A stream language system for heterogeneous compute hardware including a multi-target compiler with an automated SIMD CPU back end

Sönke Ludwig

January 1st, 2011

Masterarbeit im Studiengang Informatik
am Institut für Softwaretechnik und Programmiersprachen,
Technisch-Naturwissenschaftliche Fakultät der Universität zu Lübeck,

Ausgegeben von
Prof. Dr.rer.nat. Martin Leucker

Betreut durch
Dr.-Ing. Martin Böhme

Abstract

Various interfaces and programming languages exist for accessing the heterogeneous computational units in modern computers such as CPUs, graphics cards or dedicated stream processors. These interfaces are often limited to a certain hardware manufacturer, or their platform support in general is limited. Those that are truly multi-platform depend on special hardware-vendor-specific drivers that are often prone to bugs.

These factors can make it especially difficult to deploy software on a large number of heterogeneous systems and be able to guarantee reliable results while still using the full potential of the hardware where possible.

In this thesis, a language with an accompanying compiler is described that is able to translate a SPMD (single-program-multiple-data) stream program with an OpenCL-like syntax to several target languages that can then be used with dedicated APIs to use graphics cards and other dedicated hardware. At the same time it can execute the code on the CPU using the LLVM compiler and virtual machine framework or generate C++ code that can be compiled using any existing C++ compiler. A special code transformation on the original program is done to yield efficiently vectorized code using SIMD instructions.

With the combination of the different outputs, it is possible to use the full computational power of each system while still being able to guarantee the reliability across all platforms and hardware configurations by using the CPU to compute the results in those cases where consistency or stability problems arise in the other paths. Significant performance improvements are expected for the SIMD output when compared to a traditional scalar compilation.

Statement of Originality

The work presented in this thesis is, to the best of my knowledge and belief, original, except as acknowledged in the text. The material has not been submitted, either in whole or in part, for a degree at this or any other university.

Lübeck, January 1st, 2011

....
Sönke Ludwig

Contents

1. Introduction

In recent years, since the era of single-core high-GHz processors has ended with the Intel Pentium 4, the topic of parallel computing has become increasingly important for all kinds of computation, not only in server and cluster environments. CPUs have been equipped with vector instructions, the number of processor cores has been increased and dedicated graphics and stream processor boards have been put on the market. All these technologies require the use of special APIs or instruction sets to optimally exploit their performance potential.

The approaches that hardware vendors have taken are numerous. Some use language extensions to hide parts of the complexity. OpenMP [DM98] is an example for this: Loop statements can be augmented with hints about the contents, and the compiler will generate multi-threaded code to execute the loop in parallel on multiple CPU cores – though without using the processor's vector instructions.

Compiler intrinsics are another type of language extension, where vector instructions can be directly used from within the usual programming language without resorting to assembler code. However, the code in this case is still executed on a single processor core. Also, with both approaches, the user has to know and decide what exactly is parallelized or vectorized and how this is done.

Certain compilers (e.g. GCC [GRS]) try to perform auto-vectorization by analyzing the code and detecting vectorizable statement sequences to automatically generate vector instructions. In theory, this approach does not require explicit knowledge of the programmer about vectorization, although this is still often necessary in practice to get good results. Also, the performance of these results is rarely optimal because the detection of such sequences can be very complex and most compilers can only detect simple patterns.

Finally, especially for using the computational units on graphics cards and dedicated stream processors, there are dedicated run-time compiled languages, where the device specific binary program code is sent to the device for execution at application run-time. These languages are usually based on a special program model called stream programs.

A stream program, instead of defining a loop running over several data elements itself, just defines a function that determines the value of each element of the result data array. The same program can then be executed in parallel on any number of processors to compute all data elements of an array. OpenCL [SGS10], GLSL [KBR08] and C for CUDA [Nvi11] are widely used examples for such languages.

Note that the term "stream" is also used for different, only partially related, concepts. Most notably, it is used to describe systems that use a serial stream of input and output messages for user input or, in distributed systems, for message based communication. A related concept is the data exchange in many areas of programming, where a stream of bytes (or words) is sent over facilities such as a file descriptor. The term here, however, is specific to the program model which is used because of its implicit parallelism. It is not related to message passing.

Some of the existing stream processing languages have the advantage that they run cross-platform on hardware of multiple vendors. OpenCL in particular is designed not only to run on all modern dedicated compute hardware but also on the system's CPU(s).

However, OpenCL, like the other stream processing languages, is based on a driver model. The compiler and run-time for an OpenCL implementation change from vendor to vendor and from driver version to driver version. This leads to a risk of software defects that can not be directly controlled, as driver bugs can show up or disappear at any time. This is in contrast to the traditional development model, where the compiler compiles the code once and then the program runs predictably on any system with the same CPU architecture[1].

