a conditional may contain recursive function calls.

The first problem has been studied in [122, 70, 26]. The proposed solution is to evaluate masks for true and false branches from the predicate of a conditional and propagate those masks into the branches. Such an approach potentially introduces a large number of masking operations which can have a negative impact on the overall performance. Our context offers us the following trade-off: instead of masking every operation inside the branches while evaluating it, we can evaluate an expression first and then mask out unnecessary results. This idea is going to guide our transformation.

Assume that the condition we want to transform has the following form:

$$e = (\textbf{if } p :: \alpha \textbf{ then } e_1 :: \beta \textbf{ else } e_2 :: \gamma)$$

Ideally, we would like to apply the following transformation:

$$\mathcal{T}(e) = \begin{cases} \textbf{if } \mathcal{T}(p) \textbf{ then } \mathcal{T}(e_1) \textbf{ else } \mathcal{T}(e_2) & \alpha = 0 \land \beta = \gamma \land \beta, \gamma \neq \triangle \\ \text{mask}\ (\mathcal{T}(p), \mathcal{T}(e_1), (\oplus_{i=0}^{V} \mathcal{T}(e_2))) & \alpha = \triangle \land \beta = \triangle \land \gamma = 0 \\ \text{mask}\ (\mathcal{T}(p), (\oplus_{i=0}^{V} \mathcal{T}(e_1)), \mathcal{T}(e_2)) & \alpha = \triangle \land \beta = 0 \land \gamma = \triangle \\ \text{mask}\ (\mathcal{T}(p), \mathcal{T}(e_1), \mathcal{T}(e_2)) & \alpha = \triangle \land \beta = \triangle \land \gamma = \triangle \\ \text{mask}\ ((\oplus_{i=0}^{V} \mathcal{T}(p)), \mathcal{T}(e_1), (\oplus_{i=0}^{V} \mathcal{T}(e_2))) & \alpha = 0 \land \beta = \triangle \land \gamma = 0 \\ \text{mask}\ ((\oplus_{i=0}^{V} \mathcal{T}(p)), (\oplus_{i=0}^{V} \mathcal{T}(e_1)), \mathcal{T}(e_2)) & \alpha = 0 \land \beta = 0 \land \gamma = \triangle \\ \text{mask}\ ((\oplus_{i=0}^{V} \mathcal{T}(p)), \mathcal{T}(e_1), \mathcal{T}(e_2)) & \alpha = 0 \land \beta = \triangle \land \gamma = \triangle \end{cases}$$

where mask works as:

$$\text{mask}\,(m, e_1, e_2) = \oplus_{i=0}^{V}\,(\textbf{if } m[i] \textbf{ then } e_1[*, i] \textbf{ else } e_2[*, i])$$

A real implementation might differ from the above definition as most of the architectures support vector selections in which case instead of combining slices we can combine individual vectors. Note that `m` in this case is a vector of shape $[V]$, as predicates in non-vectorised conditions always evaluate to scalar booleans. Expressions e_1 and e_2 can be non-scalar arrays.

By replacing a condition with a mask operation, we change the normal order of evaluation (in case of the original if) into applicative order. This has two consequences:

1. we can introduce hardware exceptions that were not present in the original program; and

2. we can break the termination of a program and make turn some programs that terminate originally into non-terminating programs after the transformations is applied.

Hardware exceptions The first problem can be understood from the following example:

```
map i < [N] if a[i] == 0 then 0 else 1/a[i]
```

When the condition is vectorised, the else branch evaluation may result in a hardware exception due to division by zero. This sort of exception happens not because the original program were generating it but due to the way SIMD instructions work. To deal with this we need to rewrite the expression in the branches propagating the mask from the condition. Whenever a special operation like division of integers, or square root is found, we need to mask the arguments with a neutral element for the given operation. Implementation of such an analysis can be done straightforwardly, but for the sake of simplicity assume that the behaviour of transformed and non-transformed code is the same with respect to hardware exceptions. In case it is not statically decidable we reject the vectorisation.

Recursive calls The second problem happens when one of the branches contains a recursive call. In this case the above transformation may lead to unbound recursion as the function call is taken out of the condition branch. For example:

```
foo (x) =
    ...
    if (p (x)) then
        foo (x)
    else
        e
```

is transformed into:

```
$\vec{foo}$ ($\vec{x}$) =
    ...
    let
        t = $\vec{foo}$ ($\vec{x}$);
        f = $\vec{e}$
    in
        mask ($\vec{p}$ ($\vec{x}$), t, f)
```

It can be seen the function $\vec{\texttt{foo}}$ will never terminate, when `foo` itself might. To avoid such cases, we have to identify recursive cycles and guard a recursive call with a condition. Our type inference does not allow for unbound recursion, so functions participating in a recursive cycle will have a case like that mentioned above. Our language is first order and anonymous functions are not allowed so it is straight forward to identify recursive cycles. After that at every exit from the recursive cycle we need to guard the non-exit call. For example:

```
foo (..., x, ...) =
    ...
    if (p (x)) then
        foo (..., f (x), ...)
    else
        e
```

will be transformed into

```
$\vec{foo}$ (..., $\vec{x}$, ...) =
    ...
    let
        m = $\vec{p}$ (x)
    in
        if any (m) then
            let
                t = $\vec{foo}$ (..., $\vec{f}$ ($\vec{x}$), ...);
                f = $\vec{e}$;
            in
                mask (m, t, f)
        else
            $\vec{e}$
```

In this case `any` refers to the disjunction of all the elements of the vector `m`:

$$\text{any}(e) = \bigvee_i e_i \qquad\qquad \text{any}([m_1,\dots,m_V]) = \bigvee_{i=1}^{V} m_i$$

Finally, we have to make sure that the recursive call of $\vec{\texttt{foo}}$ can be safely called on the values of $\vec{f}$ ($\vec{\text{x}}$) that are masked by the current value of the mask and that are never passed to the original `f`. In case the hardware supports predication, we can pass the mask to the call of $\vec{\texttt{f}}$. In other case we require the following property to hold:

$$\neg p(x) \implies \neg p(f^*(x))$$

where $f^*(x)$ means any number of function compositions. Informally this property means that if we took an exit on some value, then subsequent recursive calls will still result in exit. Such an analysis can be tricky, however, it has a lot of similarities with termination analysis that is usually done by theorem provers [89] or in the context of embedded or reactive systems [35, 17]. For practical purposes, the most common recursion we deal with is tail recursive functions that represent loops. There we usually deal with one induction variable and its increment, and the predicate is a comparison with a bound:

```
# for (i = 0; i < N; i++)
for_loop (i, ...) =
    ...
    if i < N then
        for_loop (i+1, ...)
    else
        ...
```

in which case the required property holds.

Non-termination follow-up Besides the cases when non-termination is introduced because of badly-treated recursion, non-termination may be introduced by evaluating the cases of a conditional which are never evaluated in the original program. Consider the following scenario:

```
infinite (x) =
    if (1 == 1)
        infinite (x)
    else
        0
```

This function never terminates. However, the calling site of such a function may be constructed in such a way, that it never occurs. For example:

```
foo (x) =
    let
        y = x * x
    in
        if (y > 0) then
            y
        else
            infinite (x)
```

This is very unlikely to happen in practice, but in case we want to handle a vectorised version of such a function we need to have the following transformation:

```
$\vec{\text{foo}}$ ($\vec{x}$) =
    let
        $\vec{y}$ = $\vec{x}$ * $\vec{x}$;
        m = $\vec{y}$ > [0, ..., 0]
    in
        if (all (m)) then
            $\vec{y}$
        else if (not any (m)) then
            $\vec{\text{infinite}}$ ($\vec{x}$)
        else
```

```
mask (m, y⃗, infinite⃗ (x⃗))
```

In this case `all` refers to conjunction of all the elements of the vector `m`:

$$\mathrm{all}(e) = \bigwedge_i e_i \qquad\qquad \mathrm{all}([m_1, \dots, m_V]) = \bigwedge_{i=1}^{V} m_i$$

Also, in this particular case one might need to implement mask as a series of scalar ifs, as $\overrightarrow{\text{infinite}}$ might be triggered by a specific value only. Generally speaking, in this work our primarily goal is to deal with tail-recursive representation of loops and our termination analysis is fairly trivial. In case we cannot prove termination of the functions in the branches of a conditional, we implement vectorised condition via series of scalar ifs.

5.2.7 Fundef, app, let and letrec

For the transformation of these expressions we propagate $\mathcal{T}$ into its subexpressions, assuming that the preprocessing step described in the beginning of this section is done.

Transformation of a function definition is transformation of its body expression:

$$\mathcal{T}_f\,(f_i(a_1, \dots, a_n) = e) \;=\; (f_i(a_1, \dots, a_n) = \mathcal{T}(e))$$

The transformation of a function application requires the arguments to be transformed, as the layout-type overloading has been resolved during the preprocessing step:

$$\mathcal{T}(f_i(e_1, \dots, e_n)) \;=\; f_i(\mathcal{T}(e_1), \dots, \mathcal{T}(e_n))$$

The same reasoning is applicable to let expressions:

$$\mathcal{T}(\textbf{let } x = e_1 \textbf{ in } e_2) \;=\; (\textbf{let } x = \mathcal{T}(e_1) \textbf{ in } \mathcal{T}(e_2))$$

Finally the letrec construct is a vectorisation of its goal expression and all the fundefs according to the new layout-type signatures.

$$\begin{aligned}\mathcal{T}(&\textbf{letrec } f_1(\dots) = e_1, \dots, f_n(\dots) = e_n \textbf{ in } e)\\ &= \textbf{letrec } \mathcal{T}_f(f_1(\dots) = e_1), \dots, \mathcal{T}_f(f_n(\dots) = e_n) \textbf{ in } \mathcal{T}(e)\end{aligned}$$

5.2.8 Transformation summary

All the variables in the original programs can be found in the transformed program, as Fundef leaves the names of the arguments unmodified. The same holds for Let expressions.

The way we formulate the transformation of conditions does not necessarily lead to the best possible performance. For example, vectorisation of deeply nested conditions may perform worse than V original ones, when each branch implies a sufficient amount of work. On the other hand, switching between scalar and vector modes or predicating each operation with a mask has negative performance effects as well. In order to tell for sure, we have to employ a cost model that is aware of the underlying hardware and the cost of memory operations. For the time being, intuitively the proposed transformations delivers speed-ups under the following assumptions:

1. Vector and scalar arithmetic instructions operate in the same time; and

2. For vectorised conditions minimum computations is being discarded, i.e. execution of true and false branches is distributed evenly, or the less preferred branch does not imply heavy computations.

Note that the transformation of conditions implicitly requires the shapes of the expressions evaluated in both branches to be equal (because of the definition of the $\oplus$ operation). This restriction prevents our system from vectorising functions similar to filter:

```
filter (vec, x, res) =
    if len (vec) == 1 then
        if vec[0] < x then
            [vec[0]] ++ res
        else
            res
    else
        if vec[0] < x then
            filter (drop ([1], vec), x, [vec[0]] ++ res)
        else
            filter (drop ([1], vec), x, res)
```

That is because masking `[vec[0]] ++ res` and `res`, when both have layout type $\triangle$ would result in a two-dimensional array with rows of different sizes. Although it might be possible to support this in principle, for the context of this thesis we disregard such cases.

5.3 Transformation correctness

After we have introduced the transformation we would like to make sure that transformed programs evaluate the same values up to the layout mappings. In order to verify this, first of all, we introduce correctness criteria for transformed programs of a given type.

The body of a transformed program can only evaluate an expression of the $k \in \mathbb{N}$ layout type. That is because the existence of $\triangle$ or *idx* types suggests a map/reduce context and as our programs are not allowed to have unbound variables, *idx* and $\triangle$ types cannot appear in the resulting type of the goal expression of a program. For those cases if

$$e' = \mathcal{T}(e) \qquad e :: k \in \mathbb{N}$$

we say that program transformation is correct if we can demonstrate an element-wise correspondence of two arrays:

$$\text{Eval}(e') = \mathcal{T}(\text{Eval}(e)) \qquad \text{and} \qquad \text{Eval}(e) = \mathcal{T}^{-1}(\text{Eval}(e'))$$

In order to do so we will use index-mappings I_k and I_k^{-1} as follows:

$$\forall_{j<s_e} e[j] = e'[I_k(j)] \qquad \text{and} \qquad \forall_{I_k^{-1}(j)<s_e} e'[j] = e'[I_k^{-1}(j)]$$

where the quantification $\forall_{j<s_e}$ denotes the range of all valid indexes of the original expression. From the construction of I_k and I_k^{-1} it is enough to show only one correspondence, as the other will be directly deducible.

While this appears to be a straight-forward criterion, an inductive proof requires more formalism as we have to deal with expressions of arbitrary layout types including our special types $\triangle$ and $idx(k)$. The main difficulty with those types is that they represent partial results appearing inside of a map/reduce. Evaluated expressions

$$e \text{ and } \mathcal{T}(e) \qquad e :: \triangle$$

are not comparable with any equality criterion as $\mathcal{T}(e)$ carries V instances of e coming from the different iterations of a map/reduce. To solve this problem we are going to generalise the correctness criterion of the arithmetic operations:

$$\vec{f}([x_1, \ldots, x_V]) = [f(x_1), \ldots, f(x_V)]$$

and prove that the transformed expression can be represented with V instances of the original expression. For *idx* types we have to show that mappings from the original index space into the vectorised index space holds.

Layout types $idx(k)$ First of all, the *idx* layout types are being attributed only to index vectors. The transformation $\mathcal{T}$ maps a tuple of V indexes in the original program into a single index in the transformed program. To illustrate this, assume that we have a set of indexes

$$I = \{\imath_1, \ldots, \imath_n\} \qquad \text{and a transformed set} \qquad \mathcal{T}(I) = \{\vec{\imath}_1, \ldots, \vec{\imath}_m\}$$

A bijective mapping M maps tuples of V indexes from I into indexes from $\mathcal{T}(I)$:

$$M(\langle \hat{\imath}_1, \ldots, \hat{\imath}_V \rangle) = \hat{\vec{\imath}}_j \qquad \text{where} \quad \forall_{k=1}^{V} \hat{\imath}_k \in I \quad \text{and} \quad \hat{\vec{\imath}}_j \in \mathcal{T}(I)$$

The mapping M does not map arbitrary tuples of indexes to the vectorised set. Instead we define a function that groups indexes from I into tuples of V indexes so that G is a partition of I. For convenience we define G to operate on the elements of I and to return a tuple where the given element belongs to:

$$G(\imath_j) = \langle \hat{\imath}_1, \ldots, \hat{\imath}_V \rangle \qquad \exists k \in \{1, \ldots, V\} \; \hat{\imath}_k = \imath_j \wedge \forall_{k=1}^{V} \hat{\imath}_k \in I$$

For the expression e of layout type $idx(k)$ and its transform $e' = \mathcal{T}(e)$ we need to show that:

$$G(e) = M^{-1}(e') \qquad \text{and} \qquad e' = M(G(e))$$

G is defined as:

$$G(e) = \langle e\{e_k \to e_k - e_k \,\texttt{mod}\, V\}, e\{e_k \to e_k - e_k \,\texttt{mod}\, V + 1\}, \\ \dots, e\{e_k \to e_k - e_k \,\texttt{mod}\, V + V - 1\}\rangle$$

where $e\{e_k \to x\}$ denotes that the k-th component of e has been substituted with x.

The M function is defined as:

$$M(\langle \hat{\imath}_1, \dots, \hat{\imath}_V\rangle) = \hat{\imath}_1\{\hat{\imath}_1[k] \to \hat{\imath}_1[k] \,\texttt{div}\, V\}$$

where $\hat{\imath}_1[k]$ refers to the k-th component of the $\hat{\imath}_1$.

The reverse M^{-1} function is defined as:

$$M^{-1}(\vec{\imath}) = \langle \vec{\imath}\{\vec{\imath}_k \to V\vec{\imath}_k\}, \vec{\imath}\{\vec{\imath}_k \to V\vec{\imath}_k + 1\}, \dots, \vec{\imath}\{\vec{\imath}_k \to V\vec{\imath}_k + V - 1\}\rangle$$

Note that the following transformation holds due to the nature of the `mod` and `div` operations:

$$M(G(e)) = e\{e_k \to e_k \,\texttt{div}\, V\}$$

Note that the boundaries of the array G return a tuple that contains indexes which are illegal in terms of the original program. However, as indexes of *idx* layout type will be used in selections into vectorised arrays (of layout type k) those indexes will become valid, as k-type transformation pads the array. On the other hand, to match the padding policy we can consider the following behaviour of G in case e is at the boundary:

$$G([e]) = \langle [e - e \,\texttt{mod}\, V], [e - e \,\texttt{mod}\, V + 1] \dots, [e], \dots, [e]\rangle$$

to replicate the last value $V - e \,\texttt{mod}\, V$ times. In that case if $g = G([e])$, then:

$$\mathbf{vsel}(M(g), \mathcal{T}(a)) = [\mathbf{sel}(g_1, a), \dots, \mathbf{sel}(g_V, a)]$$

Layout types $\triangle$ In order to prove that the transformation of expressions of layout type $\triangle$ is correct we want to ensure that it is safe to apply them in the context of a vectorised map/reduce. To show this we are going to consider an expression $e :: \triangle$ being a function $f(a_1, \dots, a_n)$ where arguments a_i are free variables of e. The transformation of e is:

$$\mathcal{T}(e) = e' = \vec{f}(a'_1, \dots, a'_n)$$

in which case we need to demonstrate how the application of $\vec{f}$ can be replaced with applications of the original f. To do so we show that:

$$\oplus_{j=1}^{V} f(\mathcal{T}^{-1}(a'_1, j), \dots, \mathcal{T}^{-1}(a'_n, j)) = \vec{f}(a'_1, \dots, a'_n)$$

where $\mathcal{T}^{-1}$ is the inverse transformation defined as follows:

$$\mathcal{T}^{-1}(x, j) = \begin{cases} x & x :: 0 \\ x' | x'[I_k^{-1}(i)] = x[i] & x :: k \\ x\{x_k \to V x_k + j\} & x :: idx(k) \\ x[*, j] & x :: \triangle \end{cases}$$

where $x' | x'[I_k^{-1}(i)] = x[i]$ means the reconstruction of the array x before the transformation was applied. Here we use I^{-1} index mapping, and as we know the shape relation of x and x' we can do the reconstruction.

We factorise the arguments, which makes application of the original function possible. Notice that idx and $\triangle$ types with different parameters imply V iterations per parameter. For example, if $\triangle^\alpha$ and $\triangle^\beta$ are layout types of the f arguments, then we have to consider not V applications of f but $V \times V$. However, such a function can never be applied in the nest of map/reduces unless one of the arguments is dropped. This can be proven using the following sketch:

1. if a function gets an idx or $\triangle$ layout type with more than one parameter then the function is being called from within a map/reduce nest (possibly spreading across functions):

```
map i < $u_1$
    map j < $u_2$
        f (a[i] :: $\triangle^\alpha$, b[j] :: $\triangle^\beta$)
```

2. if arguments of the function f are combined via primitive built-in language combinators, then according to Fig. 4.10 parameters have to be unified;

3. if not then the parameters has to be dropped, as in the following example:

```
f (x :: $\triangle^\alpha$, y :: $\triangle^\beta$) = y

...
map i < $u_1$
    a[i] + (reduce j < $u_2$ (+)
                f (a[i] :: $\triangle^\alpha$, b[j] :: $\triangle^\beta$))
```

in which case the function can be replaced with a one-argument function where all the idx and $\triangle$ parameters are the same.

That suggests that functions applied inside map/reduces can only represent V instances of the original function.

5.3.1 Correctness criteria

With these definitions we can now define correctness criteria for the proposed transformation. We formalise the notion of correctness by means of a predicate:

$$\mathcal{C}(\mathcal{T}, e) \mapsto \{T, F\}$$

where the co-domain is (true T or false F). The predicate is defined separately for expressions (E in SAC-λ grammar), function definitions FUNDEF and the whole programs LETREC.

The expression e under transformation $\mathcal{T}$ is correct if:

$$
\begin{aligned}
e &\equiv f(a_1, \dots, a_n) \\
e' &= \mathcal{T}(e) = f'(a'_1, \dots, a'_n) \\
\mathcal{C}(\mathcal{T}, e) &= \begin{cases}
e' \equiv e & e :: 0 \\
\forall_{i<s_e} e[i] = e'[I_k(i)] & e :: k \in \mathbb{Z}_+ \\
\oplus_{j=1}^{V} f(\mathcal{T}^{-1}(a'_1, j), \dots, \mathcal{T}^{-1}(a'_n, j)) = f'(a'_1, \dots, a'_n) & e :: \triangle \\
G(e) = M^{-1}(e') \wedge e' = M(G(e)) & e :: idx(k)
\end{cases}
\end{aligned}
$$

where $f(a_1, \dots, a_n)$ is e with explicit free variables.

Expressions of layout-type 0 are correct if the transformed expression is equivalent to the original one. Expressions of layout-type $k, k \in \mathbb{Z}_+$ are correct if every element of e at position i evaluates to the same value as $e'[I_k(i)]$ for all the valid indexes in e. Expressions of layout-type $\triangle$ are correct if the transformed function can be replaced with V instances of the original function. Expressions of layout-type $idx(k), k \in \mathbb{Z}_+$ are correct if the index mapping between the transformed and the original expression is preserved.

