BACHELOR THESIS

Analysis and Extension of the Intel® SPMD Program Compiler

by Daniel Schürmann

TU Berlin, Berlin, Germany
daniel.schuermann@campus.tu-berlin.de

Faculty IV – Electrical Engineering and Computer Science
Supervisor: Prof. Dr. Ben Juurlink
01.07.2016

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 01.07.2016

Unterschrift

Abstract

The Intel SPMD Program Compiler (ispc) is a new research programming language to easily write vectorized code while maintaining high performance. However, due to a different programming model used in this language and missing language features, the obtained peformance can fall short of expectations. In this thesis, we evaluate ispc to find out it's potentials and limitations.

The language's type system and control flow constructs are analyzed and a compiler extension was written to output additional information from the compiler's front-end. With regard to performance, two applications are optimized and benchmarked. Various issues of ispc, like the missing support for templates and missing standard optimizations, are identified and workarounds discussed.

However, a multimedia application could not reach the performance of the serial implementation. On the other hand, an example with complex control flow achieved a speed-up of 1.74, showing a use-case of high performance computing applications.

Zusammenfassung *(german)*

Der Intel SPMD Program Compiler (ispc) ist eine junge Forschungssprache, die eine einfache Programmierung von vektorisiertem Code ermöglichen und dabei gleichzeitig eine hohe Performance erreichen soll. Ein abweichendes Programmiermodell sowie fehlende Sprachfeatures führen aber oft dazu, dass die erreichte Performance hinter den Erwartungen zurückbleibt. In dieser Arbeit wird ispc ausgewertet, um die möglichen Anwendungszwecke, aber auch Probleme und Einschränkungen zu ermitteln.

Dafür werden das Typsystem sowie die Sprachkonstrukte analysiert. Eine Compiler-Erweiterung wurde geschrieben, um zusätzliche Informationen aus dem Front-End des Compilers auszugeben. Zwei Beispielanwendungen werden in Hinblick auf die mögliche Performance optimiert. Verschiedene Probleme wie die fehlende Unterstützung von Sprachfeatures, zum Beispiel Templates, und fehlende Standardoptimierungen werden dargestellt und mögliche Umgehungslösungen diskutiert.

Eine Multimediaanwendung kann dennoch nicht die Geschwindigkeit der seriellen Implementierung erreichen. Ein Beispiel mit komplexem Kontrollfluss hingegen erreicht einen Geschwindigkeitszuwachs von 1,74 und zeigt damit eine Anwendungsmöglichkeit im Bereich von High-Performance Computing.

BACHELOR THESIS

Analysis and Extension of the Intel® SPMD Program Compiler

Daniel Schürmann

Table of Contents

1. Introduction

With the advent of Single Instruction Multiple Data (SIMD) units on desktop computers and the increasing number of multimedia applications, it became a demand to make use of the processors' capabilities by vectorizing code. Code vectorization can give high performance benefits, but provides challenges that are the focus of actual research.

By Intel, a new programming language, the Intel SPMD Program Compiler (ispc) was developed. The main purpose is to easily write vectorized code and to achieve high performance. For this matter, a different programming model, called Single Program Multiple Data, is used. However, the inconvenience of writing code with a different programming model can lead to results that perform below the expectations. ispc is a young research language and therefore not as optimized as e.g. modern C compilers. Although it is based on a theoretically founded model, the outcome depends on certain restrictions which are not always obvious to the user. The goal of this thesis is to find out the potentials and limitations of ispc. This includes the question which restrictions apply, how they can be circumvented and how a better understanding of the compiler can help to optimize code. For this purpose, an analysis about the capabilities and functionality of ispc is given as guidance in this thesis.

In the second section, first an overview over the functionality and history of SIMD units is given, and below the challenges of code vectorization and different programming approaches are presented. At the beginning of section three, ispc's programming model is described and a language overview is given. The middle part contains a charactization of the type system including potential type safety issues, and in the last part of section three, the control flow mechanisms are outlined.

Additionally, a new compiler mode, ispc-explained, was written for this thesis to give a better understanding of the semantics and underlying programming model of the language. This compiler mode outputs different kinds of information from the compiler's front-end and is introduced in section four. In addition, potential use-cases of ispc-explained are presented.

Finally, the last section contains two examples to illustrate the potentials and limitations of ispc. The first example refers to the Collatz Conjecture and is not auto-vectorizable. Different optimizations and their impact on the performance are shown. The second example is a kernel from a video decoder to be exemplary for modern multimedia applications.

2. SIMD extensions and their use

In 1966, Michael J. Flynn proposed a classification of microarchitectures. This classification, defined in 1966, includes single instruction streams that work either on single data streams (SISD) or multiple data streams (SIMD) [13]. In modern computer architectures, both can be found in single core processors. While the SISD part can work on scalar code being excuted in a sequential way, the SIMD units can handle mutiple data with one instruction.

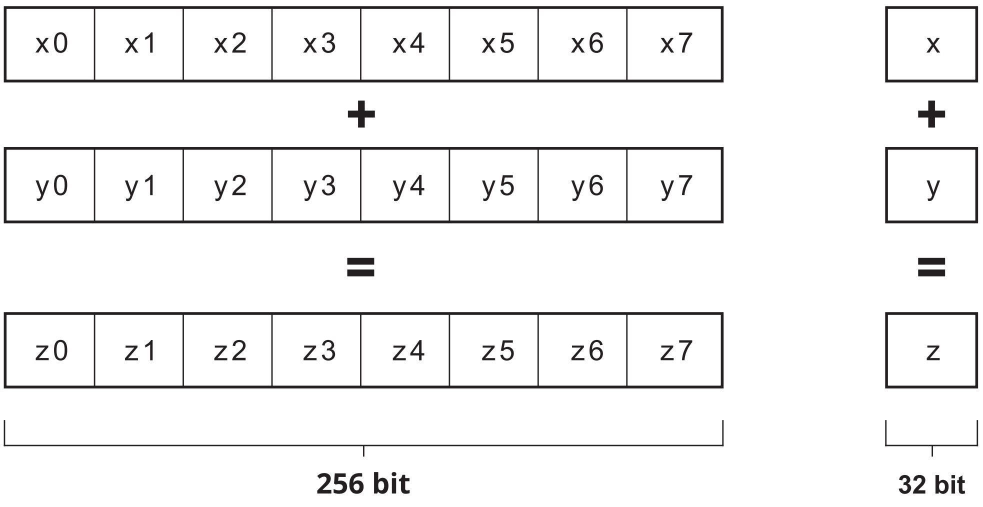


Figure 1: *SIMD Mode and Scalar Mode: The execution of an add instruction is shown on a processor's SIMD unit in comparison to a Scalar unit.*

Technically, the SIMD instructions operate on large registers which contain multiple values of the same type. For example, 8 integers of 32 bit fit in a register with the size of 256 bit. In Figure 1, an operation on two such registers is shown where the addition gets performed on 8 SIMD lanes.

The SIMD units, also called vector units, became necessary for desktop computers in the 1990s because of the increasing number of multimedia applications which can capitalize on the ability to perform an operation on numerical vectors rather than on scalar values. The use of SIMD units can provide high theoretical performance benfits, if the code is vectorized.

This section covers a brief overview over the history of SIMD extensions on desktop computers and common programming approaches.

2.1 The history of SIMD extensions on desktop computers

Before 1996, SIMD computations were widely used in supercomputers only. Multi Media Extension (MMX) was the first SIMD extension for Intel desktop computers to match the demand of parallelizable multimedia applications for vector processing. However, it was limited to integer data. In addition, it was not possible to execute scalar floating point instructions at the same time, as the MMX registers were mapped to the existing floating point registers [26]. These 64 bit registers could be partitioned into 2 x 32 bit, 4 x 16 bit, or 8 x 8 bit integers.

In 1998, AMD's 3DNow! extension added floating point capabilites, but still used the same registers. Because Intel refused to implement 3DNow! on their CPUs, this extension was unsuccessful and discontinued after 2011 [2].

Intel's answer to 3DNow! was the introduction of Streaming SIMD Extensions (SSE) in 1999, which also added floating point instructions and operate on new specific 128 bit registers, known as XMM registers. Distinct registers allowed for the processor to execute SSE and MMX instructions concurrently.

The expansion of SSE2 in 2001 made MMX obsolete by adding integer instructions to the SSE instruction set, which operate on the XMM registers [35].The iterations SSE3, SSSE3 and SSE4 introduced from 2003 to 2006 brought new instructions to improve thread synchronization and specific application areas such as media and gaming.

While all SSE instructions have a two-operand form, the new Advanced Vector Extensions (AVX), first supported by Intel in 2011, use a three-operand instruction format. AVX can be seen as a replacement for the SSE family, extending the SIMD registers to 256 bit, thus doubling the theoretical performance increase.

The newest expansion of the AVX instruction set, AVX2, enables vector elements to be loaded from non-contiguous memory locations using gather instructions. Intel plans to extend the register size in 2016 to 512 bit with AVX512. AVX512 will introduce a four-operand instruction format and 7 opmask registers(1).
This development requires current microprocessors, like Intel Haswell, to support 11 different iterations of SIMD extensions.