Another problem with OpenCL is that there is no pre-shipped driver available for important platforms, such as Windows or Linux, and for some systems, such as FreeBSD and other Unix systems, there is no driver available at all. On server environments or environments where the hardware is known, this is not necessarily a problem, but if the software is to be shipped to typical non-expert end users with diverse hardware, this can be a severe problem. In particular, if there is no alternative code path based on C or another traditional language, some users might not be able to run the program at all.

This thesis describes a stream programming system that is designed to solve all these issues by compiling the same program for multiple target languages. These languages are then either compiled dynamically at run-time, or are statically linked to the application. One of these languages is C++, which provides the code path that can be trusted to run predictably on all systems. The other languages are then optionally used to harness the additional computational power of any dedicated compute hardware in the system – with the aforementioned problems that the corresponding systems have.

As an alternative to the C++ back end, a back end based on a virtual machine exists. LLVM [LA04] is targeted to make use of its just-in-time compilation capabilities. This back end allows to compile the program down to machine code at application run-time and directly execute it on the CPU. Since the same is true for the rest of the back ends, it is possible to use a set of useful programming patterns such as dynamically generating and running the program code.

To make optimal use of the system's CPU, the compiler has to be fed with explicit vector instructions. The idea here is to use a stream of vector instructions to compute multiple result data elements (of a single run of the original program) in parallel. Section 5 describes the approach that was developed to realize this goal. This approach is working on a global program level in contrast to auto-vectorization, which attempts to locally replace statement sequences by a single vector instruction. Finally, the generated code is run for different partitions of the output data array in separate threads so that all CPU cores are used.

[1]Except for rare cases such as the Pentium FDIV flaw (`http://en.wikipedia.org/w/index.php?title=Pentium_FDIV_bug&oldid=442679761`)

The system is intended to form a language ecosystem that provides optimal and scalable use of any hardware or software configuration with at least one safe path with reliable results. At the same time, the program has to be written and maintained only once as a stream program instead of writing one program for each target. Also, any additional or future language can be supported as a target language by implementing additional back ends for the compiler.

2. Related Work

The field of stream languages and high-performance computing already offers many different solutions. Some of them target CPU computing, some target the GPU, and others target cluster computing. But only few solutions exist that are able to fully use the potential of a modern PC with a multi-core CPU and a fast GPU. The most notable of these is OpenCL [SGS10]. OpenCL has a similar programming concept to the system described here. However, it does not provide a standard CPU implementation and depends on additional drivers to be installed in the system. It therefore does not fulfill the reliability requirements, which are one of the key points in this thesis.

The following sections describe the available alternatives and compare their features and properties to the SLURP framework described in this thesis. The comparison shows that no single solution exists that meets all the requirements we are looking for. Combinations of multiple systems will be able to meet the requirements, but at the high cost of developing and maintaining the code twice or even more often. Note that no cluster compute systems are listed here because they pursue different goals. The focus here is to optimally exploit the parallel computation units available on a single machine. Some cluster based systems could be added on top to provide network compute support.

2.1. Multi-threading systems

OpenMP [DM98] is a language extension for C/C++ and Fortran that allows loops to be broken up into multiple chunks which are then processed in separate threads. Special directives are used to tell the compiler how many loop iterations should be merged into one chunk and how many threads can be used to parallelize the loop. Listing 2.1 shows a simple example with an OpenMP annotated for loop. The advantage of this approach is that legacy applications can be easily enhanced by putting the right declarations in front of the critical loops. However, it often requires extensive tweaking to get the best possible results, and such local optimizations are often subject to Amdahl's Law [Amd67], which limits the gain that can be achieved by parallelization by the amount of sequential code in a program. The programmer also has to be careful not to introduce data race conditions by parallelizing code that has data dependencies between different loop iterations.

The OpenMP extensions allow to use the multi-core and simultaneous multi-threading [MBH$^+$02] features of a system. However, the additional computational power of SIMD operations needs to be exploited by other means. The GPU or other dedicated computational units in general cannot be used by OpenMP.

```
void multiplyArray(size_t array_size, float* array, float multiplier)
{
  #pragma omp parallel for
  for( size_t i = 0; i < array_size; i++ )
    array[i] *= multiplier;
}
```

Listing 2.1: A for loop with OpenMP annotations

An alternative solution for writing multi-threaded code is provided by Intel's Threading Building Blocks (TBB) [Phe08]. These provide a broader framework to C++ applications, including basic parallel constructs such as parallel loops and thread-safe containers and memory allocators. Also, the level of control over how the code is executed is higher. The design also facilitates doing the multi-threading at higher abstraction levels, often resulting in more scalable code with a lower proportion of sequential program code. Again, this is also just a CPU solution for multi-threading. SIMD instructions have to be employed by other means, and multi-threading requires explicit use.