Finally, a function definition is correct when the function body is correct:

$$
\frac{\mathcal{C}(\mathcal{T}, e) = T}{\mathcal{C}(\mathcal{T}, f(a_1, \dots, a_n) = e) = T}
$$

The overall program is correct if all the function-definitions are correct and the goal expression is correct:

$$
\frac{\mathcal{C}(\mathcal{T}, f_1(\dots) = e_1) = T \qquad \dots \qquad \mathcal{C}(\mathcal{T}, f_n(\dots) = e_n) = T \qquad \mathcal{C}(\mathcal{T}, e) = T}{\mathcal{C}(\mathcal{T}, \textbf{letrec } f_1(\dots) = e_1, \dots, f_n(\dots) = e_n \textbf{ in } e) = T}
$$

5.4 Proof of correctness

In this section we are going to sketch a proof that any program in SAC-λ transformed by $\mathcal{T}$ is semantically correct in terms of $\mathcal{C}$. We are going to use structural induction over terms of the language proving that expressions constructed of correct subparts under the given typing are correct.

Every case of the induction compares the original expression e, which has a normal form $e = f(a_1, \dots, a_n)$ and its transformed counterpart: $e' = \mathcal{T}(e)$ which also has a normal form $e' = f'(a'_1, \dots, a'_n)$. We are going to consider all the possible layout typings of e and the transformation obtained by $\mathcal{T}$ under the given layout typing.

First of all let us start with simple general observations.

Theorem 1. *Transformation of an expression of layout type zero is correct, assuming that all the subexpressions are correct.*

$$\frac{f(a_1,\ldots,a_n) :: (0,\ldots,0) \to 0 \qquad \mathcal{C}(\mathcal{T}, a_1) = T \quad \ldots \quad \mathcal{C}(\mathcal{T}, a_n) = T}{\mathcal{C}(\mathcal{T}, f(a_1,\ldots,a_n)) = T}$$

Proof. Follows from the definition of $\mathcal{T}$ when all the subexpressions and the resulting expression are of layout type 0 and they preserve the semantics of the original program, then f' is equivalent to f as $\mathcal{T}$ is an identity function. □

The second observation concerns expressions of the *idx* layout types.

Theorem 2. *Transformation of an expression of* $idx(k)$ *layout type is correct, assuming that all the subexpressions are correct.*

$$\frac{e :: idx(k)}{\mathcal{C}(\mathcal{T}, e) = T}$$

Proof. Expressions of type $idx(k)$ can appear either as variables inside map/reduce expressions, as a result of a $+\!\!+$ operation or by means of propagation via function arguments. According to the definition of $\mathcal{C}$ we have to show that $G(e) = M^{-1}(e')$ and that $e' = M(G(e))$.

In the case of map/reduce the transformation $\mathcal{T}$ updates the upper-bound of the operation. We need to show that the transformed index space I' includes every index from the original index space I under M and G. In other words:

$$\{G(i) \mid \forall i \in I\} = \{M^{-1}(\vec{\imath}) \mid \forall \vec{\imath} \in I'\} \quad \text{and} \quad \{M(G(i)) \mid \forall i \in I\} = I'$$

If the original map/reduce is bound by the expression $[u_1,\ldots,u_n]$, the index space of the operation will be defined as:

$$I = \{[0,\ldots,0],\ldots,[u_1 - 1,\ldots,u_n - 1]\}$$

as indexes in SAC-λ start with zero. In the case that the induction variable of the map/reduce is of layout type $idx(k)$ the transformation $\mathcal{T}$ applies U_k to the upper bound of the map/reduce. That means that the transformed upper bound will look like $[u_1 - 1,\ldots,\lceil u_k/V \rceil,\ldots,u_n]$ and the transformed index space I' will look like:

$$I' = \{[0,\ldots,0],\ldots,[u_1 - 1,\ldots,(u_k - 1)\ \texttt{div}\ V,\ldots,u_n - 1]\}\}$$

because:

$$\lceil x/V \rceil - 1 \equiv (x-1)\ \texttt{div}\ V$$

Due to the definitions of M and G the following equality holds:

$$I' = \bigcup_{i \in I} \{M(G(i))\}$$

as $M(G(e)) = e\{e_k \to e_k \texttt{ div } V\}$ and due to the definition of M^{-1}, reverse equality:

$$\{M^{-1}(\vec{\imath}) \mid \vec{\imath} \in I'\} = \{G(i) \mid \forall i \in I\}$$

holds under the assumption that u_k is available for M^{-1} and G to resolve the indexes on the bounds of iteration space. This ensures that the transformed index set of a map/reduce operation is correct.

Now we consider the cases when the layout type $idx(k)$ is propagated or is obtained via a $+\!\!+$ operation. Let us consider an expression e in its normal form: $e = f(a_1, \dots, a_n)$. In that case, if $e :: idx(k)$ there exists a free variable of $iv = a_i$ of layout type $idx(m)$. In that case we have two possibilities:

1. *propagation,* $k = m$ which means that the function returns iv, in which case the correctness holds under assumption that iv is correct;

2. *concatenation* which means that $iv :: idx(m)$ is a part of either $iv +\!\!+ v$ or $v +\!\!+ iv$, where $v :: 0$. According to the type rules:

$$e = iv +\!\!+ v \qquad \text{and} \qquad e :: idx(k) \quad k = m$$

and

$$e = v +\!\!+ iv \qquad \text{and} \qquad e :: idx(k) \quad k = m + \mathcal{S}_0\,(v)$$

Let us expand both cases, assuming that:

$$iv = [iv_1, \dots, iv_m, \dots, iv_n] \quad v = [v_1, \dots, v_p]$$

In the first case the original expression would be:

$$e = [iv_1, \dots, iv_m, \dots, iv_n, e_1, \dots, e_p]$$

and the transformed expression:

$$e' = [iv_1, \dots, iv_m \texttt{ div } V, \dots, iv_n, e_1, \dots, e_p] = e\{e_m \to e_m \texttt{ div } V\}$$

which implies that $M(G(e)) = e'$ by definition and $M^{-1}(e') = G(e)$ because

$$M^{-1}(e') \overset{def}{=} \langle e'\{e'_m \to Ve'_m\}, \dots\rangle = \langle e\{e_m \to V(e_m \texttt{ div } V) + 0\}, \dots\rangle$$

which by definition is $G(e)$.

The intuition here is that $\mathcal{T}$ divides the k-th component of the vector by V, so as long as the position of the component is preserved, the transformation is correct.

Finally, when $e :: idx(m + \mathcal{S}_0\,(v))$ the original expression is:

$$[v_1, \dots, v_p, iv_1, \dots, iv_m, \dots, iv_n]$$

and similarly to the previous case, after the concatenation the component of the vector that will be divided by V is shifted p elements to the right, and $p = \mathcal{S}_0\,(v)$. So similarly to the previous case we have shown that the transformation is correct. □

The rest of the proofs use structural induction over the terms. The base case of the induction is a constant, as we do not allow external parameters, and all the values are conceptually defined by means of constants.

Theorem 3. *Constant transformation is semantically correct.*

$$\frac{e \;\text{ is constant}}{\mathcal{C}(\mathcal{T}, e) = T}$$

Proof. By the construction of CONST transformation. Let us consider three possible cases of the layout-type of a transformed constant:

1. $e :: 0$ — non-vectorised case, follows from Theorem 1.

2. $e :: \triangle$ in which case for $f() = e$ and $f'() = e'$ we show that:

$$\oplus_{j=0}^{V} f() = f'() \qquad \text{or} \qquad \oplus_{j=0}^{V} e'[*, j] = e$$

   which directly follows from the definition of $\mathcal{T}$ for constants of layout type $\triangle^0$: $e' \overset{def}{=} \oplus_{i=0}^{V} e$[2].

3. $e :: k \in \mathbb{Z}_+$ in which case we show that

$$\forall_{i<s_e} : e[i] = e'[I_k(i)]$$

   This follows from the construction of e', and the phantom elements are not in the range of i.

□

Theorem 4. *Vectorisation of selections is semantically correct.*

$$\frac{e \equiv \mathbf{sel}(iv, a) \qquad \mathcal{C}(\mathcal{T}, iv) = T \qquad \mathcal{C}(\mathcal{T}, a) = T}{\mathcal{C}(\mathcal{T}, e) = T}$$

Proof. Let us consider all the legal layout types for e, iv and a:

1. $e :: 0 \wedge a :: 0 \wedge iv :: 0$ — non-vectorised case, follows from Theorem 1.

2. $e :: \triangle \wedge a :: k \wedge iv :: idx(k)$ in which case we need to show that:

$$\oplus_{j=0}^{V} \mathbf{sel}(iv'\{iv'_k \to V iv'_k + j\},\ \mathcal{T}^{-1}(a')) = \mathbf{vsel}(iv', a')$$

   From the definition of **vsel** we deduce the following equality:

$$\mathbf{vsel}(iv', a') = \oplus_{j=0}^{V} \mathbf{sel}(iv' \mathbin{+\!\!+} [j], a')$$

[2]By the way, this is the only case for expressions of layout type $\triangle$ when the scalar elements of the transformed one can be mapped to the scalar elements of the original one.

From the reversed correctness criteria of $a :: k$ we get the following mapping between a' and $\mathcal{T}^{-1}(a')$:

$$\forall_{I_k^{-1}(i) < s_a} \ \mathbf{sel}(I_k^{-1}(i), \mathcal{T}^{-1}(a')) \equiv \mathbf{sel}(i, a')$$

From $\mathcal{C}(\mathcal{T}, a)$ the shape of a' is $[s_1, \ldots, s_n, V]$. The shape of iv' is $[s_1, \ldots, s_n]$, which allows one to rewrite the original expression as:

$$\oplus_{j=0}^{V} \ \mathbf{sel}(iv'\{iv'_k \to Viv'_k + j\}, \mathcal{T}^{-1}(a')) = \oplus_{j=0}^{V} \mathbf{sel}(iv' +\!\!+ [j], a')$$

which follows directly from the definition of I_k^{-1}.

3. $e :: \triangle \wedge a :: \triangle \wedge iv :: 0$ in which case we need to show that

$$\oplus_{j=0}^{V} \ \mathbf{sel}(iv, a'[*, j]) = \mathbf{vsel}(iv, a')$$

From $\mathcal{C}(\mathcal{T}, a)$ we deduce that $a' = \oplus_{j=0}^{V} a'[*, j]$ and from the construction of **vsel**, correctness follows directly.

4. $e :: 0 \wedge a :: k \in \mathbb{Z}_+ \wedge iv :: 0$ in which case we need to show that

$$\mathbf{sel}(iv, a) = \mathbf{sel}(I_k(iv), a')$$

which directly follows from $\mathcal{C}(\mathcal{T}, a)$.

□

Theorem 5. *Vectorisation of a primitive binary operation is semantically correct.*

$$\frac{e \equiv x + y \qquad \mathcal{C}(\mathcal{T}, x) = T \qquad \mathcal{C}(\mathcal{T}, y) = T}{\mathcal{C}(\mathcal{T}, e) = T}$$

Proof. There are four possible layout-type combinations for a given expression.

1. $e :: 0 \wedge x :: 0 \wedge y :: 0$ — non-vectorised case, follows from Theorem 1.

2. $e :: \triangle \wedge x :: \triangle \wedge y :: 0$ in which case we need to show that:

$$\oplus_{j=0}^{V}(x'[j] + \left(\oplus_{i=0}^{V} y\right)[j]) = x' \vec{+} \left(\oplus_{i=0}^{V} y\right)$$

Please note that x' has a shape $[V]$ so we can replace slice operation with a simple selection. From $\mathcal{C}(\mathcal{T}, x)$ we can deduce that $x' = \oplus_{i=0}^{V} x'[i]$. By the definition of built-in vector operations:

$$a \vec{+} b \overset{def}{=} [a[0] + b[0], \ldots, a[V-1] + b[V-1]] \equiv \oplus_{i=0}^{V} \left(a[i] + b[i]\right)$$

as a and b are of shape $[V]$. We can rewrite the original equality as:

$$\oplus_{j=0}^{V}(x'[j] + \left(\oplus_{i=0}^{V} y\right)[j]) = \oplus_{k=0}^{V} \left(x'[k] \vec{+} \left(\oplus_{i=0}^{V} y\right)[k]\right)$$

which makes it correct by definition of $\oplus$.

3. $e :: \triangle \wedge x :: 0 \wedge y :: \triangle$ — same reasoning as in the previous case.

4. $e :: \triangle \wedge x :: \triangle \wedge y :: \triangle$ in which case $\mathcal{C}(\mathcal{T}, a) \implies a' = \oplus_{j=0}^{V} a'[j]$ and $\mathcal{C}(\mathcal{T}, b) \implies b' = \oplus_{i=0}^{V} b'[i]$ in which case we can use the same reasoning as in previous cases.

□

Theorem 6. *Vectorisation of a map is semantically correct:*

$$\frac{e \equiv \textbf{map } i < u \; e_1 \qquad \mathcal{C}(\mathcal{T}, u) = T \qquad \mathcal{C}(\mathcal{T}, e_1) = T}{\mathcal{C}(\mathcal{T}, e) = T}$$

Proof. As e_1 is correct there is a function f_1 with body e_1. Let us assume, without loss of generality, that f_1 arguments have i at the first position. In case i is not a free variable of e_1 this argument will be skipped, but we can still pass it. Let us consider legal layout-type combinations of e, i and f.

1. $e :: 0 \wedge i :: 0 \wedge f_1 :: (0, \ldots) \to 0$ — non-vectorised case, follows from Theorem 1.

2. $e :: k \wedge i :: idx(k) \wedge f_1 :: (idx(k), \ldots) \to \triangle$ Let us assume that the shape of non-transformed e_1 is scalar. Later we will demonstrate that the reasoning works for non-scalars as well. We need to show that:

$$\forall_{j<s_e} (\textbf{map } i < u \; f_1(i, \ldots)) [j] = (\textbf{map } i' < u\{u_k \to \lceil u_k/V \rceil\} \; f_1'(i', \ldots)) [I_k(j)]$$

   From the correctness of f_1 we conclude that:

$$f_1'(i', \ldots) = \oplus_{j=1}^{V} f(i'\{i_k' \to V i_k' + j\}, \ldots)$$

   From the construction of I_k:

$$I_k(j) = j\{j_k \to j_k \texttt{ div } V\} \mathbin{+\!\!+} [j_k \texttt{ mod } V]$$

   As we assumed that the shape of f_1 is scalar, let us note that the index space of j under the quantifier and i in the first map are the same. The index space of $j\{j_k \to j_k \texttt{ div } V\}$ and the index space of i' are the same as well. The shape of f_1' is $[V]$, and we can rewrite the original expression as:

$$\forall_{j<s_e} f(j, \ldots) = f'(j\{j_k \to j_k \texttt{ div } V\}, \ldots)[j_k \texttt{ mod } V]$$

   which is true due to the correctness of f.

3. $e :: \triangle \wedge i :: 0 \wedge f_1 :: (0, \ldots) \to \triangle$ in which case we need to show that

$$\oplus_{j=0}^{V} \textbf{map } i < u \; e_1'[*, j] \equiv \textbf{map } i < u \; e_1'$$

   As $e_1' :: \triangle$ and $\mathcal{C}(\mathcal{T}, e_1) \implies \oplus_{j=0}^{V} e_1'[*, j] \equiv e_1'$ the correctness of the expression follows from the semantics of **map**.

4. $e :: \mathcal{S}_0\,(i) + k \wedge i :: 0 \wedge f :: (0, \dots) \to k \in \mathbb{Z}_+$ In this case we need to show that

$$\forall_{j<s_e} \left(\textbf{map } i < u\ e_1\right)[j] = \left(\textbf{map } i < u\ e_1'\right)\left[I_{\mathcal{S}_0(i)+k}(j)\right]$$

   From $\mathcal{C}(\mathcal{T}, e_1)$ follows that $\forall i : \forall j < s_{e_1} e_1[j] = e_1'[I_k(j)]$ and the expression of interest reduces to this equality if for every index we drop $\mathcal{S}_0\,(i)$ first elements.

Finally we have to make sure that if e_1 is not scalar, correctness still holds. Assuming that the shape of e_1 is s_{e_1} we apply the same reasoning considering that all the values in our program are defined as

$$\langle [s_1, \dots, s_n], [d_1, \dots, d_p] \rangle$$

where d_i is of shape s_{e_1}. That reduces a non-scalar map to a scalar one, and makes the same reasoning applicable. □

Theorem 7. *Vectorisation of reduce is semantically correct:*

$$\frac{e \equiv \textbf{reduce } i < u\ (f_{\text{bin}})\ e_1 \qquad \mathcal{C}(\mathcal{T}, u) = T \qquad \mathcal{C}(\mathcal{T}, f_{bin}) = T \qquad \mathcal{C}(\mathcal{T}, e_1) = T}{\mathcal{C}(\mathcal{T}, e) = T}$$

Similarly to map, due to $\mathcal{C}(\mathcal{T}, e_1)$ there exists a function f_1 for which without lose of generality we introduce an argument i at the first position.

First of all let us consider the transformation informally, proving the properties we will use later. A reduce operation can be considered as the following expression:

$$r = w_{i_1} \times w_{i_2} \times \dots \times w_{i_n}$$

where w_k is a result of the application of function $f_1(i, \dots)$ to $(k, \dots)$ and $\times$ is f_{bin} written in the infix notation.

First of all, as $\times$ is associative ($a \times (b \times c) \equiv (a \times b) \times c$), the original expression can be divided into two subgroups at index k where each subgroup can be computed concurrently:

$$\begin{aligned} r_1 &= (w_{i_1} \times \dots \times w_k) \\ r_2 &= (w_{k+1} \times \dots \times w_{i_n}) \\ r &= r_1 \times r_2 \end{aligned}$$

The fact that $\times$ is commutative ($a \times b \equiv b \times a$) allows us to permute elements e_k in the r expression:

$$r = w_{\psi(i_1)} \times w_{\psi(i_2)} \times \dots \times w_{\psi(i_n)}$$

where $\psi(k)$ is a valid permutation in the range of $\{i_1, \dots, i_n\}$.

Now let us consider what happens when vectorising r with $\vec{\times}$:

$$[w_1' \dots, w_{V-1}'] \vec{\times} [w_1'', \dots, w_{V-1}''] \equiv [w_1' \times w_1'', \dots, w_{V-1}' \times w_{V-1}'']$$

as follows:

$$\begin{aligned}\vec{r} &= [w_1^{(1)}, \dots, w_{V-1}^{(1)}] \vec{\times} \cdots \vec{\times} [w_1^{(n/V)}, \dots, w_{V-1}^{(n/V)}] \\ r &= \vec{r}[0] \times \cdots \times \vec{r}[V-1]\end{aligned}$$

We assume here that $n \equiv 0 \mod V$. The only thing we have to demonstrate for operations to be identical is that for $e_k^{(j)}$ and e_i there exists a bijective mapping between (j, k) and i.

Finally, if every vector in the vectorisation contains "phantom" elements starting from the k-th to $V-1$ position, then the operation has to be adjusted to:

$$\begin{aligned}\vec{r} &= [w_1^{(1)}, \dots, w_k^{(1)}, x, \dots, x] \vec{\times} \cdots \vec{\times} [w_1^{(n/V)}, \dots, w_k^{(n/V)}, x, \dots, x] \\ r &= \vec{r}[0] \times \cdots \times \vec{r}[k]\end{aligned}$$

Proof. Let us consider a combination of all valid types for $\mathcal{T}(e)$, $\mathcal{T}(i)$ and $\mathcal{T}(f)$.

1. $e :: 0 \wedge i :: 0 \wedge f_1 :: (0, \dots) \to 0$ the non-vectorised case, follows from Theorem 1.

2. $e :: \tau \wedge i :: 0 \wedge f_1 :: (\tau, \dots) \to \tau, \tau \in \mathbb{Z}_+ \cup \{\triangle\}$ In this cases, as f_{bin} is semantically correct and $f_{bin} :: (\tau, \tau) \to \tau$, the only difference to the previous case is the non-scalar shape of the f_1 application. Using the same argument of assuming that the array components of e_1 and e are of shape s_{e_1} we reduce the proof to the previous case.

3. $e :: 0 \wedge i :: idx(k) \wedge f_1 :: (idx(k), \triangle) \to \triangle$ As we know, we can permute the elements that we are reducing with f_{bin}. We need to show that the mapping between e_1 and e_1' elements is bijective for all e_1; that the number of scalar reductions is the same in the transformed and original cases and that "phantom" elements do not affect the result.

   First of all, the mapping between e_1 and e_1' is I_k, which is bijective by definition, and as e_1 is evaluated for every element in the index set, all the original e_1 are mapped. Secondly, "phantom" elements do not affect the result as every vector in $\vec{r'}$ contains exactly u_k `mod` V elements starting from the first position. As f_{bin} is correct, the operation happens component wise, so the resulting vector is not affected and contains u_k `mod` V "real" values. Finally, the construction of r_2 guarantees that only "real" values are reduced.

   Now, let us count the number of elements in a non-vectorised case, which would be $N = \prod_{j=0}^{\mathcal{S}_0(u)} u[j]$. The number of f_{bin} applications is $N-1$. Next, consider the $\vec{r}$ part of the transformation. Assuming that:

$$u_k = V p_k + q_k$$

   where $0 < q_k < V$, the number of scalar elements reduced in $\vec{r}$ is $\frac{N}{u_k} p_k$. Each vector operation is the same as V scalar operations, so we have $\left(\frac{N}{u_k} p_k - 1\right) V$ scalar operations for this part.