2.1.1 SIMD Extensions on other microprocessor architectures

Meanwhile, other processor manufacturers developed their own solutions, too, that are still in use today. For ARM the Advanced SIMD extension, i.e. NEON, provides a combined 64 and 128 bit SIMD instruction set, while for PowerPC the 128 bit SIMD instruction set is called AltiVec.

(1) These registers can hold boolean values to indicate, which SIMD lanes perform a predicated instruction and thus offer hardware support for conditionals.

2.2 SIMD Programming Approaches

The inherent differences between SIMD and scalar units provide a number of challenges for parallelizing across SIMD vector lanes. As most programming languages are developed with a SISD microarchitecture in mind, they do not provide any features to target SIMD parallelism. As a consequence, programming SIMD units is considered difficult and expensive in terms of development costs.

The general challenges to program SIMD units not only include the vectorization of the code. To prevent the processor from performing slow gather operations and instead use vector loads, the programmer also has to provide for a contiguous memory layout. As the same operation is performed on different data at the same time in a so-called *lockstep*, control flow for different SIMD lanes is limited.

Several approaches based on high-level programming languages aim to assist the programmer in writing vectorized code. The ways presented in the following passages range from intrinsic functions which offer hardly any abstraction, to auto-vectorization for the least effort, and also include some newer programming models for code vectorization.

2.2.1 Intrinsic Functions

Intrinsic functions are functions specifically handled by the compiler. Every intrinsic function represents a SIMD instruction of the microprocessor, and thus can be directly mapped to it. However, this is not always the case as compilers are free to apply optimizations. For example, the compiler might change instructions with other ones that perform better or eliminates some that are not necessary [25].

The Intel Intrinsics Guide [17] lists more than a thousand different functions that can be used within C++ compilers, such as the Intel C++ Compiler (ICC) [16], the Gnu Compiler Collection (GCC) [14], or Clang [6].

The absence of high level abstractions and the large number of different intrinsic functions require some understanding of the SIMD microarchitecture and its capabilities. Intrinsic functions only support primitive numeric data types, such as integer and floating point numbers, which means that any structured data like classes, arrays or tuples have to be rearranged before use. The memory layout has to be organized in a contiguous way to allow for vectorized memory accesses. Loop vectorization can be done by manual loop unrolling, until the vector width and control flow requires a complex implementation using masks.

Programs that use intrinsics are not portable to different microarchitectures due to variations in supported features, vector length and arithmetics. To ensure backwards compatibility and also to make use of modern features, the program has to be written for different instruction set architectures (ISA). Runtime checks have to be implemented to ensure that the right version of the program is loaded. This results in a significantly increased code size and poor readability.

From a programmer's perspective, SIMD programming with intrinsics is very labour intensive. If done properly, the benefit is a high speed-up through extensive usage of the processor's SIMD units and the precise control of the low level details. However, due to the many disadvantages, intrinsics are typically only used for performance critical programs.

2.2.2 Auto-vectorization

Auto-vectorization is a feature of most modern compilers that automatically transforms a scalar implementation into a vector implementation, so that the programmer doesn't have to care about the underlying architecture and its capabilities.

The two main approaches to automatically vectorize code are Superword Level Parallelism (SLP) and Loop Level Parallelism (LLP). SLP exploits Instruction Level Parallelism (ILP) and is performed on independent isomorphic statements, which are statements that contain the same operations in the same order in an arbitrary basic block. These statements then are packed together and parallelized. This works best on adjacent memory references [22].

As for-loops typically execute the same instructions on different data multiple times, LLP aims to vectorize these loops. This is done in a similar way a programmer would do manually using intrinsics by unrolling a loop to a specific vector width. Then the instructions of the loop body are vectorized and simple control flow is implemented by using masks. Therefore this optimization can obtain high speed-ups, close to the theoretical maximum. However, auto-vectorization has different restrictions, which prevent this optimization from being applied.

Loop vectorization is strictly restricted to countable for-loops, i.e. loops where the number of loop iterations is known before the loop is entered at run-time. This implies that there can be only a single entry and a single exit from the loop, thus there is no break and no goto allowed within the loop body. Like SLP, non-contiguous memory accesses can deny any performance benefit for loop vectorization. Some sophisticated compilers choose to not vectorize loops in this case.

Since the result of a program has to be the same, whether executed in scalar or parallel, loops with data dependencies or side-effects cannot be vectorized safely. For example, as it is possible to change pointers in C, there are cases where the compiler cannot tell by a static dependency analysis if there are data dependencies, and then conservatively assumes overlapping memory regions. Side-effects, e.g. a print statement or access to a global variable in the loop body, may lead to a result depending on the execution order and therefore prohibit loop vectorization.

Finally, complex control flow, such as `switch` statements or nested function calls, and all function calls that cannot get inlined are disallowed in loops that should be vectorized. This results in lack of transparency in terms of performance, meaning that it is difficult for users to estimate the performance gains from auto-vectorization. However, the convenience of writing code in a well-known language and the possibility of easily increasing performance makes this way of code vectorization the de facto standard of non performance-critical program parts [9].

2.2.3 Alternative Approaches

Several approaches for vector programming with a higher level of abstraction are highlighted in this section. Due to the lack of portability and abstraction of intrinsics programming, several libraries have been developed to meet the demand for more convenient programming models that are capable to achieve a performance similar as platform-specific intrinsics versions.

Boost.SIMD [11] is a C++ Template library that aims to simplify portable SIMD code. This approach is based on a pack class which encapsulates a specific number of elements. If type and number of elements match with SIMD register sizes, they get mapped onto it. The library contains several functions to work with the pack class. Generic SIMD Library [37] is a similar approach to Boost.SIMD. Despite the library's claims to be simpler and smaller than Boost.SIMD, the key difference is that the user does not work with a specific vector width, but with short vectors of a specified length, which get mapped onto the SIMD lanes.

Vc [21] provides vector types for primitive number types, as well as a set of functions and masking operations for vectorization. The vectors of Vc do not have a user-specified length, but are sized by the library according to the target's hardware. This allows for a higher level of abstraction but also reduces the user control. The same applies to Cyme [12], which in addition focuses on the abstraction of memory layout to match the requirement of SIMD units for coherent memory accesses.

Apart from these libraries that aim to ease SIMD programming with a higher level of abstraction, there are also compiler based approaches. OpenMP [30] is an application programming interface (API) for developing parallel applications. It is implemented in various compilers, such as GCC and ICC, and works with a set of preprocessor directives, so-called pragmas, functions and evironment variables. With Version 4.0, support for SIMD parallelism has been added.

Cilk Plus [33] is a language extension for C/C++, which adds few keywords and pragmas for multithreading and vectorization. Implementations for different compilers have been distributed by Intel. IVL [23] is a C-like research language developed at Saarland University. The project seemed to be abandoned, when a new programming language, Sierra, came out. Sierra [24] is a C++ language extension for SIMD programming. It adds a constructor for vector types with explicit vector length and provides several features for control flow and memory layout.

Although OpenCL [27] is primarily used for GPU high performance programming, there exist also implementations for CPUs that make use of the processors SIMD lanes. Another programming language for code vectorization with a programming model similar to OpenCL is the Intel SPMD program compiler.

3. The Intel SPMD Program Compiler

The Intel SPMD Program Compiler (ispc) is a new compiler-based approach developed by Matt Pharr and William R. Mark for vector programming [31]. The aim of ispc is to achive high-performance, close to intrinsic functions, maintaining the productivity of programming scalar C code. ispc targets transparency in terms of performance, which means that the user should be able to estimate how the written code will perform. Furthermore, ispc wants to be easily adoptable by programmers by providing familiar syntax and interoperability with existing C code.

ispc comes with its own compiler and supports the export of functions that can be linked with standard C/C++ code after compilation. Therefore it is possible to use ispc only for a few performance critical functions and leave the rest of the sources untouched. ispc is based on C89 with some C99 and C++ features like `bool` type, references and function overloading. To allocate memory dynamically, ispc provides C++'s `new` and `delete` operators.

In this section, the programming model of ispc is described, followed by a language overview. Subsequently a detailed analysis of ispc's type system and control flow mechanisms is given. Finally the compiler infrastructure will be outlined. The information about the type system and control flow mechanisms is mainly based on the ispc User's Guide [19] and a technical paper [31].

3.1 The SPMD Programming Model

In addition to SIMD and SISD microarchitectures, Flynn's taxonomy includes MIMD microarchitectures. This category describes multicore architectures that handle multiple instruction streams with multiple data streams. In 1988, F. Darema enhanced the classification with a programming model, called SPMD, which is an acronym for Single Program Multiple Data [8]. SPMD is a subcategory of MIMD in a way that multiple processors execute the same program on different data.

In the SPMD programming model, tasks are split up on multiple program instances that run concurrently with different input data. Each program instance executes the same program, which therefore can be written in a scalar manner. The convenience of writing scalar code made SPMD the dominant technique for parallel programming [34].