2.2. Compiler intrinsics

Compiler intrinsic functions are mainly a higher-level alternative to assembler or inline assembler embedded in C++. They are commonly used to access the SIMD instruction set of the CPU from within the high-level program. An intrinsic has the form of a normal function and is translated to a matching machine instruction by the compiler. There are multiple advantages compared to inline assembler. The compiler is responsible for performing register allocation and can freely optimize the code because it exactly knows the effects of the intrinsics. Inline assembler, in contrast, typically completely disables compiler optimizations in the corresponding code section. Compiler intrinsics also provide a limited amount of hardware abstraction. For example, the Intel Itanium architecture has many of the SSE instructions of the x86 architecture available, albeit with a different encoding. The intrinsics work for both architectures, and the compiler is responsible for generating the actual machine code.

The disadvantage of using compiler intrinsics directly is that they are still specific to a certain instruction set extension. Support for SSE, MMX, AVX or other SIMD instruction sets has to be implemented separately, replicating the equivalent implementation code. Multi-threading will also have to be added using other means such as OpenMP, TBB or native threading.

2.3. Performance libraries

A higher-level concept is provided by several performance computation libraries. These libraries contain implementations of a number of common algorithms that can be used to create the final program. Common algorithms are Fourier transforms, encryption algorithms and vector operations. The implementations are hand tuned for multi-core and SIMD usage and provide a very high performance level. The drawback here is that they can only be used in an efficient way if the problem at hand matches one of the provided algorithms.

The libraries are often provided by a hardware vendor and optimized towards that vendor's own processors. Performance will often be acceptable when running on CPUs of other vendors, but also not optimal. A widely used library of this type with CPU-only support is Intel's Integrated Performance Primitives [Tay07]. AMD similarly offers, among others, the Accelerated Parallel Processing Math Libraries (APPML). Although these also allow high-performance CPU and GPU computing, they are based on OpenCL and thus share its disadvantages in reliability.

Microsoft provides a library called MSR Accelerator [TPO06] with support for both the CPU and the GPU. The library allows to write a computational kernel as C++ code, which is then translated with the help of operator overloading and templates into an internal representation that can be compiled for execution on the GPU and on the CPU. This library, however, is not cross-platform. It is also statically compiled together with the C++ code that surrounds it and has limited expressibility regarding loops and conditional statements.

Intel offers a similar library, called Array Building Blocks [NSL+11]. The library focuses on using the SIMD and MIMD capabilities of the CPU and does not offer GPU support.

2.4. Stream-processing frameworks

Several libraries exist based on a stream programming language as described in Section 1. These libraries execute the same function (or "kernel") for each data element of an output array. This formulation allows for simple exploitation of implicit parallelism. Furthermore, this stream programming model is very well suited for execution in a GPU.

Currently, the only system that fulfills all of the functional requirements that are imposed in this thesis, namely cross-platform support, CPU and GPU support and good usage of the CPU's capabilities, is OpenCL [SGS10]. The programming model is a stream processing model, similar to the one used in this thesis, with a similar language syntax. A short example kernel is shown in Listing 2.2. The API is available on several platforms, and all kinds of compute hardware can be used. To handle the different hardware types, each hardware vendor has to write an OpenCL driver. Each of these drivers contains the full OpenCL implementation, including a compiler.

The complexity of the drivers is one issue that OpenCL has, and this, in the experience of the author, leads to a relatively high number of software defects in todays implementations. Even the CPU drivers cannot be considered as a reliable code path because they also change from vendor to vendor and may introduce defects in new driver versions. Also, on certain systems, Microsoft Windows being one of them, OpenCL is

```
__kernel void multiplyArray(__global const float * a, const float b, __global float * c)
{
  // the current thread index, roughly the same as the current output
  // coordinate
  int index = get_global_id(0);
  c[index] = a[index] * b;
}
```

Listing 2.2: Simple OpenCL kernel

not installed by default. The user will need to download the appropriate driver and install it, both for the CPU and for the graphics card; but not all vendors have OpenCL implementations available. This makes OpenCL difficult to employ in today's consumer market.

Nvidia provides the CUDA [Nvi11] framework, which has been widely used for high-performance computations in the past. The CUDA framework is available for several operating systems but supports only recent Nvidia GPUs. With DirectCompute, Microsoft provides a similar library as part of its DirectX API. However, DirectCompute is limited to recent Windows operating systems. It also does not support the CPU as a computation target.