For $\vec{r'}$ we have $\frac{N}{u_k} \cdot 1$ vector elements, where only q_k of them are non-phantom. This means that totally we have $(\frac{N}{u_k} - 1)q_k$ operations.

Finally, r_1 adds $V - 1$ operations to reduce $\vec{r}$, r_2 adds $q_k - 1$ operations to reduce $\vec{r'}$, and final call of f_{bin} adds one more operation. If we sum these all together we get:

$$\left(\frac{N}{u_k} p_k - 1\right) V + \left(\frac{N}{u_k} - 1\right) q_k + V - 1 + q_k - 1 + 1$$

which adds up to $N - 1$.

□

For reduce, $\vec{r}$ might result in an empty range if $u_k < V$ in which case the neutral element is being returned. Also, $\vec{r'}$ might result in empty range, if u_k `mod` $V = 0$ in which case, again, the neutral element would be used. One of $\vec{r}$ or $\vec{r'}$ is not empty as $u_k \geq 1$.

Finally we consider IF, APP, LET, FUNDEF and LETREC constructions. The correctness of IF follows directly from the correctness of the `mask` operation and the assumption that the control-flow of a vectorised recursive function does not change. The correctness of APP follows directly from the definition of $\mathcal{C}$ under assumption that the function itself and its arguments are correct. The same reasoning holds for LET.

For FUNDEF, as $\mathcal{T}(f(a_1, \ldots, a_n) = e) = f(a_1, \ldots, a_n) = \mathcal{T}(e)$, we can see that correctness trivially holds under the assumption that e is correct.

The same reasoning holds for LETREC. However, note that we might have recursive functions in the body of function definitions, for which we need a fixed point iteration, but for the co-domain of $\mathcal{C}$ the fixed point becomes trivial, so we assume that the expression is correct and continue the induction.

5.5 Application

Now we are going to consider a small example of matrix multiplication to demonstrate how the transformation is applied to real code. We are going to consider a single function, assuming that it is being called in some LETREC construct.

```
matmul (a ,b) =
    map i < [N]
        map j < [N]
            reduce k < [N] (+) a[i++k] * b[k++j]
```

Both arguments `a` and `b` are of shape $[N, N]$. Now let us consider that the layout type inference is done, and we have the following types for the transformed expressions:

```
# matmul :: (2,1) → 0
matmul (a :: 2, b :: 1) =
    map i :: 0 < [N]
        map j :: 0 < [N]
            reduce k :: idx(1) < [N] (+)
                           a[(i++k) :: idx(2)] :: △ * b[(k++j) :: idx(1)] :: △
```

Next we can apply $\mathcal{T}$ on `matmul` which results in the following function:

```
matmul(a, b) =
    map i < [N]
        map j < [N]
            let
                rv1 = reduce k < [N/V] (vplus)
                                vmul (vsel (i++k, a),
                                      vsel (k++j, b));

                rv2 = reduce k < [if N mod V == 0 then 0 else 1] (vplus)
                                vmul (vsel (i++[N/V], a),
                                      vsel ([N/V]++j, b));
            in
                (reduce i < [V] (+) rv1[i])
                + (reduce i < [N mod V] (+) rv2[i])
```

Built-in vector operations are denoted as `vplus` and `vmul` for addition and multiplication accordingly. In the final step of the **reduce** transformation we use the fact that both `rv1` and `rv2` return 1-d arrays of shape $[V]$. That allows us to replace slicing `[*,i]` with a standard selection `[i]`.

5.6 Related work

As we have mentioned in 4.7 there are a number of works in the context of data layouts that try to improve the cache behaviour or streaming through GPUs. Some attempt to solve a very similar problem of transforming data layouts for better SIMD usage. However, although the goals are very similar, to our knowledge none of the existing work considered layout modifications as a whole program transformation and to demonstrate how it might be correct.

Peter Hawkins et al. in [53] propose to synthesize data representations for concurrent programs written using concurrent relations. The main stress of this work lies in optimising locks and synchronisations in concurrent environments, making sure that the generated code stays correct. The overall technique they propose could be used to solve the problem of finding optimal data layouts for vectorised array operations. However, the proposed setup makes it very difficult to use existing languages and codes. Our setting is more restrictive, as we do not allow tree-like or graph-like data structures, and support multi-dimensional arrays only. Thus decisions about synchronisation or locking placement are much simpler, as data structures are uniform. At the same time this allows us to maintain a close gap between existing languages like C and Fortran and the proposed transformation, making it applicable to these languages. Here programs would have to be analysed in order to reveal data-parallel operations. Large portion of such an analysis can be done using the polyhedral model.

Finally, we give special attention to [134] where authors use data layout transformation to optimise grid applications for efficient execution on GPUs. The overall transformations they are studying are very similar to our work. However the application domain is bounded to one particular class. The underlying programming language used in the study is C with custom extensions for propagation of layout information. This makes it hard to judge how difficult would it be to generalise this technique on a wider class of applications.

5.7 Conclusions

We have demonstrated how layout types can be used to transform data placement in memory alongside the operations on these data structures. We have also demonstrated that the transformed program stays correct with respect to the scalar elements it computes. Finally we illustrated the proposed technique with the example of matrix multiplication.

These contributions constitute a stepping stone towards automating the layout transformation process. In particular our formulation of the correctness criterion plays a key role as it enables formal reasoning about the correctness of our transformation. It also is key for enabling a description of the layout transformation as a high-level program transformation. This, in turn can serve as a blue-print for an effective implementation.

When combining this work with the work on data layout inference presented in Chapter 4 a fully automation of the entire process seems within reach. All that is missing is a cost model to estimate the expected performance of transformed programs. Even for simple examples the number of transformations can be quite big. For example, for matrix multiplication several layout types are possible. And this number, at least potentially, grows with the dimensionality of the function arguments. We are confident that it will be possible to apply existing approaches towards cost models for functional languages to close this gap and, thus, to achieve a fully automated layout adjustment process.

Chapter 6

Implementation

In this chapter we are going to relate SaC-λ and SaC, explain the differences in the implementation of the inference and transformation when using SaC instead of SaC-λ and discuss the cost model we use to chose the vectorisation of a program.

The reference implementation of the SaC language, called `sac2c`, is an optimising compiler written in C consisting of 400K lines of code. It uses a standard compiler architecture — a front-end which parses input files and performs basic syntax de-sugaring, a middle-end which consists of a cycle of optimisations operating on the Abstract Syntax Tree (AST) and the backend responsible for code generation for various architectures. For more details refer to [50].

The proposed layout inference and transformations are integrated into `sac2c` and consist of three main parts — the inference system, the cost model and transformations together consisting of 8K lines of C code. The inference happens between the middle-end and the back-end after the optimisation cycle is finished. The transformation system is designed to output SaC code with explicit vector operations which is assumed to be compiled with `sac2c` to get binary. The staging provides a convenient verification of the transformation system and the ability to repeat the optimisations on the vectorised code.

6.1 SaC and SaC-λ relation

The main reason we used SaC-λ instead of SaC in our formalism is that SaC-λ is very close to the intermediate representation of the AST in the middle-end.

The syntax of SaC is more advanced than the subset we have covered via SaC-λ. However, most of the constructions that are left out are either syntactic sugar for programmer convenience or means to deal with standard typing, module system, etc., which are not relevant for the transformations we have presented.

6.1.1 De-sugaring SaC into SaC-λ

The syntax of SaC is designed to be close to C, while yet its semantics stays purely functional. In C functions consist of statements; syntactically this is the same in

SAC, but all the statements are transformed into pure functional expressions. The chain of statements is transformed into a nested let expression. Here is an example:

```
## SaC ##                               ## SaC-λ ##
int foo (int a, int b)                  foo (a, b) =
{                                           let a = 5 in
  a = 5;                                        let i = 0 in
  i = 0;                                            let a' = for_loop (i, a) in
  for (i = 0; i < 10; i++)                              a'
    a = ...
  return a;
}
```

Such a transformation happens when parsing SAC code and building the AST for the program. Now let us explain how individual statements are mapped.

Return statement In SAC a return statement is allowed to appear only once at the end of a function. An expression of the return statement will be treated as a result of the function application. In SAC-λ the body of a function is an expression itself, so there is no need to use **return** explicitly.

Assignments As it can be seen from the previous example, assignments are directly mapped into let-expressions. This is done via transforming the original code into Static Single Assignment (SSA) form, that is — if a variable is re-assigned in the chain of expressions, a new variable name is created, and the variable is substituted in the remainder of the assignment chain. Refer to [50] for more details. Consider the following example:

```
## SaC ##                               ## SaC-λ ##
{
  a = 5;                                let a = 5 in
  ...                                       ...       in
  a = a + 5;                                let a' = a + 5 in
  ...                                       ...                 in
  return a;                                     a'
}
```

Conditional statements An expression evaluated in a branch of the conditional statement, it has to be assigned to a non-local variable. That gives rise to a control-flow to data-flow transformation. We compute a union of variables that are modified in both branches, and split the conditional statement into multiple statements each of which updates one variable from the union using the predicate of the conditional statement. Consider the following example:

```
## SaC ##
a = 5;
b = foo (a);
if (b > 0) {
  a = 1;
  b = 2;
}
```

```
## SaC-λ ##
let a = 5 in
    let b = foo (a) in
        let a' = if (b > 0) then 1 else a in
            let b' = if (b > 0) then 2 else b
```

Variables `a` and `b` can be modified in the branches of the conditional statement. As the else branch does not exist, we can assume that in the else branches variables are being reassigned to their current values.

Loops SaC supports all three kinds of loops that are valid in C: for-loop, do-loop and while-loop. Control-flow modifiers like `break`, `continue` and `goto` are not supported, which makes it straight-forward to transform a loop into a call to a tail-recursive function. We have to find variables that are modified inside the loop which are going to be results of the tail-recursive function. Modified variables and referenced variables of the loop will form a list of parameters. Consider an example:

```
## SaC ##
a = 5;
for (i = 0; i < N; i++) {
    a = a + f (i);
}
```

```
## SaC-λ ##
rec_for (i', a', N) =
    if (i' < N)
        let a'' = a' + f (i') in
            let i'' = i' + 1 in
                rec_for (i'', a'', N)
    else
        a'

...
let a = 5 in
    let i = 0 in
        rec_for (i, a, N)
```

Do and while loops can be transformed similarly. Note that a loop can modify several variables. In that case we can either support multiple return values (this is the way it is implemented in SaC), or we can create a new recursive function for every variable. This is going to be inefficient in terms of performance, but it will be equivalent functionally, as functions and expressions in SaC do not have side effects.

Minor differences Other parts of SaC are either irrelevant in terms of transformations, for example: constructions to deal with the module system, element types or constructions that can be trivially substituted.

For example, SaC allows one to fuse assignment and operation the same way as it is done in C: `a += b`, which translates to `a = a + b`.

SaC mimics C's lazy operations `||` and `&&` by substituting for them with conditionals:

```
## SaC ##                              ## SaC-λ ##
f (a && b,                             f (if a then b else false,
   c || d)                                if c then true else d)
```

6.1.2 Non-de-sugarable differences

More significant distinctions between SaC and SaC-λ are concerned with shape-invariant array programming and more advanced map/reduce syntax.

Shape-invariant programming One of the strong features of SaC is its ability to handle arrays and operations on them without explicit knowledge of their shapes or even ranks. This is achieved by introducing special types, which allow one to define arrays where only the rank or the base type is known.

`int[.,.,.]`	Three-dimensional array with element-type `int`; exact size at each axis will be known at runtime only.
`float[*]`	An array with element-type `float` of unknown rank and size. Can be also scalar. The least precise type in SaC.

To handle data of such types SaC introduces two built-in operators, namely `shape` and `dim`:

`shape`	Returns a vector containing sizes of an array at every axis. For example, `shape ([[1,2],[3,4]])` will return `[2,2]`.
`dim`	Returns the rank of an array. For example, `dim ([[1,2],[3,4]])` will return 2.

Note that when using those operators we introduce a dependency between type components and values of a program. Consider the following example:

```
int foo (int[.,.,.] x)
{
    return sum (shape (x));
}
```

Here the components of the array type become values. Now consider:

```
int[*] bar (int x)
{
    return genarray ([x,x], 0);
}
```

In this case, a value becomes a part of the type of a generated array.

This has an impact on the layout transformations we can make. First of all, our system cannot deal with arrays of unknown dimension. That is because MAP[△] and IDX[△] refer to the length of an index vector. In case all the array ranks are known at compile time, we can compute the length of any index vector as well. In case the length of an index vector is not known, our type constraints cannot be resolved immediately, which makes our algorithm inapplicable in the form it is presented in this thesis. Although it might be possible to postpone constraint resolutions, this is out of the scope of this thesis.

Secondly, when using `shape` on the transformed programs with new data layouts it has to return the original values, otherwise transformed programs will evaluate wrong results — consider function `foo` above as an example. This means that we either have to keep the original shapes, or recompute them from the new shapes. Unfortunately, the latter is not possible due to the paddings we use. If the original shape is $[N, N]$, for the layout type 2, the new shape will be $[N, (N + V - 1)\ \texttt{div}\ V, V]$, and integer division is irreversible. We solve this problem by introducing a new component of the array in SAC programs — the original shape. Originally every SAC value was formally described with a pair $\langle \text{shape}, \text{data}\rangle$. After the transformation we extend this tuple with a new component, the original shape: $\langle \text{shape}, \text{orig\_shape}, \text{data}\rangle$, and we make sure that all applications of `shape` return the original shape.

Extended map/reduce Map and reduce operators in SAC-λ are restricted versions of the `with`-loop construct in SAC. The main difference is that **map** and **reduce** in SAC-λ do not allow one to update regions of an iteration space. That is:

```
map i < u f(i)
```

generates an index space of shape `u` and returns the result which is also of the shape `u`. That implies that in order to update a region of an array, the body of a map has to contain a condition. For example, in the case when we want to fill a region from [20,20] to [40,40] with evaluated expression and make all the other elements of the array `a` of shape [100,100] to be zero, we will have to write the following SAC-λ code:

```
a = map i < [100,100]
        if i[0] >= 20 and i[0] < 40
           and i[1] >= 20 and i[1] < 40 then
             f (i)
        else
             0
```

Semantically equivalent SAC code will look like this:

```
a = with {
      ([20,20] <= i < [40,40]): f (i);
}: genarray ([100,100], 0)
```

Both codes evaluate to the same result. However, during the execution, a comparison on every iteration results in a large performance penalty. To avoid this SAC generates code that splits the overall iteration space in subspaces, where all elements in a subspace are evaluated unconditionally. In the given example we can evaluate zero on subspaces: ([0,0] to [20,100]), ([40,0] to [100,100]), ([20,0] to [40,20]) and ([20,40] to [40,100]); and we evaluate `f (i)` on the subspace ([20,20] to [40,40]). In order to perform such an optimisation the subspaces have to be identified and as the SAC form is more restrictive than the SAC-λ form with a general condition, SAC is capable to perform more aggressive optimisations. This is a property that we would like to preserve, which means that our transformations have to deal with region-based map/reduces rather than with whole-range map/reduces.

Another form of with-loops in SAC that avoids a condition in the body is *modarray* with-loops. It is a very common pattern that a region of an array e.g. a single element has to be updated and all the other elements should be copied. Consider an example when we update an element of a two-dimensional array `a` of shape [100,100] at position [1,1] with a value `x`. The relevant SAC-λ code will look as follows:

```
a' = map i < [100,100]
         if i[0] == 1 and i[1] == 1 then
             x
         else
             a[i]
```

Semantically equivalent SAC code will look like:

```
a' = with {
         ([1,1] <= i < [2,2]): x;
}: modarray (a)
```

Again, the reason for having a separate construction for such a case is to make sure that the generated code will not execute condition on every iteration.

6.2 Transformations in details

Note that layout inference can be applied to SAC without any major adjustments. The transformations on the other hand are getting more complexity due to the more complicated nature of with-loops. We present how to transform all three kinds of with-loops: genarray, modarray and fold. The key aspect which is different from SAC-λ is to transform an index space bounded with lower and upper bounds. We start with an example of a genarray with-loop to develop an intuition:

```
a = with {
        ([12,10] <= iv < [33,17]): e;
}: genarray ([101,101], def);
```

Given that $e :: \triangle$ and $iv :: idx(1)$ the shape of the result will be of layout type 1 and the shape of the new array will be $[\lceil 101/V \rceil, 101, V]$.

Vector types Such a new array is a dual to the array of shape $[\lceil 101/V \rceil, 101]$ of vector type. A vector type is a type that holds V elements of the original array type. For example:

$$\text{float}[N, M, V] \qquad \text{is a dual to} \qquad \text{vfloat}[N, M]$$

where `vfloat` is defined as a vector of V `floats`. In SAC transformations, for convenience reasons, we are using a form with a vector type, rather than the form with extra dimension. So in our example the shape of the transformed result will be $[\lceil 101/V \rceil, 101]$.

The original index space also has to be cut at the first dimension (as $iv :: idx(1)$). We are looking at the new index space from $[\lfloor 12/V \rfloor, 10]$ to $[\lceil 33/V \rceil, 17]$. The problem we may face is that the original value at the first position of lower and upper bounds is not divisible by V. The implication is that we will have to update a part of a vector on those boundaries. In the given example, assume that $V = 8$ in which case we will start from the second octet ($\lfloor 12/8 \rfloor = 1$), but we will have to update only the last four elements. A similar situation happens with the upper bound as well, except we have to update the lower elements.

To solve this problem we divide the iteration space into three sub-spaces:

1. the inner space, where operations can be expressed on full vectors: $[\lceil 12/V \rceil, 10]$ to $[\lfloor 33/V \rfloor, 17]$ for our example;
2. the lower bound, where higher elements of a vector may need update: $[\lfloor 12/V \rfloor, 10]$ to $[\lfloor 12/V \rfloor + 1, 17]$ and we update $V - (12 \texttt{ mod } V)$ higher elements in the vector, given that $12 \texttt{ mod } V \neq 0$;
3. the upper bound, where lower elements of a vector may need update: $[\lfloor 33/V \rfloor, 10]$ to $[\lfloor 33/V \rfloor + 1, 17]$ updating the first $33 \texttt{ mod } V$ elements of a vector, given that $33 \texttt{ mod } V \neq 0$.

We can generalise this to a n-dimensional case in the following way. Given the original code:

```
a = with {
        ([$l_1,\dots,l_k,\dots,l_n$] <= iv < [$u_1,\dots,u_k,\dots,u_n$]): $e$;
}: genarray ($s$, $d$);
```

and $iv :: idx(k) \wedge e :: \triangle$, we get the following result:

```
a' = with {
        ([$l_1,\dots,\lceil l_k/V \rceil,\dots,l_n$] <= iv < [$u_1,\dots,\lfloor u_k/V \rfloor,\dots,u_n$]): $\mathcal{T}(e)$;
}: genarray ($\mathcal{T}(s)$, $\mathcal{T}(d)$);

if ($l_k$ mod $V \neq 0$)
     a'' = with {
             ([$l_1,\dots,\lfloor l_k/V \rfloor,\dots,l_n$] <= iv < [$u_1,\dots,\lfloor l_k/V \rfloor + 1,\dots,u_n$])
             : select_last ($V - (l_k$ mod $V)$, $\mathcal{T}(e)$, a'[iv]);
          }: modarray (a');
else
     a'' = a'

if ($u_k$ mod $V \neq 0$)
     a = with {
           ([$l_1,\dots,\lfloor u_k/V \rfloor,\dots,l_n$] <= iv < [$u_1,\dots,\lfloor l_k/V \rfloor + 1,\dots,u_n$])
           : select_first ($u_k$ mod $V$, $\mathcal{T}(e)$, a''[iv]);
        }: modarray (a'');
else
     a = a''
```

Selection functions when the shape of the body expression of the genarray is scalar are defined as:

```
type[V] select_last (int num, type[V] e1, type[V] e2)
{
    return with {
        ([0] <= [iv] < [V]): (iv < V - num) ? e2[iv] : e1[iv];
    }: genarray ([V], 0);
}

type[V] select_first (int num, type[V] e1, type[V] e2)
{
    return with {
        ([0] <= [iv] < [V]): (iv < num) ? e1[iv] : e2[iv];
    }: genarray ([V], 0);
}
```

These select `num` lower or higher elements from the expression `e1` and put corresponding elements of `e2` in the remaining positions. Keep in mind, that the last argument of selection functions selects a vector of V elements.

In case the shape of the body expression of the genarray is not scalar, we need to apply the above functions to every V elements of the last dimension.

The modarray with-loop is transformed in the same way with the difference that a genarray in the first with-loop has to be substituted with modarray.