Having its origin in the MIMD microarchitectures, the SPMD programming model implies independent program instances, which each can follow their own control paths. Implementations typically use threads that run on multiple processors. Therefore the program instances can be at independent points of the program and are not in lockstep. This enables complex control flow, function calls etc. The data sets each instance is working on have to be independent and communication between these instances has to be explicit. The SPMD programming model is widely used for distributed computers and GPUs. Common programming languages that use the SPMD programming model are CUDA [28] and OpenCL.

ispc uses a model which is referred to as "SPMD-on-SIMD", where program instances are mapped to the processor's SIMD lanes. The program instances running concurrently are called a gang. The size of the gang depends on the target's instruction set and is determined at compile time. It is typically a power of two in the range from four to sixteen instances. This means that the user has only limited control over the used vector width and the vector width also cannot change during the program. ispc aims to implement the SPMD programming model with all features regarding complex control flow and function calls.

3.2 Language Overview

With the call to an ispc function, all program instances, i.e. the gang, start running concurrently on the SIMD lanes with every program instance having its own set of variables. All data within ispc either exists in a distinct version for each program instance and is then called `varying` data, or exists only once in memory and is the same for each program instance, the so called `uniform` data. Uniform data corresponds to the supported data types from C/C++. Therefore, to allow for interoperability, all exported functions and extern declared functions are required to only have uniform parameters.

Code parts that work on uniform data are handled sequentially by the processor's scalar units, while the program instances, and thus the SIMD units, process all varying data.
A built-in variable `programCount` is accessible to return the total number of instances running, while `programIndex` is specific to every program instance and ranges from 0 to `programCount -1`. These variables can be used to define a parallel for-loop, like shown in the example below.

Program instances can be active or inactive. A boolean mask is maintained to indicate which program instances are active during a statement. Unlike auto-vectorization, this allows for complex control flow within the loop body, such as `switch` or multiple `return` statements with different program instances following different control paths.

```
export void hailstone(uniform int values[], uniform int length) {
    for(varying int i = programIndex; i<length; i+=programCount) {
            int iters = 0;
            int value = i + 1;
            while(value != 1) {
                    iters++;
                    if((value % 2) == 0) //even
                            value /= 2;
                    else //odd
                            value = value * 3 + 1;
            }
            values[i] = iters;
    }
}
```

Figure 2: *An ispc implementation to solve the Collatz Conjecture for different numbers in parallel.*

This small example shows an ispc function that takes an array and its length as input and calculates the number of iterations to complete the hailstone sequence [15] for the first `n` numbers. The hailstone sequence refers to the so-called *3x+1* problem, also known as Collatz Conjecture [4]. The `export` qualifier indicates that the function is callable from C/C++. The number of iterations and starting values are initialized as varying integers. The while-loop is entered by all program instances concurrently. If the condition becomes `false` for one program instance, it gets inactive until all program instances are finished. Then the values are written back, the index `i` gets incremented and the loop is again entered by all program instances. Thus, it is possible that the program instances execute the while-loop a different number of times and follow diverging control paths.

3.3 Type System

As ispc is based on C89 [3], the type system is also comparable to C's type system, adding some extensions. All pointer, array, enumeration and struct types from C are also available in ispc. Only the `union` type has not been implemented. On the other hand, ispc adds data types that are specifically useful for vector programming, for example structures of arrays, and defines the primitive data types more precisely.

3.3.1 Primitive Data Types

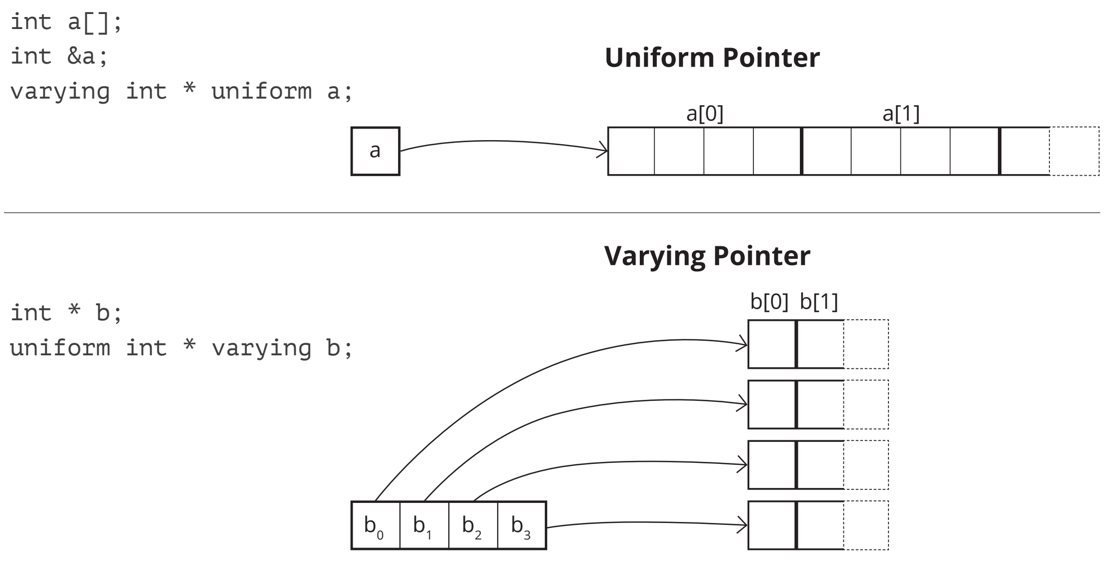
In C, the bit lengths of the primitive types are platform dependent and therefore the calculation results may vary on different platforms. In ispc the primitive types have a specified bit length, which is 32 bits for `int` and `float`. Other primitive types are `int8`, `int16`, `int64`, and `double` with 64 bits as well as the `void` type. All integer types can also be `unsigned` like in C. From C99, ispc adds the `bool` type.

Explicit type casts between these primitive types are possible in ispc, and implicit type conversions are automatically inserted by the compiler, if mixed types appear in arithmetic expressions or assignments. There is no support for a character type and outside of a rudimentary print function, the language has no support for strings.

3.3.2 Uniform and Varying Types

The biggest extension to the type system is the `uniform` and `varying` types mentioned above. In ispc, all primitive data types are available in either the uniform or varying form. If a variable has a uniform type, only one value can be assigned to it. The value is the same for all program instances, as the variable exists only once in memory. The uniform types correspond to the normal C types and must be used for interaction with C/C++ programs. On the other hand, if a variable is of varying type, it contains one value for each program instance stored as a vector. As the number of program instances depends on the target platform, the size of varying type variables is also platform dependent and assumptions about the vector width may lead to non portable code.

If no explicit `uniform` qualifier is specified for a variable, the variable is of `varying` type by default. Uniform values can be assigned to varying variables and get implicitly converted to varying types. However, the conversion from varying types to uniform is not possible. From a theoretical perspective, the implication is that uniform types are subtypes of their varying counterparts.



```
int a[];
int &a;
varying int * uniform a;
```

```
int * b;
uniform int * varying b;
```

Figure 3: *The memory layout and notation of uniform and varying pointer*

3.3.3 Pointer Types

Like the C programming language, ispc supports arbitrary pointers to any data types. These pointers can themselves be of uniform or varying type, meaning that there either exists only one single scalar pointer or one for each program instance. The same applies to the data being pointed to, which can also be either uniform or varying. ispc supports the whole expressibility of pointer arithmetics, pointers to pointers, function pointers and null pointers. Therefore ispc features the implementation of complex data structures and interoperation with existing application data structures. However, if not using the pointers appropriately, ispc has the same problems regarding memory corruption and undefined behavior like C programs. As pointer aliasing is unrestricted in ispc, data races between program instances are possible.

The possibility of function pointers to be of varying type implies that each program instance can call a different function concurrently. ispc provides reference types, which work in the same way as in C++. References are always of uniform type, though they may refer to varying data.

3.3.4 Arrays and Structs

Arrays are built-in types in ispc and can, as in C, be declared with the array length being part of the type. Array types without specified length only occur as function parameters. Arrays can be multidimensional and are applicable to either uniform or varying data.

Internally, arrays are represented as uniform pointers to their specified data and can therefore be used as uniform pointers. Array accesses outside the specified length result in undefined behavior and may crash the program.

The declaration of `struct` data types is to some degree different from other types. Syntactically, the declaration and use of structs is similar to C/C++, with the exception that no bitfields are supported. However, the variability of structs only refers to the variability of its members. Struct members can be specified as uniform or varying type or otherwise are unspecified. The variability of these struct members then gets resolved when a struct instance gets defined. Operators can be overloaded by the definition of a function using the keyword `overload`.

However, unlike in C++, this feature is restricted to binary arithmetic operators only. With the support of arrays and structs, ispc allows to implement and use all common data structures and to interoperate with C/C++ data structures.