To transform a fold with-loop we use the same approach, but instead of selecting lower and upper elements on every iteration, we reduce the relevant elements of the resulting vector. Given the original code:

```
a = with {
       ([$l_1,\dots,l_k,\dots,l_n$] <= iv < [$u_1,\dots,u_k,\dots,u_n$]): $e$;
}: genarray ($f$, $e_{\text{neut}}$);
```

and $iv :: idx(k) \wedge e :: \triangle$, we get the following result:

```
a' = with {
       ([$l_1,\dots,\lceil l_k/V \rceil,\dots,l_n$] <= iv < [$u_1,\dots,\lfloor u_k/V \rfloor,\dots,u_n$]): $\mathcal{T}(e)$;
}: fold ($\mathcal{T}(f)$, $\mathcal{T}(e_{\text{neut}})$);

if ($l_k$ mod $V \neq 0$)
    a'' = with {
              ([$l_1,\dots,\lfloor l_k/V \rfloor,\dots,l_n$] <= iv < [$u_1,\dots,\lfloor l_k/V \rfloor + 1,\dots,u_n$])
              : $\mathcal{T}(e)$;
          }: fold ($\mathcal{T}(f)$, $\mathcal{T}(e_{\text{neut}})$)
else
    a'' = $\mathcal{T}(e_{\text{neut}})$

if ($u_k$ mod $V \neq 0$)
    a''' = with {
               ([$l_1,\dots,\lfloor u_k/V \rfloor,\dots,l_n$] <= iv < [$u_1,\dots,\lfloor u_k/V \rfloor + 1,\dots,u_n$])
               : $\mathcal{T}(e)$
           }: fold ($\mathcal{T}(f)$, $\mathcal{T}(e_{\text{neut}})$)
else
    a''' = $\mathcal{T}(e_{\text{neut}})$

a = $f$ (with { ([0] <= iv < [V]): a'[iv];}: fold ($f$, $e_{\text{neut}}$),
        $f$ (with { ([$V - (l_k$ mod $V)$] <= iv < [V]): a''[iv];}: fold ($f$, $e_{\text{neut}}$),
             with { ([0] <= iv < [$u_k$ mod $V$]): a'''[iv];}: fold ($f$, $e_{\text{neut}}$)))
```

6.2.1 Cost model

The layout type system presented earlier in Chapter 4 infers all the valid layout typings for a given program. To transform a program we have to chose one of the variants out of the inferred ones. The choice can be made by a user, but in case the number of variants is large, this process becomes tedious. If we want to automate this process, we need to create a cost model that is going to estimate the performance impacts of a transformation.

A building block of such a cost model is the application of vector operations, i.e., if the transformed program replaces a scalar operation with a vector one inside the map/reduce context, we expect such a map/reduce operation to perform

$$N \cdot \mathit{Cost}(op) - \frac{N \cdot \mathit{Cost}(\vec{op})}{V}$$

time units faster, where op is a scalar operation and $\vec{op}$ is a vector counterpart and N is a number of iterations in the map/reduce with-loop, assuming that op is being executed on every iteration of the map/reduce.

Using such a technique we could sum-up such improvements per function and compute the cost of the main function we want to execute. Such a technique has a number of difficulties:

1. As the bounds of map/reduce operations are not always know, the impact will depend on a set of parameters. That implies that not only have we to perform symbolic evaluation when we compose two costs, but also we have to compare symbolic expressions when choosing the best performing variant.

2. Conditions may have different costs on different branches, which means that if we want to be precise, we have to know the distribution of true/false values when evaluating the predicate.

3. Memory overheads are not being considered. When vectorising arrays the memory footprint of an array might increase up to V times. This may have an impact on access times for the array elements. Predicting exact consequences is arbitrarily hard, as one would have to model the memory hierarchy and caches for a given architecture.

Although the above approach can produce very precise performance estimates its implementation is complex. In terms of this thesis we are going to apply the same principles but in a much more simple fashion. The general idea is to approximate the speed-ups that vector operations give us. The intuition is that by replacing a scalar arithmetic operation $a \times b$ with a vector arithmetic operation $a \vec{\times} b$ we get a V times speed-up per element for this operation. That is:

$$\mathrm{Cost}(a \times b) = x \qquad \mathrm{Cost}(a \vec{\times} b) = x \qquad \mathrm{Cost}_{pe}(a \vec{\times} b) = \frac{V \cdot \mathrm{Cost}(a \times b)}{\mathrm{Cost}(a \vec{\times} b)} = \frac{V \cdot x}{x} = V$$

where Cost is the time it takes to execute an operation and Cost_{pe} is the speed-up *per element*. To simplify, we can assume that the cost of scalar arithmetic operations is always the same:

$$\mathrm{Cost}(\times) = \mathrm{Cost}(\vec{\times}) = 1$$

and equals to one. In that case the per element cost will be:

$$\text{Cost}_{pe}(\times) = 1 \qquad \text{Cost}_{pe}(\vec{\times}) = V$$

How can we compose those costs together? We expect that two vectorised arithmetic operations preserve the per element speed-up:

$$\text{Cost}_{pe}(a\vec{\times}(b\vec{\times}c)) = V$$

as the first $\vec{\times}$ operates V elements per one unit of time, as does the second, and as a result the speedup is $\frac{2V}{2} = V$. In case one operation is vectorised and one is not:

$$a \times (b\vec{\times}c)$$

In the scalar case we will spend $V + 1$ time units for the given operation and in the vectorised case we will spend only two, which makes the expected speed-up $\frac{V+1}{2}$.

We can see that the composition function is an arithmetic mean:

$$A(x_1, \ldots, x_n) = \frac{x_1 + \cdots + x_n}{n}$$

In order to apply the cost function over the structure of a program it has to be homomorphic over terms, that is:

$$\text{Cost-fun}(e_1 \circ e_2) = \text{Cost-fun}(e_1) \oplus \text{Cost-fun}(e_2)$$

Unfortunately, the arithmetic mean does not have this property. That is:

$$A(x_1, x_2, x_3) \neq A(x_1, A(x_2, x_3))$$

In order to fix this we have to consider an arithmetic mean not as a single value but as a pair (s, n), where s is a per-element speed-up and n is a number of operations. In that case A will be defined as:

$$x_i = (s_i, n_i) \qquad A(x_1, \ldots, x_n) = x_1 + \cdots + x_n = (s_1 + \cdots + s_n, n_1 + \cdots + n_n)$$

the above property will hold, as:

$$A(x_1, A(x_2, x_3)) = A(x_1, (s_2 + s_3, n_1 + n_2)) = (s_1 + s_2 + s_3, n_1 + n_2 + n_3)$$

Using this principle, we define a cost function C which can be applied to expressions inductively in the following form:

$$C(e) = (s_e, n_e) \quad \text{and} \quad C(a) + C(b) = (s_a + s_b, n_a + n_b)$$

Further down we explain how to apply C to the individual terms.

Inductive definition

Constants yield zero speed-up.

$$C(\text{const}) = (0, 0)$$

The cost of a vectorised arithmetic primitive binary operation is $(V, 1)$, or $(1, 1)$ in case the operation is scalar. Other primitive operations (for example selections) cost $(0, 0)$.

$$C(a \times b) = (V, 1) \quad \text{where } \times \in \{+, -, \dots\}$$

The cost of variables can be looked-up in the variable environment, which is being populated in the let-rule. Arguments result in zero cost.

$$C(\text{variable}) = \text{look-up or } (0, 0) \text{ in case it is an argument}$$

The cost of function application is the cost of the arguments plus the cost of the function, unless it is an application of a recursive function, in which case it is only the cost of the arguments.

$$C(\text{App}(f)) = \sum_{a \in \text{Args}(f)} C(a) + C(f)$$

The cost of a let is the cost of its goal expression, assuming that the cost of the bound variable is the cost of the bound expression.

$$C(\textbf{let } x = e_1 \textbf{ in } e_2) = C(e_2) \text{ assuming that } C(x) = C(e_1) \text{ in } e_2$$

The cost of the map operator is the cost of its body expression.

$$C(\textbf{map } i < [N] \ e) = C(e)$$

Here we make another simplification to the cost model. If we know N at compile time, then the cost of the map would be $N \cdot C(e)$[1], in which case, the contribution to the overall speed-up will be larger. Consider an example when the cost $C(e_1 \circ e_2)$ depends on a vectorised map and a scalar operation:

$$e_1 = (\textbf{map } i < [100] \ e) \qquad e_2 = (a + b)$$

with the associated costs: $C(e_1) = (V, 1)$ and $C(e_2) = (1, 1)$. If N is included we get:

$$C(e_1) = (100 \cdot V, N) \qquad \text{and} \qquad C(e_1 \circ e_2) = (100 \cdot V + 1, 100 + 1)$$

and this value is very close to V. Unfortunately the value of N is not statically known, so we assume that for any map/reduce $N = 1$, which potentially lowers the contribution of the map/reduce into the overall speed-up, as:

$$C(e_1) = (V, 1) \qquad \text{and} \qquad C(e_1 \circ e_2) = (V + 1, 2)$$

[1] In this case $N \cdot C(e)$ is a scalar-tuple multiplication, i.e. every component of the $C(e)$ tuple is multiplied by N.

and this value is close to $V/2$.

We count the cost of reduce similarly to the cost of map. However, we add a speed-up for the reduce-function and we add a penalty, in case the layout-type of reduce-function is $(\triangle,\triangle)\to\triangle$, but the layout type of the reduce is 0. This is the price for reducing vectorised result into scalar.

$$C(\textbf{reduce}\ \mathrm{i} :: idx(k) < N\ (\mathrm{f} :: (\triangle,\triangle)\to\triangle)\ e) = C(e) + C(f) + (0, \xi_{\text{reduce}})$$

In all the other cases, the cost of reduce is a cost of body expression plus the cost of the reduce-function.

$$C(\textbf{reduce}\ i < N\ (f)\ e) = C(e) + C(f)$$

The cost of conditional operators must reflect that in the case when the condition predicate is of layout type $\triangle$, both branches are executed and then results are masked. To compute a cost of condition we must divide the time it takes to execute V instances of the scalar condition by the time it takes to execute the vectorised one. As we do not know the distribution of true/false values of the predicate we can assume the worst case scenario. That is every time the branch with larger number of operations is being executed. In that case for the condition $e \equiv \textbf{if}\ \mathrm{p} :: \triangle\ \textbf{then}\ e_1\ \textbf{else}\ e_2$ the cost will be:

$$\text{Cost}_{pe}(e) = \frac{V\cdot\max(\text{Cost}(e_1),\text{Cost}(e_2))}{\text{Cost}(\vec{e}_1)+\text{Cost}(\vec{e}_2)+\text{Cost}(\text{mask})}$$

where mask is a masking operation that joins results evaluated in both branches. Now, to construct such a value from $C(e_1)$ and $C(e_2)$ we will use:

$$C(e) = (s_e, n_e) \implies \text{Cost}(e) = n_e$$

as cost of scalar operations is one, and:

$$C(e) = (s_e, n_e) \implies \text{Cost}(\vec{e}) = Vn_e\Big/\frac{s_e}{n_e}$$

as s_e/n_e is how many elements per time unit the operation e can process. Putting all together we get the following formula:

$$\begin{aligned}
C(e_1) &= (s_1, n_1)\\
C(e_2) &= (s_2, n_2)\\
n &= \max(n_1, n_2)\\
C(\textbf{if}\ \mathrm{p} :: \triangle\ \textbf{then}\ e_1\ \textbf{else}\ e_2) &= C(p) + n\cdot\left(\frac{n}{n_1^2/s_1 + n_2^2/s_2 + \xi_{\text{mask}}}, 1\right)
\end{aligned}$$

Consider the following condition as an example:

```
if (x > 3)
    a + b + c
else
    x + y
```

Assuming that both branches are vectorised, the cost of the branches will be $(2V, 2)$ and $(V, 1)$ accordingly. The cost of the predicate is $(V, 1)$. The cost of the condition without the predicate is:

$$2 \cdot \left(\frac{2}{2^2/2V + 1^2/V + \xi_{\text{mask}}}, 1 \right) = \left(\frac{4}{3/V + \xi_{\text{mask}}}, 2 \right)$$

When adding the predicate we get:

$$\left(V + \frac{4}{3/V + \xi_{\text{mask}}}, 3 \right)$$

If we assume that ξ_{mask} is zero, the speedup for the vectorised condition approximates to $7V/9$. This value is less than V despite all the operations in the condition are vectorised. This is a price we pay for executing both branches.

If the layout type of the predicate in a conditional expression is zero then the cost is just an average speed-up of the branches.

$$C(\textbf{if } p \textbf{ then } e_1 \textbf{ else } e_2) = C(p) + C(e_1) + C(e_2)$$

Finally, the cost of a function and the cost of the letrec is the cost of its body expression.

Example Let us demonstrate the cost model in application to matrix multiply example:

```
matmul (a, b) =
    map i < [N]
        map j < [N]
            reduce k < [N] (+) a[i ++ k] * b[k ++ j]
```

If any of i, j, k is of layout-type $idx(1)$ multiplication inside the reduce over k is going to produce the cost $(V, 1)$ and, as reduce-function plus will have type $(\triangle, \triangle) \to \triangle$, its cost will be $(V, 1)$ as well. Now, if i or j are vectorised, then the cost of the function will be $(2V, 2)$, as the cost of the map is the cost of its body (by construction of the map cost function). In case vectorisation happens across the index k, the cost of `matmul` will be $(2V, 2 + \xi_{\text{reduce}})$.

Consider another example with a conditional expression inside the map:

```
foo (a) =
    map i < [N]
        if (a[i] > 3) then
            a[i] + 1;
        else
            a[i] / 2;
```

In the case condition is vectorised, the cost of both branches and the predicate will be $(V, 1)$. The overall cost of the function will be

$$(V, 1) + \left(\frac{1}{1/V + 1/V + \xi_{\text{mask}}}, 1 \right) = \left(V + \frac{1}{2/V + \xi_{\text{mask}}}, 2 \right)$$

In the case when all the subexpression in the program are of layout type 0, we expect to get a cost (n, n), where n is a number of primitive operations.

Application In order to apply the cost model to the typings produced by the inference we apply the cost function to every column of the goal expression and chose the column with the maximum speed-up.

Chapter 7

Benchmarks

In Chapters 4 and 5 we used two examples to demonstrate particular aspects of the type inference and transformation algorithms. In this chapter we extend the set of examples and evaluate our approach using seven benchmarks and present runtime figures obtained by compiling transformed SAC programs and their hand-written analogues in C.

7.1 Methodology

The main question of our investigation lies in understanding effects and boundaries of the proposed layout-transformations. We believe that the approach we took of encoding layouts as types can generally improve auto-vectorisation. Therefore, the first question is concerned with absolute performance improvement:

Q1 Given an abstract specification of a program with annotated layout-types, if we derive a C implementation[1] according to our transformation rules, does it have an advantage over the automatically vectorised version obtained from the C implementation where all the layout-types are 0.

In other words — does knowledge about layout-types improve state-of-the-art auto-vectorisation and if so by how much. Now, if it does, the second question is how close we can get to such a performance automatically.

We have introduced all our formalism and transformations using SAC-λ — an abstract language that makes memory management fully implicit and data-parallel constructs fully explicit. Such an abstract specification may come at a price as SAC-λ code-generation has to make memory management explicit again in the resulting C program. Therefore, the second question we would like to answer is:

Q2 What is the price for implementing a program in SAC (as a real-world implementation of SAC-λ) and not in C.

The third question is:

[1]Note that C in this case is a way to express an algorithm at the very low level, yet preserving portability across various hardware architectures.

Q3 What is the principal contribution of vectorisation for a given benchmark.

Intuitively we expect a good vectorisation to deliver a speed-up of factor V, the number of elements that fits in a SIMD register. However, sometimes, vectorisation is not the only contributing factor to the benchmark performance. For example, correct memory organisation may be more beneficial than any vectorisation. For that reason, in order to quantify vectorisation effects, we are going to include in the measurements a reference C implementation with maximum optimisations turned on, except for vectorisation. This is achieved by passing the `-fno-tree-vectorize` flag to GCC, the `-no-vec` flag to ICC and the `-fno-vectorize` to Clang.

7.1.1 Presentation and measurements

For simplicity, the presentation of benchmarks, results and typings will use SaC-λ. For every benchmark we run the inference, apply the cost model and perform transformations. The transformations produce SaC code that we are going to compile again, potentially with slight modifications. Those modifications have only to do with limitations of SaC staging capabilities, not with conceptual problems.

All the measurements are taken by wrapping the core functionality of the benchmark with calls to the high-resolution timer. By doing this we exclude operations like input/output or result verification. We use the `CLOCK_REALTIME` timer with resolution 10^{-9}s supported by linux kernels and available via the `rt` system library. To minimise measurement errors, for some of the benchmarks we run the core functionality several times in a loop. This is always specified in the graphs.

The set of benchmarks is chosen to demonstrate properties and limitations of the proposed approach on practical relevant problems. Vector sum and addition are chosen for their simplicity, which allow us to study vectorisation properties by analysing the assembly output. Matrix multiply is a classical high-performance benchmark which represents a class of linear algebra problems on matrices. At the example of matrix multiply we will study different non-trivial vectorisation possibilities generated by our system. We will also compare the performance achieved by vectorisation with one of the best implementations of matrix multiply using OpenBLAS library. That will demonstrate if there is still a performance gap to be bridged. The N-body and Mandelbrot problems were discussed already. Here we are going to study the performance of the SaC versions and for the example of N-body we are going to check if it would be possible in the generated C code to replace the explicit vector instructions that we currently generate with scalar code and rely on auto-vectorisers to reconstruct it back. The spectral norm benchmark will demonstrate the case of vectorisation preserving the reduction order. We will conclude with the reverse compliment benchmark that will demonstrate how vectorisation achieves very large speed-up with a very counter intuitive implementation.

For our measurements we will use the following compilers:

GCC	`gcc (Gentoo 4.8.3 p1.1, pie-0.5.9) 4.8.3`
ICC	`icc (ICC) 15.0.1 20141023`
CLANG	`clang version 3.5.0 (tags/RELEASE_350/final)`

on the machine "m-i3" from Fig 4.11.

Nomenclature We are going to use the following notation on graphs throughout this chapter:

`-scal` postfix in the name of the benchmark denotes that the code of the benchmark is expressed using scalar types and operations — no explicit vector instructions. In case SAC version is postfixed with `-scal` and a specific compiler it means that the C code that SAC have produced was compiled with the specified compiler;

`-vec` postfix means that the benchmark is written using explicit vector instructions in GCC-based syntax;

`-novec` postfix denotes the benchmark expressed using scalars with further prohibition to auto-vectorise the code.

7.2 Vector addition

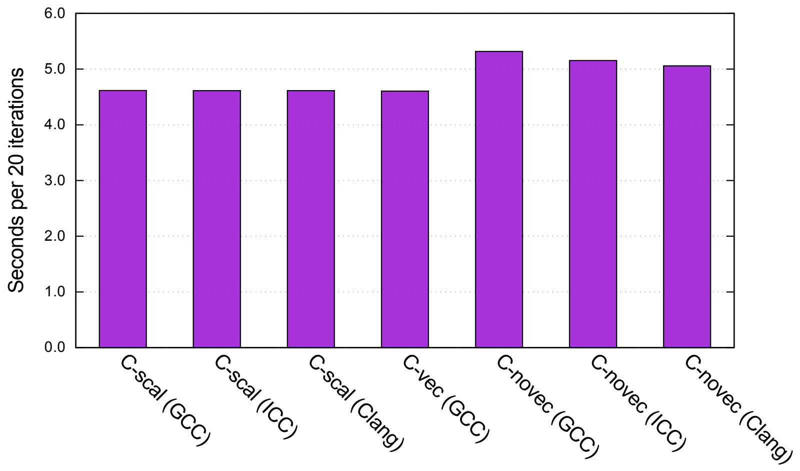


Figure 7.1: Vector addition of two vectors of type float of size $2 \cdot 10^8$ in C.

This is a function that adds two vectors `a` and `b` element-wise.

```
# (float[N], float[N]) → float[N]
vecadd (a, b) =
    map i < [N]   a[i] + b[i]
```

The only typing that allows to use vector instructions in this case is:

```
# vecadd :: (1,1) → 1
vecadd (a :: 1, b :: 1) =
    map i :: idx(1) < [N]  (a[i] :: △ + b[i] :: △)
```

Given the straight-forward nature of this example, our assumption is that C compilers should be able to vectorise such code. We start with running C versions first, the runtime of which can be found in Fig. 7.1.

As it can be seen from Fig. 7.1, vector addition is recognised by the auto-vectorisers of all the C compilers — manually vectorised version `C-vec (GCC)` performs the same as `-vec` postfixed versions. Effects from the vectorisation for vector addition (difference between `-vec` postfixed and `-scal` postfixed versions) are very small (about 10%). This is because vector addition is a memory bound application: we have two vectors to read data from and one vector to write the result. This means that we have three times more memory access operations than arithmetic operations. Note that `-novec` versions perform differently depending on the compiler. This is achieved by different loop unrolling factors. Let us relate those figures to the SaC runtime presented in Fig. 7.2.

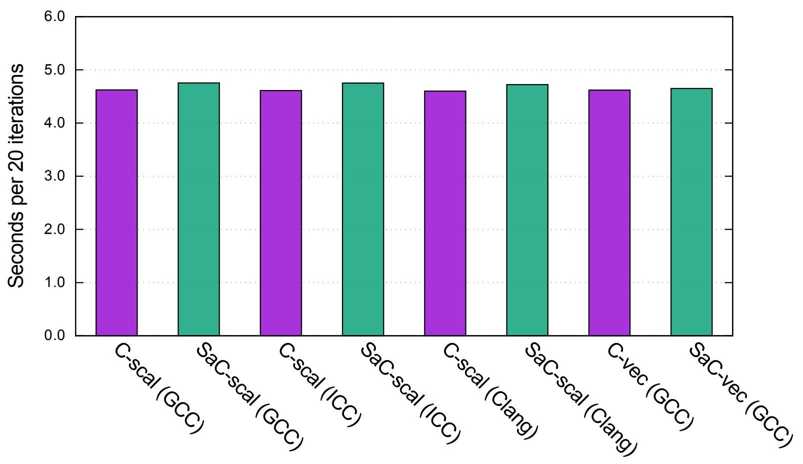


Figure 7.2: Vector addition of two vectors of type float of size $2 \cdot 10^8$ in SaC.

As can be seen, SaC versions perform on a par with C versions, although in the `-scal` case SaC is about 3% slower. This may result from the custom memory allocator that SaC uses, plus the measurements potentially include some memory allocation/deallocation, where in C all the memory operations are done outside the measurements.

Although for vector addition the auto-vectorisation of C compilers is doing a very good job, making proposed transformations unnecessary, this benchmark demonstrates several important properties. First of all, for the scalar version of vecadd SaC generates 100 lines of non-trivial C code of total size 17K characters. Despite this, we can confirm by inspecting assembly code that both for-loops are recognised by auto-vectorisers and are compiled into identical instructions (up to the names of registers):

```
# Inner loop of SaC-scal and C-scal
401128: vmovups 0x0(%r13,%r9,1),%xmm0
40112f: add     $0x1,%r10
401133: vinsertf128 $0x1,0x10(%r13,%r9,1),%ymm0,%ymm0
40113a:
40113b: vaddps (%r14,%r9,1),%ymm0,%ymm0
401141: vmovups %xmm0,(%r8,%r9,1)
```

```
401147: vextractf128 $0x1,%ymm0,0x10(%r8,%r9,1)
40114e:
40114f: add      $0x20,%r9
401153: cmp      %r10,%rbx
401156: ja       401128 <vecadd+0x138>
```

Alignment Another important observation can be obtained if we compare the code above with the assembly output of the vectorised versions:

```
# Inner loop of SaC-vec and C-vec
400ac0: vmovaps (%rbx,%rdx,1),%ymm0
400ac5: vaddps 0x0(%r13,%rdx,1),%ymm0,%ymm0
400acc: vmovaps %ymm0,(%r12,%rdx,1)
400ad2: add      $0x20,%rdx
400ad6: cmp      %r15,%rdx
400ad9: jne      400ac0 <main+0x350>
```

As can be seen from the two assembly snippets, the difference is in the way data is being loaded to and from the registers. The auto-vectorised version uses a combination of `vmovups` unaligned move of the lower part of `ymm0` register followed by insert/extract instructions. The manually vectorised version uses aligned moves (`vmovaps`) instead. For this particular example those effects are barely noticeable. However, our system solves the alignment problem in general. When we decide to vectorise an array, the transformed array will use the vector type as its base type. As a consequence, all the references into vectorised arrays will be aligned, given that the array itself is aligned.

Verifying alignments in C is very challenging mainly because arrays degenerate into pointers. This problem is well known but all the solutions that exist so far [39, 82, 12] to our knowledge propose workarounds that only reduce potentially unaligned access. Most of the proposals are based on peeling loops until some of the references inside become aligned.

7.3 Vector sum

This benchmark computes the sum of vector `a`. In SAC-λ it can be expressed as follows:

```
# (float[N]) → float
vecred (x) =
    reduce i < N (+)  a[i]
```

The only possible vectorisation is:

```
# vecred :: (1) → 0
vecred (x :: 1) =
    reduce i :: idx(1) < N (+) a[i] :: △
```

Similarly to vector addition we assume that the example is simple enough to be recognised by the auto-vectorisers of the C compilers. The runtime results for C versions can be found in Fig. 7.3.

As we can see, vector reduction has been also recognised by the C compilers — `-scal` versions perform better than `-novec` versions. Automatically vectorised versions perform on a par with the hand-written vector version `C-vec (GCC)`. Note

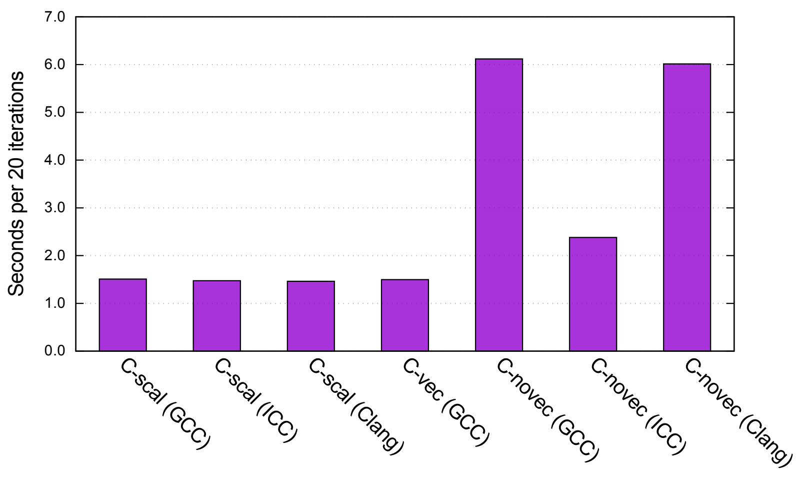


Figure 7.3: Sum of two vectors of type float of size $2 \cdot 10^8$ in C.

that there is a significant difference amongst the `novec` versions. The ICC compiler, produces the binary *without* vector instructions, which yet performs almost three times faster than the binaries produced by other compilers. As we can tell from inspecting the assembly output, ICC unrolls the inner loop, apparently in a such a way that the underlying hardware can run some of the scalar instructions in parallel.

In comparison to vector addition, the vectorisation improves the runtime by 72% (3.5 times) which is due to to much smaller memory intensity. It is one memory read per one arithmetic operation inside the for-loop. Let us now see in Fig. 7.4 how the SaC runtime compares with C.

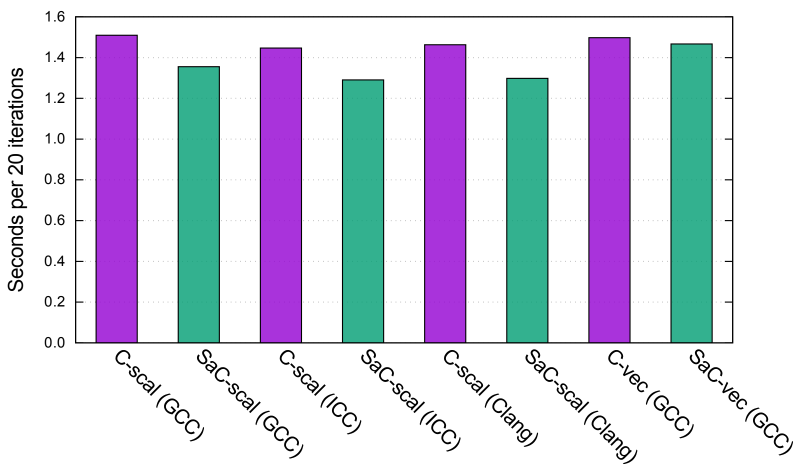


Figure 7.4: Sum of two vectors of type float of size $2 \cdot 10^8$ in SaC.

As in case of vector addition, SaC reference versions `SaC-scal` are recognised by all the auto-vectorisers — they perform on a par with the `C-scal` versions. The transformed version `SaC-vec (GCC)` performs the same as manually vectorised version `C-vec (GCC)`.

Vectorisation of reductions When vectorising reductions we have to realise that final results can differ. For example, when running the benchmark we have initialised the vector with $a_i = \frac{1000}{1+i}$. Here are the results of summation using different compilers:

GCC	ICC	Clang	GCC -fno-tree-vectorize
342697.09375	356200.59375	369396.21875	307598.78125

We can see that all these values differ depending on the order of reduction. This has to do with non-associativity of addition (and multiplication) when using the machine representations of floating point numbers. This is also the reason why optimised reduction is not a default optimisation even at the `-O3` level in C compilers. Practically this means that if we want to produce exactly the same results as the original program, such reduction optimisations must be prohibited. However, for most real world applications the non-determinism caused by reduction order changes are deemed to be negligible.

Further investigation into the variants of results found requires a closer look into the generated assembly code. Here is the code that GCC `-Ofast` generates:

```
/* GCC reduction  */
40091d: vaddps (%rdi),%ymm2,%ymm2
400921: add    $0x20,%rdi
400925: cmp    %rsi,%r9
400928: ja     400919 <main+0x199>
```

ICC does it in the following way:

```
/* ICC reduction  */
401377: vaddps (%r15,%r12,4),%ymm2,%ymm2
40137d: vaddps 0x20(%r15,%r12,4),%ymm0,%ymm0
401384: add    $0x10,%r12
401388: cmp    %r11,%r12
40138b: jb     401377 <main+0x817>
```

Here is the code that Clang generates:

```
/* Clang reduction  */
401010: vmovups -0x60(%rcx),%xmm5
401015: vmovups -0x40(%rcx),%xmm6
40101a: vmovups -0x20(%rcx),%xmm7
40101f: vmovups (%rcx),%xmm8
401023: vinsertf128 $0x1,-0x50(%rcx),%ymm5,%ymm5
40102a: vinsertf128 $0x1,-0x30(%rcx),%ymm6,%ymm6
401031: vinsertf128 $0x1,-0x10(%rcx),%ymm7,%ymm7
401038: vinsertf128 $0x1,0x10(%rcx),%ymm8,%ymm8
40103f: vaddps %ymm5,%ymm1,%ymm1
401043: vaddps %ymm6,%ymm2,%ymm2
401047: vaddps %ymm7,%ymm3,%ymm3
40104b: vaddps %ymm8,%ymm4,%ymm4
401050: sub    $0xffffffffffffff80,%rcx
401054: add    $0xffffffffffffffe0,%rdx
401058: jne    401010 <main+0x7b0>
```

The main difference that we can see is in the number of vector registers that are being used in order to implement reduction. GCC choses to use only one, when ICC and Clang use two and four respectively. The situation also changes if we pass the `-funroll-all-loops` option, which increases performance by about 2%.

7.4 Matrix multiplication

In the previous benchmarks we have seen that the capabilities of auto-vectorisers of C compilers are strong. We have also seen that the proposed transformation system produces comparable results. However, for those two benchmarks, strictly speaking, our system is not necessary. Let us now have a look at a set of more complicated cases where the effects of the proposed transformations will be more visible. We start with a matrix multiply benchmark. Here is how it can be formulated in SAC-λ:

```
# (float[N,N], float[N,N]) → float[N][N]
matmul (a, b) =
    map i < [N]
        map j < [N]
            reduce k < [N] (+) a[i ++ k] * b[k ++ j]
```

Let us start by looking at runtime results in Fig. 7.5.

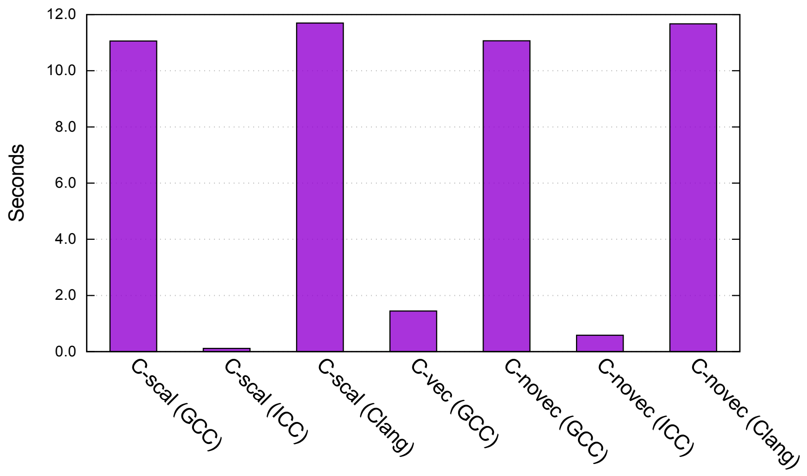


Figure 7.5: Matrix multiplication of two square matrixes of type float, 1000×1000 elements each.

First of all, we can see that GCC and Clang decided not to vectorise the code at all — the `C-scal` and `C-novec` runtimes for those compilers are the same. Secondly, the scalar version of ICC is an order of magnitude faster than the scalar versions of GCC and Clang. This is due to careful memory organisation, which most likely minimises memory stalls. It seems that ICC (starting from version 15.0.0) is recognising matrix-multiplication-like patterns and infers an efficient way to stream the data. Thirdly, manual vectorisation derived from the proposed transformation system substantially decreases the runtime: 7.5 times (the cost model has chosen the `(0,2)->2` layout-type of matmul). Now let us compare these figures with the SAC versions. The runtimes can be found in Fig. 7.6 perform on par with the hand-coded implementations.

First of all, the automatic SAC version `SaC-vec (GCC)` performs the same as manually coded C version `C-vec (GCC)`. Secondly, the C code that the SAC compiler produces is not recognised by ICC — the runtimes of `SaC-scal (ICC)` and `C-scal (ICC)` versions differ. We will discuss the implications of this at the end of this section.

The layout transformations generated by the inference that leads to some vectorisation can be divided into three classes. Further down we are going to consider one key representative of each class. Other variants in the class can be obtained by replacing the layout type 0 with any allowed k layout type — 1 and 2 in the case of two-dimensional arrays. Consider the first variant:

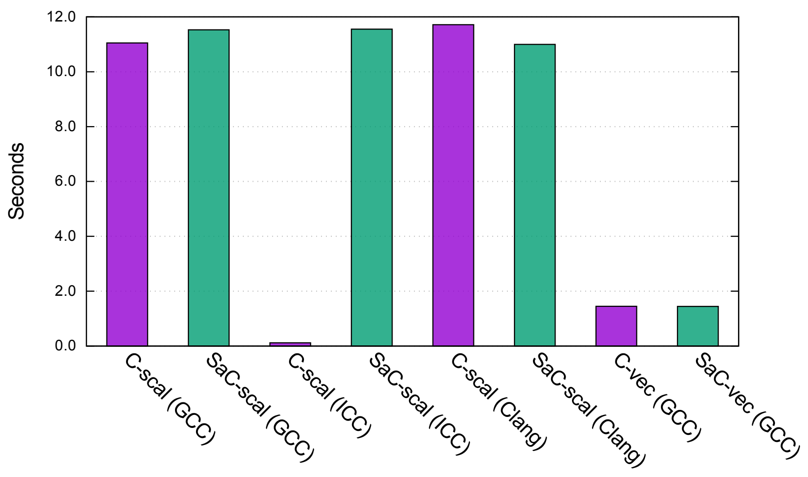


Figure 7.6: Matrix multiplication of two square matrices of type float, 1000 × 1000 elements each.

```
# matmul :: (2,1) → 0
matmul (a :: 2, b :: 1) =
     map i :: 0 < [N]
          map j :: 0 < [N]
               reduce k :: idx(1) < [N] (+)
                        a[i ++ k] :: △ ∗ b[k ++ j] :: △
```

```
for (i = 0; i < N; i++)
     for (j = 0; j < N; j++) {
          vecfloat s = {0f,...,0f};
          for (k = 0; k < N; k += V)
               s += a[i][k:(k+V)]
                     ∗ b[k:(k+V)][j];
          c[i][j] = sum (s);
     }
```

In this case we vectorise array `a` across the rows and `b` across the columns. The result is not vectorised. On the right, we can see how the transformed code for this variant can be expressed in pseudo-C. The inner loop (index k) is the source of vectorisation. We initialise a vector accumulator `s` with zeroes, and we select a vector of V sequential elements from the row starting at `a[i][k]`, (denoted with `a[i][k:(k+V)]`). Then we select a vector of V sequential elements from the column starting at `b[k][j]`, multiply these two vectors and accumulate the result in `s`. After the loop we sum-up the elements of the accumulator which gives us the value for the `c[i][j]` element.

The less obvious typings are: `(0,2)->2` and `(1,0)->1`. Here is how it looks like:

```
# matmul :: (0,2) → 2
matmul (a :: 0, b :: 2) =
     map i :: 0 < [N]
          map j :: idx(1) < [N]
               reduce k :: 0 < [N] (+)
                        a[i ++ k] :: 0 ∗ b[k ++ j] :: △
```

```
for (i = 0; i < N; i++)
     for (j = 0; j < N; j += V) {
          vecfloat s = {0f,...,0f};
          for (k = 0; k < N; k++)
               s += broadcast (a[i][k])
                     ∗ b[k][j:(j+V)];
          c[i][j:(j+V)] = s;
     }
```

At every $[i, j]$ iteration we compute V adjacent row elements of the resulting array starting from `c[i,j]` (denoted with `c[i,j:(j+V)]`). This is achieved by replicating `a[i][k]` V times (the `broadcast` function call).

Finally, we can vectorise the computation of `c` across columns, which gives us the following variant:

```
# matmul :: (1,0) → 1
matmul (a :: 1, b :: 0) =
    map i :: idx(1) < [N]
        map j :: 0 < [N]
            reduce k :: 0 < [N] (+)
                a[i ++ k] :: △ * b[k ++ j] :: 0
```

```
for (i = 0; i < N; i += V)
    for (j = 0; j < N; j++) {
        vecfloat s = {0f,...,0f};
        for (k = 0; k < N; k++)
            s += a[i:(i+V)][k]
                 * broadcast (b[k][j]);
        c[i:(i+V)][j] = s;
    }
```

Note that from our cost model perspective, the last two variants have an advantage over the first layout typing because we avoid an extra fold step on every $[i, j]$ iteration. Also the costs of the latter two variants are the same (as they are symmetric). The runtime of all the three versions are in Fig 7.7.

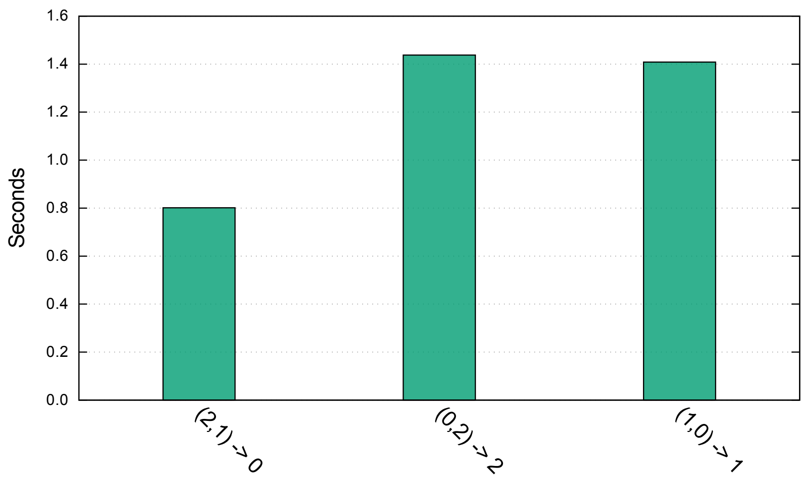


Figure 7.7: Matrix multiplication of two square matrices of type float, 1000×1000 elements each (SAC).

The layout `(2,1)->0` is about two times faster than the other two variants, which suggests that our simplistic cost model did not pick the fastest variant this time. The exact reason of the difference is not obvious and some further investigation is required. Possible explanations could be that scalar reads from an array of layout type 0 with further broadcasting of the element break the instruction pipeline, or the broadcast operation is slow.

However, despite the chosen layout `(0,2)->2` being slower than the alternative, note that `(0,2)->2` preserves the original data in memory up to array paddings. This is a general property of the presented transformation system: an n-dimensional array of layout-type n preserves the flattened representation of the array up to the paddings. As a consequence, such a typing allows one to generate vectorisations on the original layouts by introducing code for boundary conditions of arrays. That might be very useful when we have some external constraints on the array types.

Absolute performance Finally, we would like to emphasise that the speed-up graphs presented for the matrix multiplication are mainly concerned with studying the effects of vectorisation. This is not sufficient, if we are looking for the absolute performance of this benchmark. It is well known, that matrix multiplication is a memory bound problem [49], and the key aspect to the efficient solution is blocking, and only then vectorisation. Blocking is out of the scope of the layout-type system, so in terms of absolute performance there is room for improvement. See Fig. 7.8 for the runtime results.

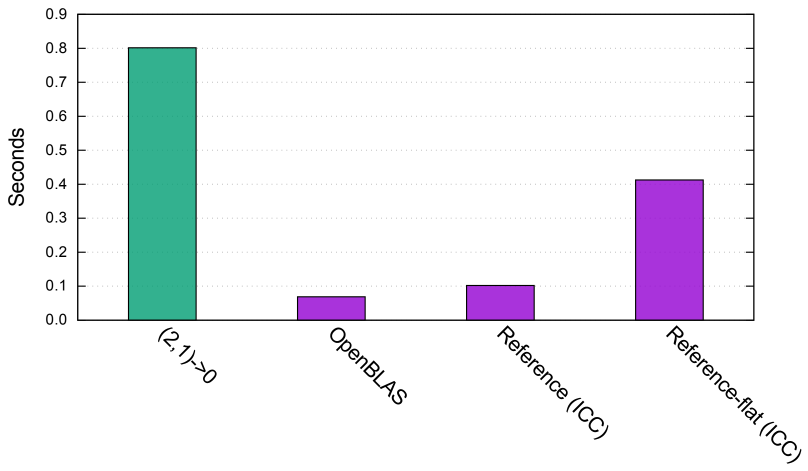


Figure 7.8: Matrix multiplication of two square matrices of type float, 1000×1000 elements each.

As we can see, the OpenBLAS [149] version reduces the runtime we have achieved eight times. It is remarkable that ICC can achieve a runtime `Reference (ICC)` that is only two times slower than the highly-optimised library from the box-standard C specification. To our knowledge this became possible only starting from version 15. However, such an optimisation brings the question — to what extent can we rely on this optimisation if we generate C code?