3.3.5 Structures of Arrays

A common and comfortable way to represent data is the representation as array of structures (AOS). This leads to the problem of slow performance of vectorizations, as the struct members get physically scattered in memory, and therefore have to be accessed by slow gather instructions instead of faster vector loads. A more efficient memory layout that allows for consecutive memory accesses would be the use of a structure of arrays (SOA). However, the notation of SOAs is inconvenient and uncommon. Another disadvantage of SOAs is that more pointers have to be maintained by the program while iterating over these structures. Because more pointer arithmetics have to be done, this causes a higher register pressure and thus a small performance loss. To overcome this disadvantage, hybrid SOAs can be used. Hybrid SOAs are AOSs, where the members of the original struct get widened to short arrays with the length of the vector. Although being the most efficient data structure for SIMD computations, they are more difficult to implement.

```
// array of structures              // hybrid structure of arrays
struct Pixel { int8 x, y, z; };     struct Pixel4 { int8 x[4], y[4], z[4]; };
uniform Pixel A[16] = { ... };      uniform Pixel4 A[4] = { ... };
```

ispc offers syntactic sugar to make the definition and use of hybrid SOAs as easy as of AOSs. They are even syntactically equal, the only difference being the keyword `soa`. With the keyword `soa` and the desired vector length as applied parameter, ispc transforms an AOS recursively into a hybrid SOA.

```
// ispc hybrid SOA notation
struct Pixel { int8 x, y, z; };
soa<4> struct Pixel A[16] = { ... };
```

For optimal performance, the vector length should match with the number of program instances. As the gang size is platform dependent, this leads to the code working best only on a specific platform. A soa transformation, which always matches the target's gang size should be considered for the further development.

3.3.6 Type Definitions, Enumerations and Short Vector Types

The definition of new names for types is supported in ispc in the same way as in C using the `typedef` keyword. This means that no new type will be defined, but a different, e.g. simpler name can be used for a complex type.

Enumerations are part of ispc's type system, introducing a new type for each defined enumeration. Internally, enumerations are represented as integer values, which can as well be user defined, and thus conversion between them and any number types is possible. Also, all arithmetic operations and the usage as integer parameter are allowed, like in C. Many multimedia computations use short vectors to represent data like pixel colors or vertices. For that purpose, in C, a small struct or an array needs to be used, while in ispc primitive data types can be enhanced to be a vector of arbitrary length.

```
uniform int<3> foo = { 1, 2, 3 }   // vector of three integers
```

This includes all arithmetic and logical operations and the capability of being either uniform or varying. These short vectors are not used for the vectorization of the code, but only as a shortcut for a small struct or array. Vector members can be accessed via array indexing operator or structure member access operator extended for swizzling.

3.3.7 Type Safety

The main purpose of a programming language's type system is to prevent the occurrence of execution errors during the running of a program [5]. A language is called *type-safe* if a well-typed program cannot run into any forbidden errors. These are in particular untrapped errors, respectively undefined behavior.

As not all possible errors can be detected statically by the compiler, the safety of a programming language costs either performance due to necessary runtime-checks or expressiveness of the language, which can also cost performance. ispc has the same priority on high performance as C, and with the type system being based on C89, ispc inherits many unsafe features from C, resulting in possible undefined behavior. It can cause buffer overflows due to unchecked array bounds. This problem cannot be addressed at compile time, and runtime checks would lead to a significant loss in performance.

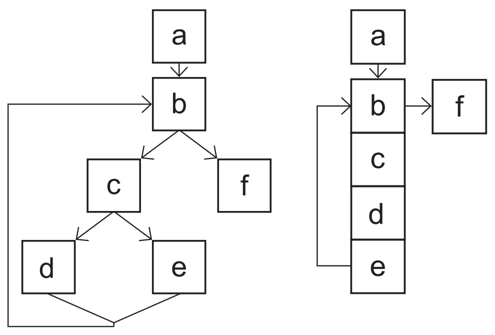
In ispc, implicit type casts are inserted by the compiler. While most of them only provide an upcast from a subtype to its supertype, and therefore are uncritical for safety reasons, pointers get implicitly cast to and from the void type. This means that any pointer type can implicitly be converted to any other pointer type without explicit notation of the programmer. More precisely, any pointer type can be explicitely cast to any other pointer type. As a result, dereferencing can cause memory corruption and undefined behavior. The same applies to pointer arithmetic.

These unsafe features are directly inherited from C and are carried forward to the new introduced SOA and short vector types. Because of the subtype relation between uniform and varying types and the strict policy of only allowing uniform types to be cast to varying types and not the other way around, the distinction of uniform and varying types does not lead to any new potential errors.

3.4 Control Flow

Control flow depends on conditions, which in ispc have the same distinction of being either uniform or varying. While the evaluation of uniform conditions leads to all program instances following the same control path, varying conditions can have different results for each program instance.

Due to the SIMD property of having only one instruction pointer and therefore program instances being in lockstep, they cannot follow different control paths in terms of executing different code. To nonetheless provide independent control flow functionality for the program instances, ispc transforms the control dependence to data dependence. The approach described for If-Conversions by Allen et al.[1] and generalized by Karrenberg et al. [20] is used to achieve this. The general idea is that a boolean mask indicates which of the statements have to be executed by which program instances. The mask is created or updated by conditional statements, as well as `break`, `continue` or multiple `return` statements. To enable function calls within diverging control paths, ispc functions get a mask as an implicit parameter. As all program instances always execute the same instruction at the same time, a mask cannot prevent instructions from being executed, but side-effects from being visible. This is primarily done by hiding out write- backs and will result in a decreased SIMD efficiency, depending on the number of program instances following diverging control paths. Another implication is, that if one program instance is locked in an infinite loop, this propagates to all program instances resulting in the program not to terminate.



Program Instance	Trace
1	a b c d e b c d e b f
2	a b c d e b c d e b f
3	a b c d e b c d e b f
4	a b c d e b c d e b f

Figure 4: *Diverging Control Flow: Whole-Function Vectorization after Karrenberg et al. shows Control-Flow to Data-Flow Conversion. The (converted) CFG represents the for-loop body of the hailstone example and the traces represent the execution with the numbers 1 to 4.*

Because only newer SIMD extensions like AVX have hardware support for mask predicated instructions, for older platforms like SSE the mask has to be implemented and handled by the compiler. The new AVX-512 SIMD extension added specific opmask registers as a result of the demand.

3.4.1 Gang Convergence Guarantees

Like any other general purpose programming language, ispc offers various constructs for control flow within the program. As the underlying programming model of ispc is SPMD, the control flow can cause a different control path being taken by each program instance. Because the program instances are mapped to the hardware's SIMD lanes and there is only one instruction pointer, program instances following the same control path perform every instruction concurrently. Another derived property from being in lockstep is that program instances following diverging control paths re-converge at the earliest possible point.
This automatic synchronisation makes communication between program instances possible without the need for explicit synchronization, e.g. through shared variables. Due to the guaranteed synchronization of the program instances, cross-lane operations are particularly suitable. The ispc standard library provides several cross-lane operations, which get directly mapped to the underlying hardware's instructions. However, results may depend on the SIMD width of the target platform and therefore the code may be platform dependent. Despite the synchronization, data races within a gang are possible, for example, if multiple program instances write to the same location, for example to an array with the same index.

3.4.2 Conditional Statements

The conditional statements `if` and `else` are the most basic constructs to express control flow. In ispc, varying conditional expressions get evaluated to a mask that represents the boolean result for each program instance. Then both, the if- and the else-branch get executed with the mask preventing side-effects from being visible. For the else-branch the mask is reversed. If all program instances follow the same control path, that is if the mask evaluated to all-on, all-off, or if the condition is of uniform type, only one branch will be executed.

An unstructured `switch` is provided with the same behavior as in C for expressions of uniform integer type. Other than in auto-vectorized code, a vectorized version of the `switch` statement is supported by simply using a varying integer type as expression. All cases will be executed with the execution mask indicating which program instance is active in each case.

3.4.3 Coherent Conditionals

If varying conditionals are in the common case expected to get evaluated by the program instances in a coherent way, and thus the execution mask is all-on or all-off, the programmer can provide a hint to the compiler. The keywords `cif`, `cfor`, `cdo` and `cwhile` feature an extra unmasked code path that will be taken in this case to increase the performance.

3.4.4 Basic Iteration Statements

ispc features `for`, `while` and `do` loops, including `break` and `continue` statements, in the same way as C/C++. A loop is entered by all active program instances concurrently. There is no implicit vectorization other than operations on varying data types, particularly no automatic loop vectorization. For the program instances it is possible to have different exit points by the use of exit conditions with varying data type. The execution mask is used to indicate the active program instances and the loop is left if no more program instances are active. This means that `break` and `continue` statements are allowed on varying conditions and program instances following different control paths within the loop. Via the built-in variables `programIndex` and `programCount`, it is possible to explicitly partition a loop across multiple program instances with more low level control. However, this needs some assumptions about the number of program instances and the data size and is therefore not recommended.

3.4.5 Parallel Iteration

Explicit parallel code can be written using the `foreach` loop. The foreach-loop iterates over one or more integer ranges, where each of the integer values is mapped to one program instance. Corner-cases, where the number of integer values does not divide evenly the number of program instances, are handled automatically by the compiler. The foreach-loop is more restrictive than other loops, as `break`, `return` and a nested foreach-loop is not allowed in the loop body. The execution order of the integer values is not defined. However, the compiler aims to subdivide the iteration domain in a way that leads to contiguous memory accesses in a linear way.

A second parallel for-loop, `foreach_tiled`, is provided with equivalent output. The only difference is the way in which the iteration domains are partitioned. For multidimensional iteration domains, e.g. matrices, the foreach_tiled-loop aims for compact regions, e.g. squares, instead of linear memory accesses. In certain cases this offers advantages in cache and control flow coherence.

3.4.6 Sequential Iteration

The `foreach_active` loop is a way to iterate sequentially over the active program instances only. The usecase is to prevent data races from happening as only one program instance is active in each iteration. For example, a global uniform counter could be incremented from every active program instance.

The `foreach_unique` loop allows to iterate over unique values in the set of varying data. The benefit is to not have to do the same calculations with the same values, if the program instances are often processing same values. The `break` and `return` statements are illegal in the body of both kinds of loops.

3.4.7 Unmasked

The execution mask can be re-established or deactivated using the keyword `unmasked`. In this case, the following code block will be executed by all program instances, regardless of previously by control flow statements established execution masks. This is particulary helpful to reduce the overhead of masking operations or to make use of all program instances within a foreach_active-loop.

3.4.8 Unstructured Control Flow

The `goto` statement is allowed in ispc, as long as it depends on uniform conditions only and it is guaranteed that all program instances are following this control path.

3.4.9 Functions

In ispc, functions work like in C. They have to be declared before use and can be exported following the C Calling Conventions. Also, ispc provides access to C functions with the `extern "C"` qualifier. However, as C can only handle uniform data types, all parameters of exported and extern functions have to match this requirement. If an extern function is called from ispc, only a single function call is made for the entire gang. This is a consequence of all parameters being uniform and calculations on uniform data being done only once. ispc adds an `inline` qualifier to force inlining and an `unmasked` qualifier to indicate all program instances as active at the beginning of the function. In this case the execution mask is not passed as parameter to the function. Like in C++, it is possible to overload functions by parameter type. As C linkage does not support name mangling overloaded functions can not be exported.

A small system for multithreaded task launches is provided. Functions of type `void` with `task` qualifier can be executed as an asynchronous thread when called with the `launch` keyword. Using `sync`, the parent task can wait for the started task to finish.

3.5 Compiler Infrastructure and Implementation

ispc is written in C++ and based on the *LLVM toolkit*. The programming style is strictly object oriented and therefore functions are implemented as methods that work on objects. The `main()` function calls the `Module::CompileAndOutput()` function, which executes the compiler phases shown in figure 5.

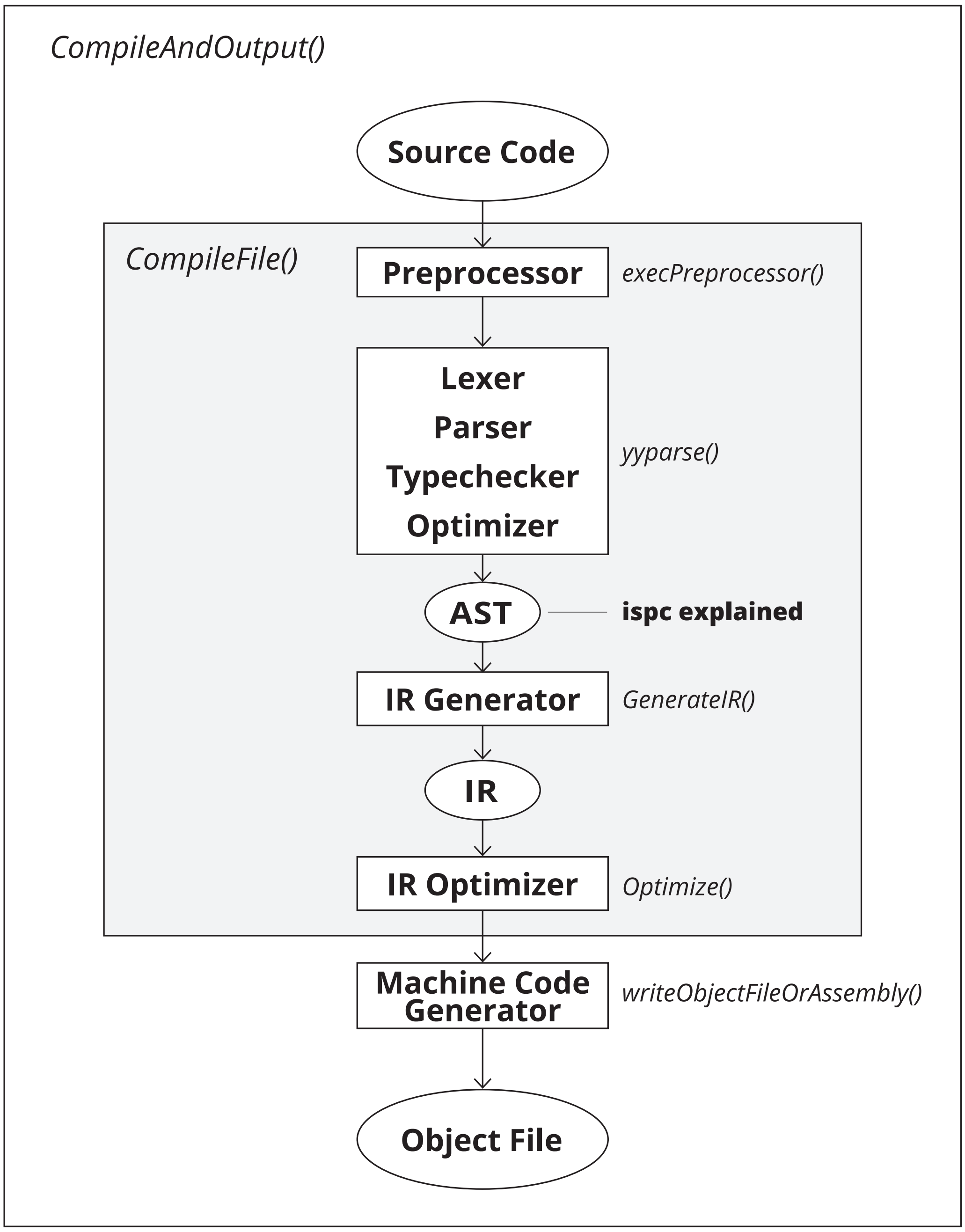


Figure 5: *The ispc compiler phases.*

For this purpose, the functions creates a module and calls the `CompileFile()` method. This method first runs the LLVM C preprocessor on the code, which is part of the Clang compiler front-end. *Flex* and *bison* are used for tokenization and parsing of the source code. A typechecking method is implemented for each expression and statement and directly called from it's constructor in the compiler's parsing phase. After typechecking, small code optimizations, such as constant folding, are performed on each expression. The lexical, syntactic, and semantical analysis as well as the front-end optimizations are completed in one step for each expression and generate the abstract syntax tree (AST). This is possible because of the requirement that every identifier has to be declared before use and thus all necessary information for typechecking is available. The disadvantage is that features which need more information for typechecking, such as templates, cannot be added to the compiler. However, a new compiler front-end is in development to overcome this restriction.

A method to generate the LLVM intermediate representation (IR) is called on the generated AST. And in the last step of `CompileFile()`, on the IR, a number of standard and custom optimizations are performed using the LLVM toolkit. Finally, the `CompileAndOutput()` function generates machine code appropriate to the target's instruction set.

4. ispc-explained

ispc-explained is a new compiler mode for ispc to provide source code annotations with additional information for the user. The code annotations of this compiler mode are the output of the compiler's front-end and should not be confused with compiler hints as in languages like Java or OpenMP.

In most of today's compilers, this kind of additional information is provided by integrated development environments (IDE) rather than by source code annotations. For example, Eclipse [10] offers analytical information for supported programming languages about types and functions by mouseover. The direct annotation of source code with additional information is rarely seen for high-level languages, but common practice for assembly. ICC can produce assembly listings with source code annotations that aim to understand the code and where it comes from. The linux tool opannotate [29] can annotate C and assembly code with previously obtained profiling information.

In this section, ispc-explained is illustrated starting with the motivation and implementation details, followed by the different features.

The following features are part of ispc-explained:

- **Type annotations:** The annotation of expressions with their type and of implicit type conversions.
- **Overload Resolution:** The annotation of function calls with the line number of the callee.
- **Gather and Scatter annotations:** The annotation of memory accesses with their kind of instruction.

As ispc-explained is implemented in the compiler's front-end, optimizations of the back-end are not included in the code annotations. This does not only apply to the optimizations of gather and scatter instructions (see section 4.3), but also neither masking information nor corresponding assembly code are provided.

In order to enable the new compiler mode, a flag `-explain=<num>` must be set as parameter for the ispc compiler. Num refers to the degree of completeness of type annotations. When enabled, the compiler takes the input file and copies it with the code annotations added to all lines of code. A distinct version for each targeted architecture gets created. The implementation of ispc-explained and all examples can be found online.(1)

(1) https://github.com/daniel-schuermann/ispc/

4.0.1 Motivation

To make it easy to adopt the language, ispc provides interoperability with C code and C syntax. However, the SPMD programming model differs from that of other general purpose languages, which work sequentially on single data streams. Code annotations can be a way to obtain a better understanding of the underlaying programming model. Like in C, in ispc the type of a variable is only written at the declaration of the variable. With an increasing number of variables, it becomes more difficult to keep track of the types of variables. As the code annotations are placed directly next to the source code, it is easier to understand the semantics and typing rules of the programming language.