From Fig. 7.8 we can see that the ICC optimisation is very fragile. For example, the `Reference (ICC)` version was implemented using dynamic two-dimensional arrays:

```
float (*a)[n][n];
float (*b)[n][n];
float (*c)[n][n];
...
for (size_t i = 0; i < n; i++)
    for (size_t j = 0; j < n; j++) {
        float s = 0;
        for (size_t k = 0; k < n; k++)
            s += (*a)[i][k] * (*b)[k][j];
        (*c)[i][j] = s;
    }
```

When we write the code on the flattened arrays, the runtime gets two times slower (the `Reference-flat (ICC)`) version in Fig. 7.7:

```
float *a;
float *b;
float *c;
...
for (size_t i = 0; i < n; i++)
    for (size_t j = 0; j < n; j++) {
        real s = 0;
        for (size_t k = 0; k < n; k++)
            s += a[i*n+k] * b[k*n+j];
        c[i*n+j] = s;
    }
```

Also, if we try to apply Intel's compiler to the SAC generated code, it does not recognise the matrix multiply pattern and produces the same code as GCC or Clang. From the perspective of the code generation targeting C this situation is worrying, as on the one hand C compilers can do very successful optimisations, but on the other hand, an incorrect format of the C code may cancel those optimisations out. Formats are compiler-specific, so the code generation has to guess the right one.

7.5 N-body

We have discussed the runtime of the N-body benchmark based on the hand-written C implementations in Chapter 4. Here we look at the SAC version of the benchmark to see if we can match the hand-written C implementation. Secondly, we are going to test the following hypothesis:

> Would it be possible to avoid explicit GCC vector operations in the generated C code by making an auto-vectoriser reconstruct vector operations. In other words — can we transform the layouts of the data structures in a program, adjust the code using scalar operations and still get the desired vectorisation?

The runtimes are presented in Fig. 7.9.

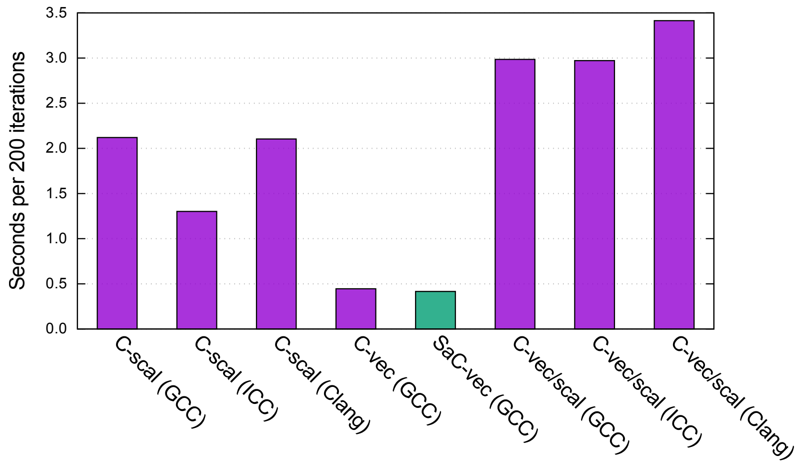


Figure 7.9: N-body problem for 1024 planets, using type float.

The automatically transformed SAC version `SaC-vec (GCC)` matches the performance of the C version `C-vec (GCC)`. The `C-vec` version in Fig 7.9 is the `Reference [N/2]` version from Fig. 4.12. Note that although ICC generates a binary that runs 1.6 times faster than binaries produced by GCC and Clang, the version with modified layout can reduce the runtime at least three times.

As for expressing vector operations with scalars (`-vec/scal` postfix) — it did not work with any of the C compilers and even made the code run slower than the reference implementation. The lesson we can learn from this is that right now we cannot avoid using manual vector instructions, as auto-vectorisers are not as advanced yet to reconstruct vector operations from a scalar representation.

7.6 Mandelbrot

The Mandelbrot example nicely demonstrates the way our system can vectorise recursive functions. Let us start with the formulation of the benchmark in SAC-λ which matches our experiment.

```
# (float, float, float, float, float) → float
iter (i, z0, z1, a0, a1) =
    if i < DEPTH and (z0*z0 + z1*z1 < 4) then
        iter (i + 1,
              z0*z0 - z1*z1 + a0,
              z0*z1 + z1*z0 + a1,
              a0,
              a1)
    else
        i

# (float, float, float, float) → float[HEIGHT,WIDTH]
mandel (height, width, x1, dx, y1, dy) =
    map i < [HEIGHT]
        map j < [WIDTH]
            iter (0.0, 0.0, 0.0, x1+dx*j[0], y1+dy*i[0])
```

The function `mandel` generates a matrix of size HEIGHT×WIDTH using scalar type float. Note that we deliberately use floats instead of integers, as our implementation for the time being only considers vectors of base type float. This is not a conceptual limitation; it is just a restriction of the implementation.

The result that we expect from when transformation is to get the function `iter` to return the type $\triangle$, when one of the maps of the `mandel` function can be vectorised and the vectorised `iter` will be applied.

In order to make it work, we need to introduce a mechanism of projecting a $idx(k)$ layout type in the value domain. In the way we have presented the layout type system, the only operation which is allowed on $idx(k)$ is selection into arrays of type k. This restriction can be relaxed, as we will discuss in Section 8.1, and to do so formally we are going to introduce a layout-type conversion primitive for the idx types.

A new primitive function `idxtoval` serves this purpose. Semantically (in the scalar case) this is just an identity function:

$$\textsc{IdxToVal}\quad \frac{\rho \vdash e \Downarrow v}{\rho \vdash \textbf{idxtoval}\ (e) \Downarrow v}$$

The inference rule for **idxtoval** should convert $idx(k)$ types into $\triangle$ and bypass 0 types. The rule looks like:

$$\textsc{IdxToVal}\quad \frac{\mathcal{F},\mathcal{E} \vdash e : \langle \tau_1, \dots, \tau_n \rangle \qquad \underset{1 \le i \le n}{\rho_i} = \begin{cases} \triangle & \tau_i = idx(k) \\ 0 & \tau_i = 0 \end{cases}}{\mathcal{F},\mathcal{E} \vdash \textbf{idxtoval}\ (e) : \langle \rho_1, \dots, \rho_n \rangle}$$

Finally, the transformation rule for the index vector of $idx(k)$ layout type looks like:

$$\mathcal{T}(\textbf{idxtoval}\,(e \equiv [e_1,\dots,e_n])) = \begin{cases} \begin{bmatrix} [e_1, & e_1, & \dots, & e_1], \\ [e_2, & e_2, & \dots, & e_2], \\ & & \dots & \\ [V\cdot e_k + 0, & V\cdot e_k + 1, & \dots, & V\cdot e_k + V - 1] \\ & & \dots & \\ [e_n, & e_n, & \dots, & e_n] \end{bmatrix} & e :: idx(k) \\ e & e :: 0 \end{cases}$$

We reconstruct V values of the indexes in the original program that correspond to the index vector of type $idx(k)$. As a result of the application of **idxtoval** we get a vector of layout-type $\triangle$ and as an index vector is always a one-dimensional array, the resulting vector is going to be a two-dimensional array with the inner dimension being V.

With these definitions at hand, we can adjust the `mandel` function like this:

```
mandel (height, width, x1, dx, y1, dy) =
    map i < [HEIGHT]
        map j < [WIDTH]
            iter (0.0,
                  0.0, 0.0,
                  x1+dx*idxtoval(j)[0], y1+dy*idxtoval(i)[0])
```

As we expected, this produces two typings. The first one is coming from vectorisation of the outer map in the `mandel`:

```
# iter :: (△, △, △, 0, △) → △
iter (i :: △, z0 :: △, z1 :: △, a0 :: 0, a1 :: △) =
    if (i < DEPTH) :: △ and (z0*z0 + z1*z1 < 4) :: △ then
        iter ((i + 1) :: △,
              (z0*z0 - z1*z1 + a0) :: △,
              (z0*z1 + z1*z0 + a1) :: △,
              a0 :: 0,
              a1 :: △)
    else
        i :: △

# mandel :: (0, 0, 0, 0) → 1)
mandel (height}, width, x1, dx, y1, dy) =
    map i :: idx(1) < [HEIGHT]
        map j :: 0 < [WIDTH]
            iter (0.0 :: △, 0.0 :: △, 0.0 :: △,
                  x1+dx*(idxtoval(j) :: 0)[0],
                  y1+dy*(idxtoval(i) :: △)[0])
```

The other variant stems from the vectorisation of the inner map of the `mandel`:

```
# iter :: (△, △, △, △, 0) → △
iter (i :: △, z0 :: △, z1 :: △, a0 :: 0, a1 :: △) =
    if (i < DEPTH) :: △ and (z0*z0 + z1*z1 < 4) :: △ then
        iter ((i + 1) :: △,
              (z0*z0 - z1*z1 + a0) :: △,
              (z0*z1 + z1*z0 + a1) :: △,
              a0 :: 0,
              a1 :: △)
    else
```

```
                i :: △

      # mandel :: (0,0,0,0) → 2)
      mandel (height}, width, x1, dx, y1, dy) =
           map i :: 0 < [HEIGHT]
                map j :: idx(1) < [WIDTH]
                     iter (0.0 :: △, 0.0 :: △, 0.0 :: △,
                           x1+dx*(idxtoval(j) :: △)[0],
                           y1+dy*(idxtoval(i) :: 0)[0])
```

Now, similarly to N-body, we are going to consider how the SAC implementation compares to a hand-written C implementation. See Fig. 7.10.

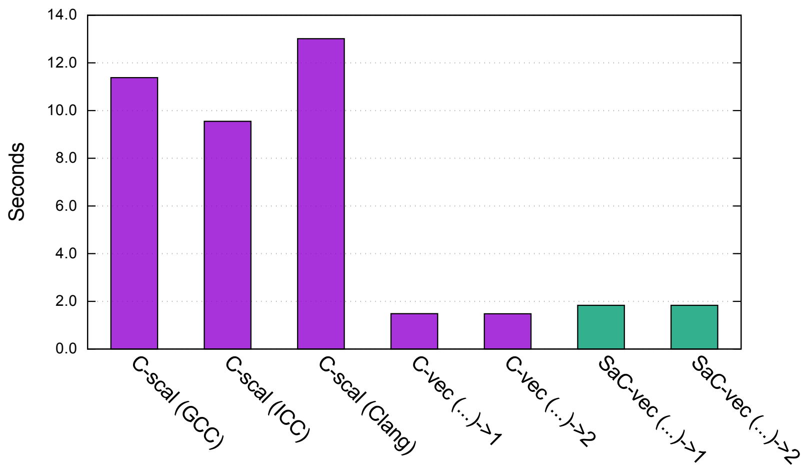


Figure 7.10: Mandelbrot 2048×2048 with depth 4096. Compiler: GCC `-Ofast -march=native -mtune=native`.

First of all, it does not matter which map of the `mandel` we decide to vectorise — the bars with postfixes `(...)->1` and `(...)->2` perform the same. This is because the Mandelbrot benchmark is compute bound, and the effects of memory access are negligible. Secondly, hand-written C code is about 20% faster. Let us investigate where the runtime difference is coming from.

First of all, the exit from the recursive function, that is ${z_0}^2 + {z_1}^2 < 4$ in the SAC code, is expressed using the following primitive:

```
      vec_mask_t cond = z0*z0 + z1*z1 < __simd_broadcast__ (4.0f);
      ...
      if (__simd_any__ (cond)) {
           ...
      }
```

We carry on with a recursive call if any of the elements of the resulting vector are less than four. The implementation of `__simd_any__` that we use in SAC looks like:

```
      static inline bool __simd_any__ (vec_mask_t x) {
        return (bool)(x[0] | x[1] | x[2] | x[3] | x[4] | x[5] | x[6] | x[7]);
      }
```

for the cases when V is 8. The hope was that on Intel architectures supporting AVX extensions such a reduction can be replaced with a `VMOVMSKPS` instruction (or an intrinsic `_mm256_movemask_ps`), which computes a bit mask of the argument. This means that `__simd_any__` can be implemented as:

```
static inline bool __simd_any__ (vec_mask_t x) {
  return (_mm256_movemask_ps ((__m256)x) & 255) != 0;
}
```

As it does not happen, the original SAC code runs slower. Such an optimisation should be added to the GCC vector framework.

The second part of the slowdown lies in a shortcut of the masking operation. On every recursive call of `iter` we add one to `i` if the predicate of the surrounding condition holds. In SAC we implement this using masking in the following way:

```
...
if (__simd_any__ (cond)) {
    i1 = i;
    i2 = i + __simd_broadcast__ (1);
    i3 = __simd_select__ (cond, i1, i2);
    iter (i3, ...);
}
```

The observation made of the hand-written C code is that the `__simd_select__` call can be avoided in the following way:

```
...
if (__simd_any__ (cond)) {
    i3 = i + (__simd_broadcast__ (1) & cond);
    iter (i3, ...);
}
```

This is valid, because zero is a neutral element of addition, and we select from `i + 1` and `i`. This duality was mentioned in [12] in Section 3.8 — if the semantics of `__simd_select__ (mask, a, b)` is:

$$(m \land a) \lor (\neg m \land b)$$

then the following substitution is valid:

$$(m \land (e \oplus x)) \lor (\neg m \land e) \quad \rightarrow \quad e \oplus (m \land x)$$

for $\oplus \in \{+, -\}$. Unfortunately such an optimisation is not implemented either in SAC or in C.

To conclude: if we fix both issues, then the runtime of the C and SAC vectorised versions will match.

7.7 Spectral norm

This benchmark[2] is computing a spectral norm of an infinite matrix A with elements:

$$a_{11} = 1, a_{12} = 1/2, a_{21} = 1/3, a_{13} = 1/4, a_{22} = 1/5, a_{31} = 1/6, \ldots$$

The formulation in SAC-λ looks as follows:

```
eval_A (i, j) =
    1.0 / ((i + j) * (i + j + 1) / 2 + i + 1)
```

[2] See http://benchmarksgame.alioth.debian.org/u32/performance.php?test=spectralnorm for more details.

```
eval_A_times_u (u) =
    map i < [N]
        reduce j < [N] (+) eval_A (idxtoval (i), idxtoval (j)) * u[j]

eval_At_times_u (u) =
    map i < [N]
        reduce j < [N] (+) eval_A (idxtoval (j), idxtoval (i)) * u[j]

eval_AtA_times_u (u) =
    eval_At_times_u (eval_A_times_u (u))
```

The structure of this benchmark is very similar to that of the Mandelbrot problem. In functions `eval_A_times_u` and `eval_AtA_times_u` there is a reduce construct inside the map, where the body of the inner reduce does an arithmetic operation on indexes. The operation is defined in the function `eval_A` and it is much simpler than the `iter` function of mandelbrot, as it is not recursive and does not have conditions.

Vectorisation of such a benchmark is straight-forward. All three compilers figure out that both inner reduces can be executed in parallel. They all generate code for that, and probably it is the best vectorisation for this problem. See the runtime in Fig. 7.11.

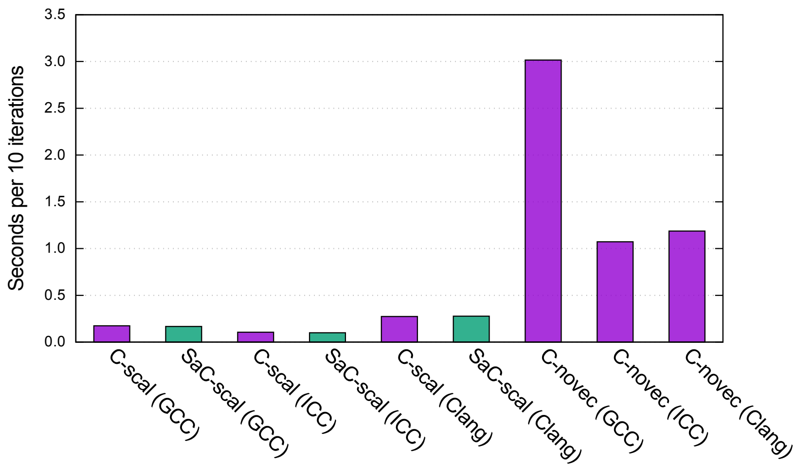


Figure 7.11: Spectral norm $N = 2000$, element type is float. Compilers: GCC/ICC/-Clang `-Ofast -march=native -mtune=native`.

We can see that all the compilers managed to vectorise this code, even in case when it was generated by SaC. It is also remarkable, that ICC is able to perform three times faster without using any vector instructions. Nevertheless, vectorisation is achieved and our proposed framework is not necessary in this case.

However, the results in Fig. 7.11 are obtained using `-Ofast` which allows one to perform unsafe floating-point optimisations. When we want to preserve the order of reduction and use safe math, all the compilers fall back to scalar mode. It is still possible to preserve reductions and use vector operations at the same time. All we need to do is to vectorise the outer-most maps in functions `eval_A_times_u` and `eval_AtA_times_u` (the maps over index `i`).

In Fig. 7.12 we present runtime figures for the variants preserving reductions and using safe math. That is: compilation with `-O3` enforcing IEEE 754 compliance, i.e. passing

`-frounding-math -fsignaling-nans` and `-fno-unsafe-math-optimizations` to GCC/Clang and passing `-fp-mode strict` to ICC.

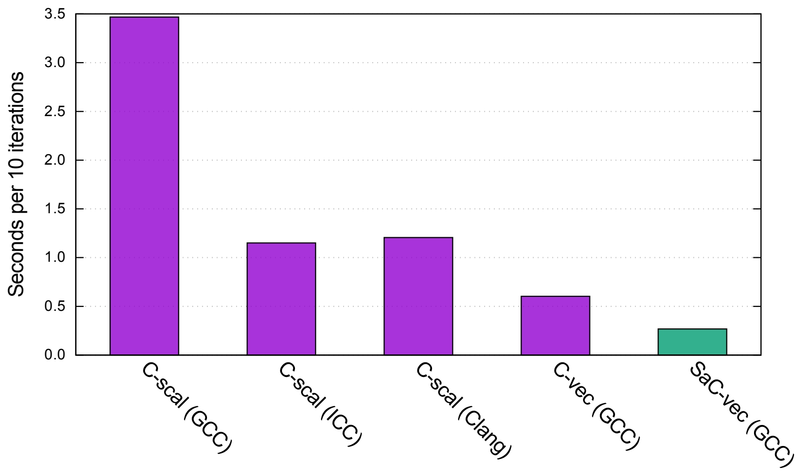


Figure 7.12: Spectral norm $N = 2000$, element type is float.

The `C-vec` and `SaC-vec` versions that are vectorising the outer loop are faster than any automatically produced version. SaC version is twice as fast as C version, which is not a better vectorisation, but an effect of high-level optimisations reducing the memory footprint.

7.8 Reverse complement

The final benchmark[3] demonstrates how modern hardware may break the standard algorithmic intuition of a programmer. Here is the code of the reverse compliment benchmark in SaC-λ.

```
revcomp (a) =
    map i < [N]
        if a[i] = 'A' then
        else if a[i] = 'A' then 'T'
        else if a[i] = 'B' then 'V'
        else if a[i] = 'C' then 'G'
        else if a[i] = 'D' then 'H'
        else if a[i] = 'G' then 'C'
        else if a[i] = 'H' then 'D'
        else if a[i] = 'K' then 'M'
        else if a[i] = 'M' then 'K'
        else if a[i] = 'R'
                or a[i] = 'T' then 'Y'
        else if a[i] = 'U' then 'A'
        else if a[i] = 'V' then 'B'
        else if a[i] = 'Y' then 'R'
        else a[i]
```

Given a string where every character belongs to a certain set, the algorithm replaces every character in the set using a set of rules, which can be represented as a table where one character replaces the other.

[3]See http://benchmarksgame.alioth.debian.org/u32/performance.php?test=revcomp#about for more details

Note that as the implementation of the transformations currently supports only arrays of floats, we cannot derive the transformed version of the problem in SAC automatically. However, we can obtain layout types for the vectorised `transform` function by substituting `char` with `float` and running the type inference. We will use those layout types as a guideline to the first implementation.