The second advantage of code annotations is the debugging purpose. Implicit conversions and unexpected overload resolution can cause bugs that are hard to debug. The code annotations provide a direct feedback about the compiler's interpretation of the source code and therefore can help to identify the cause of unexpected results. *Gather* and *scatter* operations can be one reason for slow performance. ispc-explained can help to analyse performance issues and make these kinds of problems visible.

Finally, code annotations can also help to debug the compiler itself. Changes to the typing rules, implicit casts and other transformations, or the introduction of new features are directly visible from the compiler's front-end perspective, which can be used to check the correctness of the compiler.

4.0.2 Implementation

ispc-explained is written in a top-down approach on the AST representation of the source code and gets executed from the `CompileFile()` method, if the compiler flag is set and no error occured during typechecking. This is done via the `Module::annotateCode()` method, which was written to collect the necessary information and write the annotations into a file. First, information about every declared global variable is extracted from the symbol table. An annotation is generated for every declaration and stored in a hashmap with the linenumber being the key.

Afterwards, annotations for every function in the source file are generated. For this purpose, we iterate over the functions in the AST. If a function origins in the source file, an annotation is generated for the function declaration and added to the hashmap. Then, a recursive method, `Stmt::GetComments()`, is called on the function's body. This method collects all annotations for a statement, adding them to the hashmap, and is implemented for every statement subclass, e.g. for-loop. As the function's body is a statement-list statement, recursive calls for every statement in this list are made.

From occurring expressions, for example in conditional or expression statements, the annotations are retrieved by calling the `Expr::GetComment()` method, which returns a string. In case of the expression being nested, this method makes recursive calls and concatenates the strings.
After the annotations have been collected for each function, the source code file gets copied and the annotations are written next to each line of code in the copied version of the source code file.

4.1 Type Annotations

All possible types of the ispc language are described in section 3.3. The compiler mode ispc-explained is able to annotate all expressions, including subexpressions, with their type.

As deep nested expressions can lead to cumbersome type annotations that are difficult to read, the user can choose the degree of completeness of the annotated types. With a higher degree of completeness, the annotations contain more typing information, while a lower degree leads to better readability. A degree of 1 corresponds to having only the top-level expressions annotated. Better readability is also achieved by surrounding brackets and a concise form of all types. In the code annotations, the annotated expressions and declaration statements are preceded by their resulting type.

```
// annotated with degree of 1
uniform bool b = x%2 == 0;     // [bool] b = [bool] ((x % 2) == 0)

// annotated with degree of 3
uniform bool b = x%2 == 0;     // [bool] b = [bool] ([i32] ([i32]x % [const i32] 2) == [const i32] 0)
```

The notation of the types follows the C standard for the most part to provide convenience for the user. For varying types, the number of program instances on the target architecture is shown.

```
varying int v = 42;            // [4xi32] v = [const 4xi32] 42, 42, 42, 42
```

Regardless of the given degree of completeness, type conversions are always visible in the code annotations. This is due to the fact that type conversions change the resulting type of an expression and may be implicitly inserted by the compiler. The notation of type conversions is inspired by that of explicit type casts.

```
int r = x%2;                   // [4xi32] r = (4xi32)[i32] ([i32] x % [const i32] 2)
```

The example below shows again the while-loop from the hailstone example. The code is annotated with the command `-explain=2` which is a good compromise between readability and information. The target architecture is AVX with a vector width of 256 bit, into which eigth program instances are translated.

```
while(value != 1) {                     // for(; [8xbool] ([8xi32] value != [const 8xi32] 1);)
  iters++;                              // [8xi32]([8xi32] iters)++
  if((value % 2) == 0) //even     // if ([8xbool] ([8xi32] (value % 2) == [const 8xi32] 0))
     value /= 2;                        // [8xi32] value /= [const 8xi32] 2
  else //odd
     value = value * 3 + 1;       // [8xi32] value = [8xi32] ([8xi32] (value * 3) + [const 8xi32] 1)
}
```

The annotations show that, internally, while-loops are represented as for-loops without initialization and update. The execution of the while-loop is completely vectorized, what is indicated by the number of program instances, which process each value. The conditionals apply a mask on the following statements, as evaluated values are varying.

4.1.1 Issue: Type Promotion

The second example shows a problem that was previously found in [32]. This function takes two varying short integers and multiplies them. The result is returned as varying 32-bit integers to catch possible overflows.

```
int shortMult(int16 x, int16 y) {  // [4xi32] shortMult([4xi16] x, [4xi16] y)
    int res = x * y;               // [4xi32] res = (4xi32)[4xi16] (x * y)
    return res;                    // return [4xi32] res
}
```

However, other than in C, which guarantees a type promotion to integer before executing operations on a variable, in ispc the multiplication is executed first and afterwards the type promotion is performed. This results in possible overflows. Explicit type casting before the multiplication is a possible solution in this case.

4.2 Overload Resolution

ispc provides function overloading. This means that multiple functions can have the same name if they have different parameter types. If a function call is made to such an overloaded function, the compiler chooses the function that fits best with the arguments' types. This happens during the typechecking of the function call expression. For exact matches between parameter types and argument types, this decision is straightforward. However, if a function call involves implicit type conversions, ispc uses a cost model to determine the conversion with the smallest final cost. Although the precedence of the conversion costs is provided in the ispc user's guide [19], combinations of multiple parameters can cause an unexpected decision. The ispc-explained compiler mode shows the line number of the matched function on every function call. The line number is also provided if the function is not overloaded, the exception being built-in functions.

4.2.1 Issue: Type Conversion Cost

Figure 6 shows an example of an overloaded function that takes two arguments. The function defined first takes two varying integers and returns the dot product. The second function takes a uniform float and integer and returns the product. If the function is called with two uniform integers, neither of the two definitions match the arguments' types exactly. A call to the first function would need two broadcasts from uniform to varying type to be performed, while for a call of second function one conversion from integer type to floating point would be sufficient.

```
01  int foo(int x, int y){
02    return reduce_add(x*y);
03  }
04  int foo(uniform float x, uniform int y){
05    return x*y;
06  }
07  int bar(uniform int a, uniform int b){       // [4xi32] bar([i32]a, [i32]b)
08    return foo(a,b);                           // return [4xi32] foo<from Line 01>([i32] a, [i32] b)
09  }
```

Figure 6: *Unexpected overload resolution of a function call*

Unexpectedly, the first function gets called. The compiler decides that two broadcasts are less costly than one conversion from integer to floating point. The result of the function call is therefore not a multiplication but a dot product of two vectors and depends on the target architecture due to the individual vector size.

This example might seem artificially constructed, however it shows a kind of bugs that are difficult to debug. A solution, which is also recommended by a compiler warning, is explicit type casting to the appropriate parameter type.

4.3 Gather and Scatter

Data has to be loaded from memory by the CPU. A *gather* operation reads separate elements from individual memory locations and packs them in a SIMD register, whereas a *scatter* operation unpacks data and writes the elements to individual memory locations. On the other hand, a *vector load* can load multiple elements in one instruction from a contiguous memory location into a SIMD register, while a *vector store* writes them back. Until AVX2, which offers hardware support for gather operations, elements from non-contiguous memory locations had to be accessed by multiple scalar loads and stores.

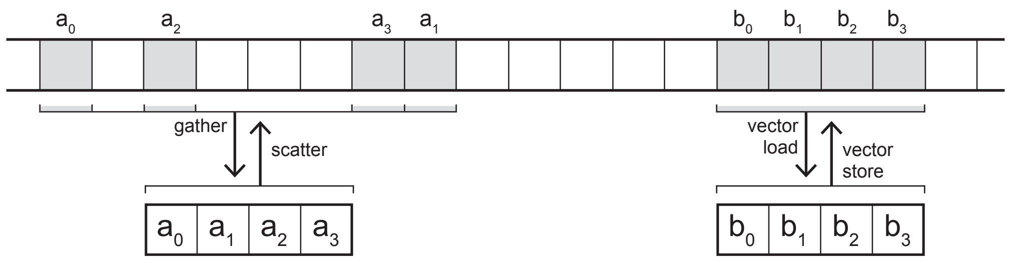


Figure 7: *Illustration of a gather/scatter operation in comparison with a vector load/store*

Because of the high latency of memory accesses, gather and scatter operations have a negative impact on the performance of a program. Furthermore, the accesses to non- contiguous memory locations can lead to more cache misses due to more memory pages that have to be accessed.

Memory accesses are in particular necessary when pointers are dereferenced. In ispc, these can be explicit pointer dereferences, reference dereferences or array accesses. References and arrays are internally represented as uniform pointers and thus point to a single location in memory, while varying pointers consist of one memory location per program instance (see section 3.3) and therefore should be avoided. Array accesses with a varying index can also make gather operations necessary, as the offset, and thus the memory location, may be different for each program instance. However, if the compiler is able to statically determine that the index is the same for each program instance or a linear sequence, the gather operation is optimized to a load or vector load instruction [18]. Because of this optimization not being applied on the AST, but in a later compiler phase on the IR, the code annotations will still indicate a gather operation.

Figure 7 shows three functions that each load values by dereferencing a pointer. The first function accesses an array with varying indices, the second function accesses an array with a uniform index and the third function dereferences a varying pointer and stores the values in a reference.

```
int load(int array[], int i) {            // [8xi32] load([8xi32*] array, [8xi32] i)
   return array[i];                       // return [8xi32]([gather!] [8xi32*] array[i])
}
int load(int array[], uniform int i) {    // [8xi32] load([8xi32*] array, [i32] i)
   return array[i];                       // return [8xi32]([vector load] [8xi32*] array[(i64)i])
}
void load(int* p, int &ref) {             // [void] load([i32* x8] p, [8xi32 &] ref)
   ref = *p;                              // [8xi32] ([vector store] [8xi32 &] ref)
                                                   = [8xi32] ([gather!] *[i32* x8] p)
}
```

Figure 8: *Three functions, annotated to show gather and scatter operations.*

The first function needs a gather instruction as the index argument can be different for each program instance, pointing to a different position in the array. For the second function, a uniform index is used. Therefore a single vector load instruction is sufficient to load all values for the program instances. In the last example, a pointer is dereferenced and the values are stored in a reference. The pointer can be different for each program instance, resulting in a gather to load the data. References on the other hand point to a single location in memory. Thus, a vector store or load is always possible.

5. Applications

In the following section, two examples are presented to show the possible performance increase of optimizing ispc code, but also potential issues when working with ispc. These examples and the applied optimizations are written with the understanding from the ispc analysis and with the information from ispc-explained.

The hailstone sequence from section 3.2 is an algorithm known for being not auto- vectorizable. The Collatz conjecture is that every positive integer number will complete the hailstone sequence in a finite number of steps. A research project that uses grid computing tries to verify the Collatz conjecture for larger numbers [7].

The second example is a kernel of the High Efficiency Video Coding (HEVC) from Fraunhofer Institute and used for video decoding [36]. As a modern multimedia application, it is a designated example for vectorizable programs.

5.1 Hailstone Sequence

The hailstone example has issues resulting in slow performance on computers with SIMD-extensions up to SSE4. The reasons for the slow performance is due to the diverging control flow and masking overhead. SSE does not offer predicated instructions and therefore all masking operations are inserted by the compiler manually. Other impacts on the performance are missing optimizations of modulo and division operations. For these integer operations, no specific instructions are available for the SIMD extensions.
In a first optimization step, we try to reduce the masking overhead and perform a *strength reduction*.

```
 ...
 while(value != 1) {
    iters++;

    if((value % 2) == 0) //even
      value /= 2;
    else //odd
      value = value * 3 + 1;

}
...
```

```
...
while(value != 1) {
    iters++;
    unmasked {
      if((value & 1) == 0) //even
        value = value >> 1;
      else //odd
        value = value * 3 + 1;
    }
}
...
```

Figure 9: *Original and optimized while-loop in the hailstone function using strength reduction and unmasking.*

5.1.1 Unmasking

The `unmasked` keyword prevents the compiler from including masking operations in the code block. Here, the affected statements do not have any side-effects regarding the program's output and are performed as long as at least one program instance is active. This means that the mask, derived from the while-loop, does not have to be applied on these statements and can be omitted.

5.1.2 Issue: Strength Reduction

In a manual strength reduction the modulo and division operations are transformed into an and and a shift operation, respectively. These operations are native on the processor's SIMD units, and thus much faster. Sophisticated compilers are able to perform the strength reduction automatically. The LLVM back-end of ispc offers some standard optimizations, which also include the strength reduction. However, differences in the used data types, here these are vector types, might be the reason for this optimization not being applied. A manual optimization shows a performance increase of 34%.

5.1.3 Control Flow Optimization

A second possible optimization is the reduction of diverging control flow. In the previous implementation, program instances which exit the while-loop are idle until all other program instances exit the loop as well. This behavior can be changed by rewriting the algorithm to using only one loop. The idea is that each program instance, after finishing the calculation of one value, directly picks up a new one, indicated by a global index. In this way, the results cannot be stored as vectors. However, the advantage of converging control flow outweighs this drawback. The code can be found in Appendix.

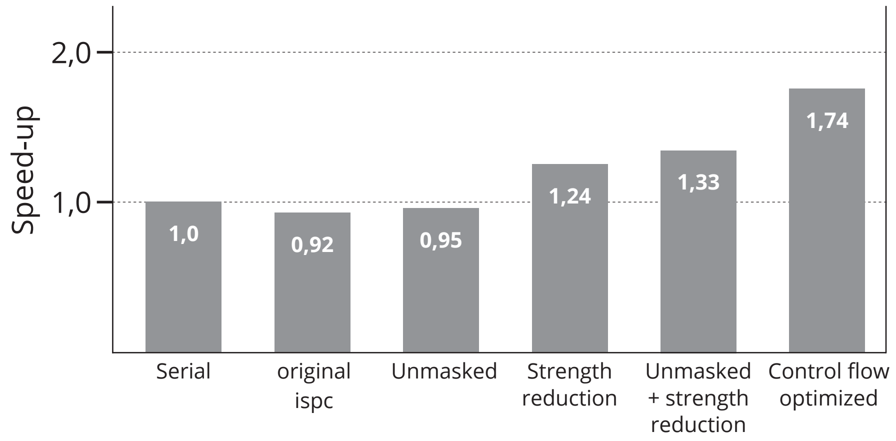


Figure 10: *Speed-ups obtained with three different optimizations applied to the original ispc code.*

5.2 HEVC-Decoder

The discussed HEVC-Decoder kernel was already part of a comparison of different approaches to vectorize code [32]. In this comparison, ispc faced different problems which resulted in a performance lower than the scalar version of the code. In this section, we try to identify the issues that hinder the performance. The kernel is a filter function to be applied to a video frame. The function calculates a dot product between pixels and coefficients. Afterwards, the result is clamped in an interval and stored back.

```
foreach (row = 0 ... height, col = 0 ... width) {
  int dst_index = row * dstStride + col;
  int src_index = row * srcStride + col;

  int sum = 0;
  sum += src[ src_index + cstep*0] * (int)coeff[0];
  sum += src[ src_index + cstep*1] * (int)coeff[1];
  sum += src[ src_index + cstep*2] * (int)coeff[2];
  sum += src[ src_index + cstep*3] * (int)coeff[3];

  dst[dst_index] = qpelC_store(sum, dst[dst_index], ...); // more parameters
  }
}
```

Figure 11: *ispc source code of the HEVC decoder function body. qpelC_store() clamps the result in an interval according to different parameters.*

5.2.1 Strength Reduction and Uniform Parameters

The `qpelC_store()` function was optimized in two ways. Like in the previous example, a strength reduction is possible as a performed modulo instruction has either one or two as operand only. The second optimization relates to the parameters, which are all varying. Changing the type to uniform prevents the parameters from being cast unnecessarily. Both optimizations provide only minor improvements in terms of performance.

5.2.2 Templates

The scalar version of this kernel is a templated function, and called, in this example, with seven different combinations of template parameters. The template parameters are used to eliminate unecessary branch instructions by *constant propagation* and *dead code elimination*. As ispc does not offer templated functions, either all template parameters have to be implemented as regular function parameters, resulting in low performance, or the function has to be implemented multiple times, which increases the code size significantly.

5.2.3 Issue: Dead Code Elimination

A different approach would be to `inline` the function and rely on constant propagation and dead code elimination of the function's arguments, like shown in this small example.

```
inline int helper(bool b, int x) {
   if(b) return -x;
   else return x;
}
int negate(int x) {
   return helper(true, x);
}
```

Like the previously discussed optimization, strength reduction, the dead code elimination will also not be applied. In this example, the condition gets evaluated, although it is known at compile time to be always true.

From this optimization, no speed-up but a small slow-down is obtained for the HEVC decoder. The only possible workaround is to implement the function multiple times according to the different constant values.

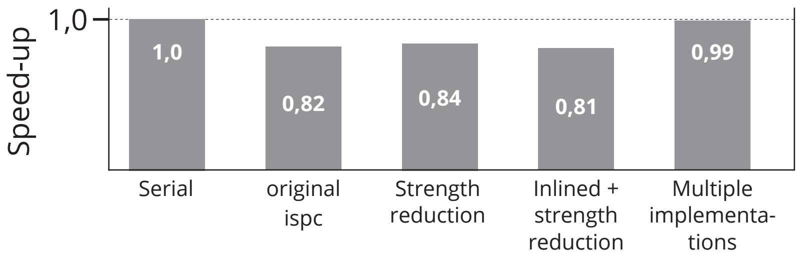


Figure 11: *Performance comparison of the different implementations of the HEVC decoder.*

5.2.4 Issue: Small Data Types

The SPMD programming model, and thus ispc, is not able to represent the ability of SIMD units to perform an instruction on either 32 bit integer values or on twice as much short integers. In ispc, there is always a fixed number of program instances running, which can not be changed during the execution of the program. As multimedia applications and also the discussed HEVC-decoder kernel often make use of short integers for data representation, it is not possible to fully exploit the processor's SIMD units.

A saturated conversion from integer to short integer is implemented in ispc as a combination of `min()` and `max()` functions, although since SSE2 there is an instruction available to do this directly. Furthermore, modern SIMD extensions support specific instructions for short integers, for example to perform multiplications and additions in one step. These instructions are not used by ispc and therefore the possible speedup is far below the theoretical performance of assembly code or instrinsics.