The standard intuition suggests that a look-up table implemented as a switch operator or as a constant array in memory should provide the best possible performance. In C, it would be common to implement it as:

```
void transform (char *l, size_t len)
{
  for (size_t i = 0; i < len; i++)
    {
      switch (l[i])
        {
      case 'A': l[i] = 'T'; break;
      case 'B': l[i] = 'V'; break;
      case 'C': l[i] = 'G'; break;
      case 'D': l[i] = 'H'; break;
      case 'G': l[i] = 'C'; break;
      case 'H': l[i] = 'D'; break;
      case 'K': l[i] = 'M'; break;
      case 'M': l[i] = 'K'; break;
      case 'R': l[i] = 'Y'; break;
      case 'T':
      case 'U': l[i] = 'A'; break;
      case 'V': l[i] = 'B'; break;
      case 'Y': l[i] = 'R'; break;
      /* We can leave those cases as it is
         a mapping to the same value.

         case 'N': l[i] = 'N'; break;
         case 'W': l[i] = 'W'; break;
         case 'S': l[i] = 'S'; break;   */
        }
    }
}
```

If our architecture supports SIMD operations on characters, we can vectorise the function `revcomp`, i.e. to type a function with $(\triangle) \to \triangle$. This is the kind of vectorisation inferred by the inference. The induced transformation will split the input string into vectors and for each vector we can update all the elements that are equal to a pattern. Here is an example if we want to replace all characters 'a' with a character 't':

```
...
vecchar x = {'a','b','a','c'};
vecchar mask_a = (x == 'a')               /* will result in {-1,  0, -1,  0 } */
vecchar y = 't' & mask_a | x & ~mask_a;  /* will result in {'t','b','t','c'} */
```

This can be applied to all the cases and we will have the first version of the vectorised transformation, which can be expressed in C with GCC extensions as follows:

```
void vec_transform (char *l, size_t len)
{
  assert (len % V == 0);
  for (size_t i = 0; i < len; i += V)
    {
      vecchar vc = *(vecchar *)&l[i];
      vecchar vc1 = vc;
      vecchar vcA = (vc == 'A'); vc1 = (vcA & 'T') | (~vcA & vc1);
      vecchar vcB = (vc == 'B'); vc1 = (vcB & 'V') | (~vcB & vc1);
      vecchar vcC = (vc == 'C'); vc1 = (vcC & 'G') | (~vcC & vc1);
      vecchar vcD = (vc == 'D'); vc1 = (vcD & 'H') | (~vcD & vc1);
      vecchar vcG = (vc == 'G'); vc1 = (vcG & 'C') | (~vcG & vc1);
      vecchar vcH = (vc == 'H'); vc1 = (vcH & 'D') | (~vcH & vc1);
      vecchar vcK = (vc == 'K'); vc1 = (vcK & 'M') | (~vcK & vc1);
      vecchar vcM = (vc == 'M'); vc1 = (vcM & 'K') | (~vcM & vc1);
      vecchar vcR = (vc == 'R'); vc1 = (vcR & 'Y') | (~vcR & vc1);
      vecchar vcTU = ((vc == 'T') | (vc == 'U')); vc1 = (vcTU & 'A')
                                                   | (~vcTU & vc1);
      vecchar vcV = (vc == 'V'); vc1 = (vcV & 'B') | (~vcV & vc1);
      vecchar vcY = (vc == 'Y'); vc1 = (vcY & 'R') | (~vcY & vc1);

      *(vecchar *)&l[i] = vc1;
    }
}
```

Please note that in the above version we apply `select` operation on every condition. We also had to copy the vector `vc` into `vc1` to avoid manifold replacements, i.e. if an algorithm replaces `'a'` with `'b'` and `'b'` with `'c'`, by not copying `vc` into `vc1` the string `"aabb"` will turn into `"cccc"` instead of `"bbcc"`.

For this particular benchmark, we can perform an optimisation based on the knowledge that the range of values we can find in the input string is restricted and small. In that case we can drop the `~m` part of the select operation, as we will have a mask for every possible value. Here is the code for such an approach:

```
void vec_transform_set (char *l, size_t len)
{
  assert (len % V == 0);
  for (size_t i = 0; i < len; i += V)
    {
      vecchar vc = *(vecchar *)&l[i];

      vecchar vcA = (vc == 'A') & 'T';
      vecchar vcB = (vc == 'B') & 'V';
      vecchar vcC = (vc == 'C') & 'G';
      vecchar vcD = (vc == 'D') & 'H';
      vecchar vcG = (vc == 'G') & 'C';
      vecchar vcH = (vc == 'H') & 'D';
      vecchar vcK = (vc == 'K') & 'M';
      vecchar vcM = (vc == 'M') & 'K';
      vecchar vcR = (vc == 'R') & 'Y';
      vecchar vcTU = ((vc == 'T') | (vc == 'U')) & 'A';
      vecchar vcV = (vc == 'V') & 'B';
      vecchar vcY = (vc == 'Y') & 'R';
      vecchar vcrest = ((vc == 'N') | (vc == 'S') | (vc == 'W')) & vc;

      vc  = vcA | vcB | vcC | vcD | vcG | vcH  | vcK
            | vcM | vcR | vcTU | vcV | vcY | vcrest;

      *(vecchar *)&l[i] = vc;
    }
}
```

As can be seen the number of vector operations has been significantly reduced. Let us now see if any of the proposed vectorisations improve the runtime.

The measurements for the reference version `Reference` compiled with all three C compilers, the straight-forward vectorisation `VecTransform` and for the optimised vectorisation `VecTransformSet` are presented in Fig. 7.13.

The speed-up of the vectorised versions is very impressive (about 20 times). `VecTransform` and `VecTransformSet` runtimes are 0.07 and 0.06 seconds accordingly.

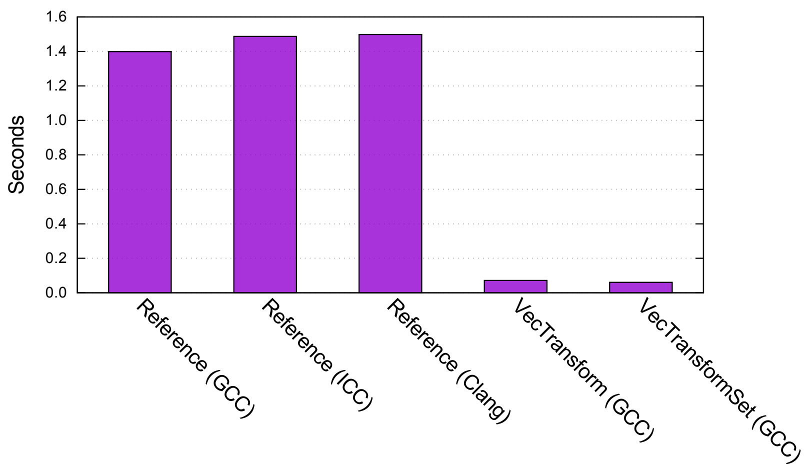


Figure 7.13: Reverse complement on a 100 MB random string, using 16-character vector type.

In order to automate such a pattern we have to:

1. Find the cost model that can estimate if vectorisation of such nested conditions is not harmful; and
2. Introduce a way in the language to express a enum-like type in order to get the optimised transformation.

The main conclusion from this benchmark is that efficient implementation of some algorithms (even trivially simple like this one) might be counter-intuitive. That is why we believe that the form of expression of an algorithm must be as compiler-friendly as possible. In that case we can hope that automatic tools will be able to deduce the best possible implementations not only for today's hardware but also for the future hardware as well.

7.9 Summary

In this chapter we have seen how the application of the proposed inference and transformation systems. We have seen a 3 times improvement for the N-body benchmark and 6.5 times improvement for Mandelbrot examples, when compared with the Intel compiler with all the optimisations turned on. We have seen an 8 times improvement for matrix multiplication resulting only from better vectorisation in the case of GCC. We have demonstrated a 2 times improvement comparing with the Intel compiler for the spectral norm benchmark in the case of preserved order of reductions. Finally we have seen a proof of concept implementation of the reverse complement benchmark which results in a 20 times speed-up in comparison with the Intel compiler. From that we can conclude that the answer for Q1 is yes — we can improve existing state-of-the-art auto-vectorisers if we consider data layouts.

As for Q2 we have seen that, for the given set of benchmarks, automatically derived SaC versions perform similarly to manually encoded C versions and we identified several optimisations that would be desirable to have in SaC and GCC to make the runtimes identical.

The contribution of vectorisation in the overall runtime varies from benchmark to benchmark, but for most of the benchmarks it is very significant.

As a side-effect of the proposed transformation we solved an alignment problem in a systematic way.

We have also observed that the proposed cost model is too simple, at least for the hardware that we ran our experiments on. The cost model helps to eliminate layout types that do not lead to vectorisation, but in order to chose the right vectorisation it has to be improved.

Chapter 8

Conclusions and future work

Most programming languages fix the way data structures are mapped into the memory. In this thesis we relax this constraint and allow every data structure in a program to have its own data layout. By doing so we have demonstrated that carefully chosen data layouts for arrays have a very noticeable impact on program vectorisation. One of the consequences when relaxing a data layout mapping strategy is a large set of possible data layout configurations per program. As every array in principle can be mapped in memory in at least $n!$ different ways, where n is a number of elements, for the overall program we have to consider a Cartesian product of the individual mappings of each array. Besides that, changing the data layout of a data structure impacts the way this data structure has to be referenced and traversed. As a consequence the program has to be modified to adhere to the new layouts and we are required to verify if functions and operators are compatible with the data structures of given layouts. Finally we have to find suitable data abstractions to express the resulting programs.

The main insight we discovered while searching for a solution to the above problem is that using a type system for representing data layouts not only allows an elegant and compact formulation of the problem but this approach also has a number of important advantages. First of all, type checking guarantees consistency of data layouts in a given program. Secondly, the ability to type check functions in isolation solves the problem of separate compilation and allows one to split a program into various modules, where each module can be type checked separately. Finally, type inference techniques allow us to reconstruct data layouts automatically. Further code transformations, that are required for a program to adhere to the inferred data layouts, are based on types. This makes it easier to guide the code generation and reason about correctness of the transformations.

The main technical contributions of this thesis lie in the implementation of the type inference and code generation in SAC; providing a proof that the transformations preserve semantics of the original program; extending C language with vector operations, as discussed in Chapter 3, and implementing them in the context of GNU GCC; evaluating the quality of vectorisation delivered by the proposed approach on a set of benchmarks.

We compare the impact of our transformation for several examples that are known to be difficult to vectorise. Despite being a prototypical implementation, we have seen that the system delivers results that outperform autogenerated code and are on a par with manually written one. We believe that the nature of the chosen benchmarks represents a large class of applications all of which could benefit from the proposed technique. The transformation happens in a fully automatic manner which suggests that the thesis we have formulated in Chapter 1 is satisfied.

We believe that by bringing the proposed approach in the existing widely used compilers will lead to significant advances in closing performance gap for the cases when it is caused by underutilisation of SIMD instructions.

In future research we would like to:

1. understand the limits of the proposed layout-type-based analysis and apply it to more complex vectorisation patterns or even outside the vectorisation problem.

2. apply our framework in the context of languages like C or Fortran to bring the power of the proposed technique into existing scientific applications.

3. build a more reliable and configurable cost model that will consider hardware properties and user annotations.

4. allow for runtime layout modifications so we can change layout dynamically for the benefit of a hotspot.

8.1 Layout types generalisations

First we are going to consider the way layout types can be extended to cover the cases when data layout transformations get more complicated than tilings across one of the array dimensions. The real power of the layout type system as we see it is in its genericity with respect to the transformation each layout type encodes.

Let us present layout types from a different perspective than we have done so far. First of all, let us note that the formal representation of arrays can be approached from at least two different perspectives; on the one hand an array is a tuple of values:

$$Arr : \langle v_1, \ldots, v_n \rangle, \text{where } v_i \text{ is a value}$$

This representation reflects the way arrays are stored in memory and the only possible indexing of such an array is a position in the tuple. On the other hand arrays can be seen as a function from indexes into values:

$$Arr : Idx \to Val$$

where Idx is a finite set of indexes and Val is a finite set of values. All the values in Val are indexed at least by one index from Idx which makes Arr a surjection. Also, to relate this representation to the first one, we can say that:

$$Val = \bigcup_i \{v_i\}$$

Now, from a programmer or program perspective the first representation might not even be relevant, as algorithms are written in terms of indexes and selection on a certain index implies a value. The process of mapping an index to the position in the tuple is irrelevant. Conceptually it is not important. However a mapping implies a cost, especially if it has to happen at runtime.

Vectorisation is a process when selections start to operate not on a single index, but on a tuple of indexes to yield a tuple of values. So if in the scalar case we have:

$$\text{sel} : Arr \times Idx \to Val$$

in the vectorised case we have:

$$\text{vsel} : Arr \times \langle\underbrace{Idx, \ldots, Idx}_{V \text{ times}}\rangle \to \langle\underbrace{Val, \ldots, Val}_{V \text{ times}}\rangle$$

Semantically these functions are related as follows:

$$\text{vsel}(a, \langle i_1, \ldots, i_V\rangle) \equiv \langle \text{sel}(a, i_1), \ldots, \text{sel}(a, i_V)\rangle$$

That is the most generic view of vectorisation. In reality this is usually not very efficient. The point of vectorisation is to perform concurrent operations on the values inside the tuple. That implies that every element of a non-vectorised array resides in only one tuple. Otherwise we have to deal with locking, race conditions, element invalidation or similar problems that will dramatically increase complexity of the model. If index tuples never share an index we can regroup indexes of the original array into groups of V-element tuples, index the tuples and reason about selection on those new indexes. Formally the vectorisation of an array $a : Idx \times Val$ is:

$$\vec{a} = \text{Vec}(a) \quad \text{where} \quad \vec{a} : Idx_V \to \langle\underbrace{Val, \ldots, Val}_{V \text{ times}}\rangle$$

where Idx_V is a set of new indexes that have a mapping into the original set Idx. The vectorisation is defined by this mapping. Let us call it ξ. It maps every $iv \in Idx_V$ into a V-element tuple of indexes from the original index set Idx of the array a:

$$\forall iv \in Idx_V : \xi(iv) = \langle idx_1^{iv}, \ldots, idx_V^{iv}\rangle, \quad \text{where} \quad idx_i^{iv} \in Idx$$

We require vectorisation to map all the indexes of the original array into the vectorised one in a unique way. Formally it can be expressed as:

$$\forall i \in Idx \; \exists iv \in Idx_V \; \exists j \in \{1, \ldots, V\} : \xi(iv)[j] = i$$

where $\xi(iv)[j]$ refers to the j-th element of the tuple $\xi(iv)$. The reverse property should hold as well:

$$\forall iv \in Idx_V \; \forall 1 \le j \le V \; \exists i \in Idx : \xi(iv)[j] = i$$

Finally, uniqueness is defined as:

$$\xi(iv)[j] = i \wedge \xi(jv)[k] = i \implies iv = jv \wedge j = k$$

and

$$i = \xi(iv)[k] \wedge j = \xi(iv)[k] \implies i = j$$

Now let us consider what happens when the number of elements in the array is not divisible by V. We can still regroup indexes in V-element tuples, but we will need a notion of holes in both index domain and in the value domain. We could regroup indexes of an array the a in M groups of V elements where:

$$\frac{|Idx|}{V} \leq M \leq |Idx|$$

which allows us to have holes in vectors, but at the same time does not allow vectors to be empty:

$$\xi(iv) = \langle idx_1^{iv}, \ldots, idx_V^{iv} \rangle, \quad \text{where } idx_i^{iv} \in Idx \cup \{\epsilon\}$$

All the properties we have defined for ξ must hold in this case as well for non-empty elements.

Finally, for all practical purposes ξ must have a closed form, as we cannot afford to have a lookup table for index translation. Such a mapping ξ is the essence of any layout type, assuming that linearisation of Idx_V is fixed.

8.1.1 Current types

Using this idea we can describe current layout types as follows. In the case of a multidimensional array A of shape $[s_1, \ldots, s_m]$, vectorisation of type k yields M groups of V-element tuples of indexes where:

$$M = s_1 \cdot \dots \cdot \lceil s_k/V \rceil \cdot \dots \cdot s_m$$

Idx is defined as a set of m-element tuples ranging from $[0, \ldots, 0]$ to $[s_1 - 1, \ldots, s_m - 1]$. The vectorised set of indexes is also an m-element tuple, ranging from $[0, \ldots, 0]$ to $[s_1 - 1, \ldots, \lceil s_k/V \rceil - 1, \ldots, s_m - 1]$. The mapping represented by the layout type k is defined as:

$$k([i_1, \ldots, i_m]) = \langle [i_1, \ldots, Vi_k + 0, \ldots, i_m], \ldots [i_1, \ldots, Vi_k + V - 1, \ldots, i_m] \rangle$$

Let us consider some of the properties of such a vectorisation. First of all we use the same representation for vectorised and non-vectorised indexes, which is mainly chosen for convenience reasons, but in principle is not fixed. Secondly, we are free to chose a mapping for the vectorised indexes (assuming it has a closed form) as it will not impact vectorisation. Now, for all practical purposes, we want to linearise Idx_V in an order of its potential traversal, so it will be natural to put elements in some lexicographical order, for example row-major order.

8.1.2 Adding new type

When we are adding a new kind of layout type, we have to consider several aspects.

Composability Layout types usually come in families because of the way multidimensional arrays are constructed. That is, if we have arrays e_i of dimension d, then the following language construct:

$$[e_1, \dots, e_n]$$

is going to produce an array of dimension $d + 1$. The same property holds for the `map` construct. That is:

$$\textbf{map}\, i < u \quad e_i$$

is perfectly legal and will produce an array of dimension $d + \mathcal{D}(u)$.

This is possible due to the nature of index linearisation, which allows one to add an extra dimension without changing the order of the existing elements in the array. The layout types presented in this thesis also support this property, as extra dimensions on the vectorised arrays do not change the existing layout.

However, it does not have to be like that for every layout type family. For example one could think of aggressive element reshuffling which will not be able to easily accommodate an extra dimension. For example, think of layout type that transposes an array. In this case we can prohibit array constructors out of elements of a certain type as well as maps that result in non-scalar body expressions.

Dual index types In our setup every layout type k has a dual index type $idx(k)$, which indicates that we traverse an array not scalar by scalar but vector by vector. Selections on such a type will result in a $\triangle$ type.

A new family of layout types has to come with its own index types that are going to be dual to the array layout types and with a version of a $\triangle$ type which may or may not be compatible with the original $\triangle$ layout type.

Values of index types Index types are special, as the only way we use them is to mark index variables of map/reduce constructs and see if selection happens on such a type. However, index types can potentially be used as values. For example consider the following expression:

$$\textbf{map}\, \mathrm{i} :: idx(k) < u \quad i$$

The questions are do we allow this and what would be the resulting type? In case of the layout types used in the thesis, this is straight forward to support. However in general case it might be not that simple.

Coexisting with other layout types Further questions include what would be the semantics of the operations that are performed on the objects belonging to various layout type families. Do we allow this and if so, how do we make sure that it is sound?

8.1.3 Reverse-friendly layout

As an exercise let us consider a layout type to perform fast reverse operations on strings. Now, in order to develop an intuition, let us consider an in-place reverse operation implemented in C:

```
char s[N];

for (size_t i = 0; i < N/2; i++)
    swap (s[i], s[N-i-1]);
```

This is not a candidate for vectorisation if the choices of layout types are limited to the family of k-types. However, if we are going to lay our string as $\langle 0, N-1, 1, N-2, 2, \dots \rangle$ then the string could be reversed by swapping neighbouring elements. Vectorisation will be possible in this case, as the string can be divided into chunks of size V (assuming that V is even) and each chunk can be reversed in place.

First of all let us construct the index mapping. We shall have an index relation function R and define it as:

$$R(x,n) = \begin{cases} x/2 & x = 0 \mod 2 \\ n - x/2 - 1 & x = 1 \mod 2 \end{cases}$$

Now we can construct an index-mapping function for the layout type ξ_r:

$$\xi_r(iv) = \langle R(Viv, N), \dots, R(Viv + V - 1, N) \rangle$$

With those definitions we can address the questions from the previous section:

Composability In this case we can introduce composability by vectorising the inner rows of the multi-dimensional array. Keep in mind that we do not need a family of types in this case.

Dual index type We need to introduce a dual index layout type for the reversed layout.

Values of index types The values of index types are defined by ξ_r so we can allow index types to act as values.

Coexistence The newly introduced type will not be able to share any vector operations on the new layout. Index types inside the map will be either $idx(k)$ or of a new dual type, which means that selections will happen either from reversed array or from an array of layout type k. However, compatibility is still achievable on scalars.

8.1.4 Further type extensions

Layout types can be used in much wider contexts than program vectorisation. For example, [7] demonstrates how layout types can be used to optimise data distributions of arrays when running programs on a cluster. Their approach allows one to "cut" an array in parts and distribute those parts across the cluster via MPI. Such a cutting and distribution is described as one type. If we combine this work with the framework proposed in this thesis, then on the one hand the two kinds of layout types are orthogonal, as distributed types are responsible for cutting array at a high-level and the layout types described in this thesis are used on the parts of arrays on the individual nodes. On the other hand those types form a hierarchy — that is the way individual parts of the array will be vectorised depends on how we cut the overall array. Such a hierarchy opens interesting research opportunities. First of all the size of the hierarchy can be larger than two levels, as certain operations we may want to optimise might require specific data layouts. Secondly, we can shift optimisation efforts from level to level. For example, we can cut an array into less parts to get more advanced vectorisation, or vice versa — we can decide not to vectorise parts of the array, but increase the number of parts. It is likely that such hierarchical types can be formulated using Homotopy Type Theory [140]. The possible levels of layout types can be: SIMD vector units, multi-threaded execution, execution on GPUs, distributed execution on a cluster.

8.2 Using C or Fortran as input languages

Most of the existing applications that could benefit from the proposed technique are written in imperative languages like C/C++, Fortran, etc. Our formalism is presented using a functional language where all the parallel constructs are made explicit. In order to apply our technique to a C/Fortran program, it has to be transformed in a functional form. There are existing approaches that bridge the gap between functional and imperative styles. For example, as noticed by Appel [4], the Static Single Assignment (SSA) form which all the modern compilers use to represent the control-flow of programs is a functional program. In the example of SAC we have seen that some of the C constructions can be transformed into a purely functional form.

Nevertheless, the transition from the C world into a functional world is very challenging for the following reasons. First of all, there are no explicit parallel constructs in C/Fortran which means that they have to be identified. The polyhedral model is at our services, in case parallel constructs are expressed as loop-nests. If a parallel construction is expressed by means of recursive functions or using non-trivial goto operations, another kind of analysis will be necessary to identify it. Luckily this does not happen very often in practice.

Secondly, a number of low-level constructions like longjmp or complex pointer arithmetic, where aliasing cannot be resolved, will make transition undecidable. This means that we will have to consider a subset of possible C/Fortran programs.

Thirdly, in C, there is no arrays may degenerate into pointers which makes it very hard to determine if we are doing an operation on an array or there is some value passed by reference. Also, we always assume that all the references in the array of vector base type are aligned. In C that has to be proven and as arrays are not first class types, that can be undecidable. For example, in:

```
void foo (void *  p)
{
   vector (4, float) (*a)[100] = (vector (4, float) (*)[100])p;
}
```

we might not know where the pointer `p` is coming from. It is likely that for C it will be necessary to add a way to annotate if a certain pointer is an array.

Finally, as C programs can be split in multiple translation units and joined together during linking, the information regarding layout types has to be propagated outside modules. That can be done by extending the language or performing layout transformation during the linking phase when the whole program will be available.

Although the above challenges are non trivial we believe that it would be worth while to tackle this problem.

8.3 Better cost models & advanced vectorisation

The cost model presented in this work is too simple to make correct choices all the time. The main problem is that the cost model does not take into account the sizes of map/reduce operations. It also does not differentiate between the costs of individual vector and scalar operations which can be different on some hardware. Finally, the cost model does not consider memory operations — when we pad we can substantially increase the amount of memory it takes to store a data structure. That in principle can destroy our vectorisation efforts.

We can solve the problem of varying costs of vector operations by reusing the information available to existing auto-vectorisers. For example, in GCC for every supported backend each instruction gets a cost, which makes it possible to approximate the cost of expressions.

If we want to consider the sizes of map/reduce operations, the problem is that they might be not statically known. One possible solution could be to employ symbolic evaluation and to compare the overall costs based on comparison of symbolic expressions. This approach has severe limitations as map/reduce sizes may depend on conditions, recursive functions or external parameters. Another possibility would be to ask a programmer to provide annotations for some of the variables or map/reduce constructs. When a programmer annotates a range we can apply interval analysis techniques. Alternatively annotations could use some different form of algebra that allows one to compare the cost of map/reduce expressions.

Considering memory operations properly is very hard because of the presence of caches. It is possible to approximate or model this behaviour like in [14] to reason about costs. In case the cost of memory operations can be established, we can look into dynamic layout transformations. Currently it is possible to construct an example where different layout requirements for a given array will prohibit vectorisation of a program. In the case of a more advanced cost model we can either decide to satisfy all the operations by changing layouts dynamically before and after the operation, or we can chose the most expensive operation and vectorise only it.

Further we intend to use our framework as a basis for more complicated vectorisation patterns, for example the stencil-like computations. Currently they are not supported, i.e. if we have an expression similar to:

```
map i < N
    a[i-1] + a[i] + a[i+1]
```

our system will not vectorise it. In order to make this possible we can consider techniques described in [55]. To do so we will need to introduce new data layouts and to run an analysis on boundary conditions to regroup computations according to the new data layouts.

Bibliography

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., March 2008. Version 1.0.

[2] Randy Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, October 1987.

[3] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, December 14 1989.

[4] Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, April 1998.

[5] ARM, 110 Fulbourn Road, Cambridge, UK. *RealView Compilation Tools® Version 4.0 Assembler Guide*, June 2010.

[6] N. Arora, A. Shringarpure, and R.W. Vuduc. Direct N-body kernels for multicore platforms. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 379–387, Sept 2009.

[7] Tristan Aubrey-Jones and Bernd Fischer. Synthesizing MPI implementations from functional data-parallel programs, 2014.

[8] Sandro Wenzel Axel Naumann. C++ needs language support for vectorization. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3774.pdf`, 2013. (visited on June 2015).

[9] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The Fortran automatic coding system. In *IRE-AIEE-ACM '57 (Western): Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198, New York, NY, USA, 1957. ACM.

[10] John Backus. Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[11] G.H. Barnes, R.M. Brown, M. Kato, David J. Kuck, D.L. Slotnick, and R.A. Stokes. The ILLIAC IV computer. *Computers, IEEE Transactions on*, C-17(8):746–757, Aug 1968.

[12] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the Intel architecture. *Int. J. Parallel Program.*, 30(2):65–98, April 2002.

[13] Guy E. Blelloch. NESL: A nested data-parallel language. Technical report, Pittsburgh, PA, USA, 1992.

[14] Guy E. Blelloch and Robert Harber. Cache and I/O efficent functional algorithms. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 39–50, New York, NY, USA, 2013. ACM.

[15] Robert L. Bocchino and Vikram S. Adve. Vector LLVA: a virtual vector instruction set for media processing. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 46–56, New York, NY, USA, 2006. ACM.

[16] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[17] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 – Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 488–502. Springer Berlin Heidelberg, 2005.

[18] J. A. Brown. An APL2 description of the IBM 3090 vector facility. In *Proceedings of the International Conference on APL*, APL '88, pages 44–48, New York, NY, USA, 1988. ACM.

[19] M. Burrows and D.J. Wheeler. *A Block-sorting Lossless Data Compression Algorithm*. Number no. 124 in A block-sorting lossless data compression algorithm. Digital, Systems Research Center, 1994.

[20] William W. Carlson, Jesse M. Draper, David Culler, Kathy Yelick, Eugene Brooks, and Karren Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Bowie, MD, May 1999.

[21] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM.

[22] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[23] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[24] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 444–453, New York, NY, USA, 1999. ACM.

[25] Siddhartha Chatterjee, A.R. Lebeck, P.K. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. *Parallel and Distributed Systems, IEEE Transactions on*, 13(11):1105–1123, Nov 2002.

[26] Arun Chauhan and Ken Kennedy. Optimizing strategies for telescoping languages: Procedure strength reduction and procedure vectorization. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 92–101, New York, NY, USA, 2001. ACM.

[27] G. Chen. A constraint network based approach to memory layout optimization. In *In Proc. of the Conference on Design, Automation and Test in Europe*, pages 1156–1161, 2005.

[28] Kuan-Hsu Chen, Bor-Yeh Shen, and Wuu Yang. An automatic superword vectorization in LLVM. In *16th Workshop on Compiler Techniques for High-Performance and Embedded Computing*, pages 19–27, Taipei, 2010.

[29] Michał Cierniak and Wei Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 205–217, New York, NY, USA, 1995. ACM.

[30] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. The reconfigurable streaming vector processor (RSVPTM). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 141–, Washington, DC, USA, 2003. IEEE Computer Society.

[31] Philippe Clauss and Benoît Meister. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *SIGARCH Comput. Archit. News*, 28(1):11–19, March 2000.

[32] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-array Fortran and Unified Parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[33] Paul Cockshott and Greg Michaelson. Orthogonal parallel processing in Vector Pascal. *Computer Languages, Systems & Structures*, 32(1):2 – 41, 2006.

[34] I. Bernard Cohen. Mark I, Harvard. In *Encyclopedia of Computer Science*, pages 1078–1080. John Wiley and Sons Ltd., Chichester, UK.

[35] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 415–426, New York, NY, USA, 2006. ACM.

[36] Joëlle Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986)*, pages 193–205. IEEE Computer Society Press, June 1986.

[37] Jack Dongarra. The LINPACK benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474, London, UK, UK, 1988. Springer-Verlag.

[38] Jack Dongarra, Robert Graybill, William Harrod, Robert F. Lucas, Ewing L. Lusk, Piotr Luszczek, Janice McMahon, Allan Snavely, Jeffrey S. Vetter, Katherine A. Yelick, Sadaf R. Alam, Roy L. Campbell, Laura Carrington, Tzu-Yi Chen, Omid Khalili, Jeremy S. Meredith, and Mustafa M. Tikir. Darpa's HPCS program: History, models, tools, languages. *Advances in Computers*, 72:1–100, 2008.

[39] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, volume 39, pages 82–93, New York, NY, USA, May 2004. ACM Press.

[40] Roger Espasa, Mateo Valero, and James E. Smith. Vector architectures: Past, present and future. In *International Conference on Supercomputing*, pages 425–432, 1998.

[41] Pierre Estérie, Mathias Gaunard, Joel Falcou, Jean-Thierry Lapresté, and Brigitte Rozoy. Boost.SIMD: Generic programming for portable SIMDization. In *Proceedings of the 21st International Conference on Parallel Architectures*

and Compilation Techniques, PACT '12, pages 431–432, New York, NY, USA, 2012. ACM.

[42] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[43] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[44] Free Software Foundation. GCC. `http://gcc.gnu.org`. (visited on June 2015).

[45] HSA Foundation. HSA platform system architecture specification 1.0 provisional. `http://www.hsafoundation.com/?ddownload=4944`, 2104. (visited on June 2015).

[46] F. Franchetti, H. Karner, S. Kral, and C.W. Ueberhuber. Architecture independent short vector FFTs. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, volume 2, pages 1109–1112 vol.2, 2001.

[47] Freescale Semiconductor. *AltiVec Technology Programming Interface Manual*, June 1999.

[48] Wolfgang Gentzsch. Vectorization of computer programs with applications to computational fluid dynamics. *NASA STI/Recon Technical Report A*, 85:15571, 1984.

[49] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[50] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[51] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A binding scope analysis for generic programs on arrays. In *Proceedings of the 17th international conference on Implementation and Application of Functional Languages*, IFL'05, pages 212–230, Berlin, Heidelberg, 2006. Springer-Verlag.

[52] Dan Halacy. *Charles Babbage: father of the computer.* Crowell-Collier, New York, NY, 1970.

[53] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent data representation synthesis. *SIGPLAN Not.*, 47(6):417–428, June 2012.

[54] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[55] 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 international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 225–245, Berlin, Heidelberg, 2011. Springer-Verlag.

[56] High Performance Fortran Forum (HPFF). High performance Fortran language specification, 1997.

[57] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[58] R. G. Hintz and D. P. Tate. Control Data STAR-100 processor design. In IEEE, editor, *Digest of papers, 1972: innovative architecture. Continuing Compcon theme: innovation and change in computer design.*, pages 1–4, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1972. IEEE Computer Society Press.

[59] David Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[60] Intel, Santa Clara, CA, USA. *Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*, September 2013.

[61] Intel, Santa Clara, CA, USA. *User and Reference Guide for the Intel® C++ Compiler 15.0*, September 2014.

[62] Intel Corporation. Intel Cilk Plus homepage. `http://www.cilkplus.org/`. (visited on June 2015).

[63] ISO/IEC. ISO/IEC 9899:1999: Programming languages – C. Technical report, International Organization for Standardization, Geneva, Switzerland., 1999.

[64] ISO/IEC. ISO/IEC 9899:2011: Programming languages – C. Technical report, International Organization for Standardization, Geneva, Switzerland., 2011.

[65] Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons, Inc., New York, NY, USA, 1962.

[66] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[67] G. Kahn. Natural semantics. In FranzJ. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer Berlin Heidelberg, 1987.

[68] D.D. Kalamkar, J.D. Trzasko, S. Sridharan, M. Smelyanskiy, Daehyun Kim, A. Manduca, Yunhong Shu, M.A. Bernstein, B. Kaul, and P. Dubey. High performance non-uniform FFT on modern X86-based multi-core systems. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 449–460, May 2012.

[69] M. Kandemir and I. Kadayif. Compiler-directed selection of dynamic memory layouts. In *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*, pages 219 –224, 2001.

[70] Ralf Karrenberg and Sebastian Hack. Whole Function Vectorization. In *Code Generation and Optimization*, 2011.

[71] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[72] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 20(4):869–916, July 1998.

[73] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2*, November 2011. (visited on June 2015).

[74] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. *SIGPLAN Not.*, 48(6):127–138, June 2013.

[75] Matthias Kretz and Volker Lindenstruth. Vc: A C++ library for explicit vectorization. *Softw. Pract. Exper.*, 42(11):1409–1430, November 2012.

[76] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[77] Chidamber Kulkarni, Francky Catthoor, and Hugo De Man. Advanced data layout optimization for multimedia applications. In *Proc. Workshop on Parallel and Distributed Computing in Image, Video and Multimedia Processing (PDIVM'00), IPDPS'2000*, pages 186–193, 2000.

[78] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth*

International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IV, pages 63–74, New York, NY, USA, 1991. ACM.

[79] Leslie Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, February 1974.

[80] P. J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Commun. ACM*, 8(2):89–101, February 1965.

[81] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part II. *Communications of the Association for Computing Machinery*, 8(3):158–165, March 1965.

[82] Samuel Larsen, Emmett Witchel, and Saman P. Amarasinghe. Increasing and detecting memory address congruence. In *IEEE PACT*, pages 18–29. IEEE Computer Society, 2002.

[83] Ruby Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16:51–59, 1996.

[84] Roland Leißa, Sebastian Hack, and Ingo Wald. Extending a C-like language for portable SIMD programming. In *Principles and Practice of Parallel Programming*, 2012.

[85] Geng Daniel Liu and Wen mei W. Hwu. Dl: A data layout transformation system for heterogeneous computing. In *Proc. IEEE Conf. Innovative Parallel Computing (InPar 12), IEEE*, 2012.

[86] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++ 3.0: accelerating smith-waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14(1):117, 2013.

[87] Qingda Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, and Tin-Fook Ngai. 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, Sept 2009.

[88] 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 International Conference on*, pages 372–382, Oct 2011.

[89] Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In Thomas Ball and RobertB. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414. Springer Berlin Heidelberg, 2006.

[90] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[91] Shintaro Momose, Takashi Hagiwara, Yoko Isobe, and Hiroshi Takahara. The brand-new vector supercomputer, SX-ACE. In JulianMartin Kunkel, Thomas Ludwig, and HansWerner Meuer, editors, *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 199–214. Springer International Publishing, 2014.

[92] Peter Naur et al. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, 1960.

[93] Bradford Nichols, Dick Buttlar, and Jaqueline Proulx Farrell. *Pthreads Programming. A POSIX Standard For Better Multiprocessing.* O'Reilly, Beijing, 1998.

[94] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[95] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[96] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 132–143, New York, NY, USA, 2006. ACM.

[97] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: Revisited for short SIMD architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 2–11, New York, NY, USA, 2008. ACM.

[98] NVIDIA. Parallel thread execution ISA version 4.1. http://docs.nvidia.com/cuda/pdf/ptx_isa_4.1.pdf, 2014. (visited on June 2015).

[99] M. F. P. O'Boyle and P. M. W. Knijnenburg. Non-singular data transformations: Definition, validity and applications. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 309–316, New York, NY, USA, 1997. ACM.

[100] M.F.P. O'Boyle and P.M.W. Knijnenburg. Integrating loop and data transformations for global optimisation. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 12–19, Oct 1998.

[101] OpenACC Rewview Board. OpenACC — Directives for Accelerators. http://www.openacc-standard.org, 2012. (visited on June 2015).

[102] OpenMP Architecture Review Board. OpenMP application program interface. http://www.openmp.org/mp-documents/OpenMP3.1.pdf, 2011. (visited on June 2015).

[103] OpenMP Architecture Review Board. OpenMP application program interface. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013. (visited on June 2015).

[104] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 12 1985.

[105] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986.

[106] Liu Peng, R. Seymour, Ken-ichi Nomura, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, A. Loddoch, M. Netzband, W.R. Volz, and C.C. Wong. High-order stencil computations on multicore clusters. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11, May 2009.

[107] Liu Peng, Richard Seymour, Ken ichi Nomura, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, Alexander Loddoch, Michael Netzband, William R. Volz, and Chap C. Wong. High-order stencil computations on multicore clusters. In *IPDPS*, pages 1–11. IEEE, 2009.

[108] Simon J. Pennycook, Chris J. Hughes, M. Smelyanskiy, and S. A. Jarvis. Exploring SIMD for molecular dynamics, using Intel® Xeon® processors and Intel® Xeon phi® coprocessors. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 1085–1097, Washington, DC, USA, 2013. IEEE Computer Society.

[109] M. Pharr and W.R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13, May 2012.

[110] Michael Philippsen. Automatic alignment of array data and processes to reduce communication time on DMPPs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 156–165, New York, NY, USA, 1995. ACM.

[111] Fred J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address). In *Proc. 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, Washington, DC, USA, 1999. IEEE Computer Society.

[112] Gang Ren, Peng Wu, and David Padua. A preliminary study on the vectorization of multimedia applications for multimedia extensions. In *In 16th International Workshop of Languages and Compilers for Parallel Computing*, pages 420–435, 2003.

[113] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 38–49, New York, NY, USA, 1998. ACM.

[114] R. Ronen, A. Mendelson, K. Lai, Shih-Lien Lu, F. Pollack, and J.P. Shen. Coming challenges in microarchitecture and architecture. *Proceedings of the IEEE*, 89(3):325–340, mar 2001.

[115] Paul Rubinfeld, Bob Rose, and Michael McCallig. Motion video instruction extensions for Alpha. Technical report, Semiconductor Engineering Group, 77 Reed Road, HLO2-3/D11 Hudson, MA 01749, USA, October 1996.

[116] Richard M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, January 1978.

[117] Jean E. Sammet. The early history of COBOL. *SIGPLAN Not.*, 13(8):121–161, August 1978.

[118] 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.

[119] Sven-Bodo Scholz. Partial evaluation as universal compiler tool: Experiences from the SAC eco system. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 95–96, New York, NY, USA, 2014. ACM.

[120] J. Seward. Bzip2. http://www.bzip.org. (visited on June 2015).

[121] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures, 2002.

[122] Jaewook Shin, Mary Hall, and Jacqueline Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 165–175, Washington, DC, USA, 2005. IEEE Computer Society.

[123] Artjoms Šinkarovs and Sven-Bodo Scholz. Data layout inference for code vectorisation. In *The 24th symposium on Implementation and Application of Functional Languages*, IFL 2012, 2012.

[124] Artjoms Šinkarovs and Sven-Bodo Scholz. Portable support for explicit vectorisation in C. In *16th Workshop on Compilers for Parallel Computing (CPC'12)*, 2012.

[125] Artjoms Šinkarovs and Sven-Bodo Scholz. Data layout inference for code vectorisation. In *International Conference on High Performance Computing & Simulation, HPCS 2013, Helsinki, Finland, July 1-5, 2013*, pages 527–534, 2013. http://ashinkarov.github.io/publications/data-layouts.pdf.

[126] Artjoms Šinkarovs and Sven-Bodo Scholz. Functionally redundant declarations for improved performance portability. In *The 25th symposium on Implementation and Application of Functional Languages*, IFL 2013, 2013.

[127] Artjoms Šinkarovs and Sven-Bodo Scholz. Semantics-preserving data layout transformations for improved vectorisation. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 59–70, New York, NY, USA, 2013. ACM.

[128] Artjoms Šinkarovs and Sven-Bodo Scholz. Type-driven data layouts for improved vectorisation. *Concurrency and Computation: Practice and Experience*, 2015. http://dx.doi.org/10.1002/cpe.3501 (visited on June 2015).

[129] Artjoms Šinkarovs, Sven-Bodo Scholz, Robert Bernecky, Roeland Douma, and Clemens Grelck. SAC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. *Concurrency and Computation: Practice and Experience*, 26(4):952–971, 2014. http://ashinkarov.github.io/publications/sexynbody.pdf.

[130] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. The SOLOMON computer. In *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference*, AFIPS '62 (Fall), pages 97–107, New York, NY, USA, 1962. ACM.

[131] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28:363–400, 2000.

[132] Sun Microsystems. Accelerating core networking functions using the UltraSPARC VIS instruction set. Technical Report WPR-0013, Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303-4900, USA, August 1996. Demonstrates speedups of 2x to 5x on key networking kernels from use of the VIS instruction set.

[133] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 513–522, New York, NY, USA, 2010. ACM.

[134] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 513–522, New York, NY, USA, 2010. ACM.

[135] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[136] Sudarsan Tandri and T.S. Abdelrahman. Automatic partitioning of data and computations on scalable shared memory multiprocessors. In *Parallel Processing, 1997., Proceedings of the 1997 International Conference on*, pages 64–73, Aug 1997.

[137] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[138] TOP500 project. Top 10 sites for november 2014. http://www.top500.org/lists/2014/11, 2014. (visited on June 2015).

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

[140] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013. (visited on June 2015).

[141] John von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, June 1945.

[142] Haichuan Wang, Peng Wu, Ilie Gabriel Tanase, Mauricio J. Serrano, and José E. Moreira. Simple, portable and fast SIMD intrinsic programming: Generic SIMD library. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 9–16, New York, NY, USA, 2014. ACM.

[143] W. J. Watson. The TI ASC: A highly modular and flexible super computer architecture. In *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*, AFIPS '72 (Fall, part I), pages 221–228, New York, NY, USA, 1972. ACM.

[144] Michele Weiland. *Chapel , Fortress and X10 : novel languages for HPC*. UoE HPCx Ltd., 2007.

[145] David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for Morton-order matrices. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 24–33, New York, NY, USA, 2001. ACM.

[146] Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi GF100 GPU architecture. *IEEE Micro*, 31(2):50–59, 2011.

[147] L. Wittgenstein. *Philosophical Investigations*. Basil Blackwell, Oxford, 1953.

[148] J.A. Wittreich. *Feminist Milton*. Cornell University Press, 1987.

[149] Zhang Xianyi, Wang Qian, and Werner Saar. OpenBLAS. `http://www.openblas.net/`. (visited on June 2015).

[150] Yuanrui Zhang, Wei Ding, Mahmut Kandemir, Jun Liu, and Ohyoung Jang. A data layout optimization framework for NUCA-based multicores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 489–500, New York, NY, USA, 2011. ACM.

[151] Yuanrui Zhang, Wei Ding, Jun Liu, and M. Kandemir. Optimizing data layouts for parallel computation on multicores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 143–154, Oct 2011.

[152] Victoria Zhislina. From ARM NEON to Intel MMX&SSE — the automatic porting solution, tips and tricks. `http://goo.gl/G2Beuy`, 2012. (visited on June 2015).