6. Conclusion

The Intel SPMD program compiler was developed to easily write vectorized code and maintain the high performance of code written with intrisic functions. For this purpose the SPMD programming model is used. In this thesis, ispc was analysed and an extension was written to output additional information from the compiler's front-end. With the understanding from the analysis and the compiler extension, two applications were optimized.

The analysis of ispc brought up that the SPMD programming model is well implemented and makes complex control flow possible within vectorized code. The C type system is consistently extended with varying types, which serve the purpose of vector types. Considering type safety, ispc has the same advantages, but also the same drawbacks as C. Compability to C/C++ provides easy integration with existing code.

The written compiler mode, ispc-explained, can provide useful information from the compiler's front-end. The type information and overload resolution showed two minor issues of ispc which can easily be avoided by appropriate code writing. The gather and scatter hints are limited in their utility as back-end optimizations are not presented. The missing information could be part of a further improvement in the future.

The example applications showed several issues with negative impact on the performance of ispc programs. To begin with, standard optimizations like strength reduction and dead code elimination are not applied to varying data types. This problem is back-end related and should be resolved in future releases of ispc by new LLVM optimizations, which can handle vector types. A bigger concern, regarding multimedia applications, is the missing support for small data types and specific instructions. As this problem is inherent in the language's programming model, there is no workaround possible. In the future, small improvements can be expected if more built-in functions, which are mapped to the corresponding instructions, are available.

On the other hand, the implementation of the Collatz problem showed that ispc stands out in the ability to vectorize code with complex control flow. In this case, implementations using intrinsic functions are particular difficult to write and ispc can make use of predicated instructions of the newer SIMD extensions.

The use of ispc could therefore be considered for high performance computing applications with complex control flow.

7. References

[1] Allen, J. R., Kennedy, K., Porterfield, C., & Warren, J. (1983, January). Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 177-189). ACM.

[2] AMD. (2012, March). 64-Bit Media and x87 Floating-Point Instructions (Vol. 5, AMD64 Architecture Programmer’s Manual), p.333. Advanced Micro Devices.

[3] American National Standards Institute. (1989). American National Standard Programming *Language C, ANSI X3.159-1989*.

[4] Andrei, Ş., & Masalagiu, C. (1998). About the Collatz conjecture. *Acta Informatica*, 35(2), 167- 179.

[5] Cardelli, L. (1996). Type systems. *ACM Computing Surveys*, 28(1), 263-264.

[6] Clang: A C language family frontend for LLVM. (n.d.). Retrieved June 24, 2016, from http://clang.llvm.org/

[7] Collatz Conjecture. (n.d.). Retrieved June 21, 2016, from http://boinc.thesonntags.com/collatz/

[8] Darema, F., George, D. A., Norton, V. A., & Pfister, G. F. (1988). A single-program-multiple-data-computational model for EPEX/FORTRAN. *Parallel Computing,* 7(1), 11-24.

[9] Deilmann, M. (2012). A Guide to Vectorization with Intel C++ Compilers. *Intel Corporation, April.*

[10] Eclipse desktop & web IDEs. (n.d.). Retrieved June 25, 2016, from https://www.eclipse.org/ide/

[11] Estérie, P., Falcou, J., Gaunard, M., & Lapresté, J. T. (2014, February). Boost. simd: generic programming for portable simdization. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing* (pp. 1-8). ACM.

[12] Ewart, T., Delalondre, F., & Schürmann, F. (2014, June). Cyme: A Library Maximizing SIMD Computation on User-Defined Containers. In *International Supercomputing Conference* (pp. 440- 449). Springer International Publishing.

[13] Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9), 948-960.

[14] GCC, the GNU Compiler Collection. (n.d.). Retrieved June 24, 2016, from https://gcc.gnu.org/

[15] Hailstone Number. [Def. 1] Weisstein, Eric W. (n.d.) *MathWorld--A Wolfram Web Resource*. Retrieved June 21, 2016, from http://mathworld.wolfram.com/HailstoneNumber.html

[16] Intel® C++ Compilers. (n.d.). Retrieved June 24, 2016, from https://software.intel.com/en- us/c-compilers

[17] Intel® Intrinsics Guide. (n.d.). Retrieved June 21, 2016, from https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[18] Intel® SPMD Program Compiler Performance Guide. (n.d.). Retrieved June 21, 2016, from http://ispc.github.io/perfguide.html#understanding-gather-and-scatter

[19] Intel® SPMD Program Compiler User's Guide. (n.d.). Retrieved June 21, 2016, from http://ispc.github.io/ispc.html

[20] Karrenberg, R. (2015). Whole-function vectorization. In *Automatic SIMD Vectorization of SSA-based Control Flow Graphs* (pp. 85-125). Springer Fachmedien Wiesbaden.

[21] Kretz, M., & Lindenstruth, V. (2012). Vc: A C++ library for explicit vectorization.*Software: Practice and Experience*, 42(11), 1409-1430.

[22] Larsen, S., & Amarasinghe, S. (2000). *Exploiting superword level parallelism with multimedia instruction sets* (Vol. 35, No. 5, pp. 145-156). ACM.

[23] Leißa, R., Hack, S., & Wald, I. (2012). Extending a C-like language for portable SIMD programming. *ACM SIG PLAN Notices*, 47(8), 65-74.

[24] Leißa, R., Haffner, I., & Hack, S. (2014, February). Sierra: a SIMD extension for C++. *In Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing* (pp. 17- 24). ACM.

[25] LiraNuna's Development Blog (2009, July 24). *SSE intrinsics optimizations in popular compilers*. Retrieved June 21, 2016,
from http://www.liranuna.com/sse-intrinsics-optimizations-in-popular-compilers/

[26] Mittal, M., Peleg, A., & Weiser, U. (1997). MMXTM technology architecture overview.

[27] Munshi, A. (2009, August). The opencl specification. In 2009 *IEEE Hot Chips 21 Symposium* (HCS) (pp. 1-314). IEEE.

[28] NVIDIA. (n.d.). CUDA Toolkit. Retrieved June 24, 2016, from https://developer.nvidia.com/cuda-toolkit

[29] Opannotate - Linux man page. (n.d.). Retrieved June 25, 2016,
from http://linux.die.net/man/1/opannotate

[30] OpenMP, A. R. B. (2013). OpenMP 4.0 specification, June 2013.

[31] Pharr, M., & Mark, W. R. (2012, May). ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar)*, 2012(pp. 1-13). IEEE.

[32] Pohl, A., Cosenza, B., Mesa, M. A., Chi, C. C., & Juurlink, B. (2016, March). An evaluation of current SIMD programming models for C++. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing* (p. 3). ACM.

[33] Robison, A. D. (2012). Cilk plus: Language support for thread and vector parallelism. *Talk at HP-CAST*, 18.

[34] single program multiple data [Def. 2]. Laplante, P. A. (Ed.). (2000). *Dictionary of computer science, engineering and technology*, p. 452. CRC Press.

[35] Slingerland, N. T., & Smith, A. J. (2000). *Multimedia instruction sets for general purpose microprocessors*: a survey. Computer Science Division, University of California.

[36] Sullivan, G. J., Ohm, J. R., Han, W. J., & Wiegand, T. (2012). Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12), 1649-1668.

[37] Wang, H., Wu, P., Tanase, I. G., Serrano, M. J., & Moreira, J. E. (2014, February). Simple, portable and fast SIMD intrinsic programming: generic simd library. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing* (pp. 9-16). ACM.

8. Appendix

```
// serial implementation in C
void hailstone(int values[], int length) {
    for(int i = 0; i<length; ++i) {
        int iters = 0;
        int value = i+1;
        while(value != 1) {
            iters++;
            if((value % 2) == 0) //even
                value /= 2;
            else //odd
                value = value *3 + 1;
        }
        values[i] = iters;
    }
}
```

```
// original implementation in ispc
export void hailstone(uniform int values[], uniform int length) {
    for(int i = programIndex; i < length; i+=programCount) {
        int iters = 0;
        int value = i+1;
        while(value != 1) {
            iters++;
            if((value % 2) == 0) //even
                value /= 2;//value /= 2;
            else //odd
                value = value *3 + 1;
        }
        values[i] = iters;
    }
}
```

```
// implementation with strength reduction and unmasked optimization
export void hailstone(uniform int values[], uniform int length) {
    for(int i = programIndex; i < length; i+=programCount) {
        int iters = 0;
        int value = i+1;
        while(value != 1) {
            iters++;
            unmasked {
            if((value &1) == 0) //even
                value = value >> 1;//value /= 2;
            else //odd
                value = value *3 + 1;
        }}
        values[i] = iters;
    }
}
```

```
// implementation in ispc with control flow optimization
export void hailstone(uniform int values[], uniform int length) {
    int i = programIndex;
    int iters = 0;
    int value = i+1;
    uniform int top = programCount;
    while(true) {
        cif(value != 1) {
            iters++;
            unmasked{
                if((value&1) == 0)
                    value = value >> 1;
                else
                    value = 3*value +1;
            }
        }else{
            values[i] = iters;
            foreach_active(index)
                i = top++;
            iters = 0;
            value = i+1;
            if(i > length) break;
        }
    }
}
```

All source code is available online at:

https://github.com/daniel-schuermann/ispc/