During the work on this thesis, Intel independently released their ISPC (Intel SPMD Program Compiler) [Int11] project. The compiler aims at making the SIMD capabilities available to the programmer using the implicit parallelism in stream languages. The ISPC language is a modified C dialect with some additional annotations and type modifiers. These annotations help the compiler parallelize the code. Interestingly, they appear to do a similar SIMD transformation to what is described in Section 5. However, there is no literature available about their approach apart from some comments in the user manual. The language is similar to the one developed in this thesis, but it requires the programmer to explicitly specify the loop over the output data elements which is parallelized. Two built-in variables, *programCount* and *programIndex*, are then used inside the loop to logically access the individual SIMD elements. This means that the underlying SIMD target is only partially abstracted by the language. The compiler targets only the SIMD units of the CPU. Multi-threading has to be done explicitly, and the GPU is not supported as a target.

2.5. Array languages

Another category of languages that complements stream-oriented languages is the class of array languages. Array languages are functional languages that directly operate on arrays to make use of the implicit data parallelism of array operations. They allow for more flexible algorithm specification, because there is no implicit data loop into which the algorithm has to fit, as is the case for more restricted stream languages. The programmer specifies the whole algorithm, including how the output is structured and how it is computed. In contrast, when using a stream language, the same algorithm may have to be broken up into multiple computation kernels that are then run sequentially on intermediate results. This is a reason why it can be difficult to generically translate array languages for execution on the GPU, especially on restricted stream language models, as they are used for GLSL and other shader languages.

Array languages, such as APL [Ive62] and SISAL [MSA$^+$83], have been used for a long time with an implicit approach to parallelization. However, they have only been used to target CPU and cluster systems, but not the GPU. A more recent development is Single Assignment C (SAC) [GS06]. The SAC program formulation is purely functional, although the syntax is largely similar to C. Arrays are treated as value types that can be returned from functions or passed as parameters. Uniform operations on arrays can be implicitly parallelized because of the functional nature of the language. Experiments with a translation from SAC to CUDA was done in [GDORT$^+$11]. However, only recent Nvidia GPUs support CUDA, which is why it has to be considered platform-specific in this context. It also does not allow dynamic compilation and does not use the SIMD units of the CPU for parallel execution, although the language model in general allows it.

2.6. Comparison

The following table gives an overview over all mentioned stream language systems. It lists the most important criteria and how they apply to each system. One criterion, the support for older GPUs, was added due to the fact that older graphics card generations, although still being quite fast, lack certain features that would make them OpenCL compatible. These GPUs can therefore only be employed by using a shader language such as GLSL. In some years this criterion will probably not be a valid differentiator anymore, however. The first column of the table lists the characteristics of the system described in this thesis termed SLURP (Stream Language Unified Runtime Programming), followed by the alternatives.

	SLURP	OpenCL	DirectCompute	MSR Accelerator
Cross-platform	yes	yes	no	no
Architecture-independent	yes	yes	yes	yes
No special driver required	yes	no	yes	yes
Supports CPU	yes	yes	no	yes
Uses multiple cores	yes	(yes)	no	yes
Uses SIMD	yes	(yes)	no	(yes)
Supports GPU	yes	yes	yes	yes
Supports older GPUs	yes	no	no	no
Reliable fall-back path	yes	no	no	yes
Frees the programmer from hardware specifics	yes	yes	yes	yes
Full expressibility	yes	yes	yes	no
Supports direct memory access	no	yes	yes	(no)

	OpenGL	Direct3D	IPP	OpenMP	Compiler-Intrinsics
Cross-platform	yes	no	yes	yes	yes
Architecture-independent	yes	yes	(yes)	yes	no
No special driver required	(yes)	(yes)	yes	yes	yes
Supports CPU	no	no	yes	yes	yes
Uses multiple cores	no	no	yes	yes	no
Uses SIMD	no	no	yes	no	yes
Supports GPU	yes	yes	no	no	no
Supports older GPUs	yes	yes	no	no	no
Reliable fall-back path	no	no	yes	yes	yes
Frees the programmer from hardware specifics	yes	yes	yes	(yes)	no
Full expressibility	yes	yes	no	yes	yes
Supports direct memory access	no	no	(no)	yes	yes

Figure 2.1 shows a broader comparison with more systems, but only considering the four key requirements CPU support, GPU support, platform independence and reliability. Although not all systems are listed, it shows a tendency that many systems exist that support either CPU or GPU, but few that support both. Those systems that do support GPU and CPU are either platform specific, possibly unreliable, or do not support some of the remaining criteria, such as dynamic compilation or high language expressibility.

No existing system matches all criteria; to achieve this, it is always necessary to program for at least two of these systems. The multi-target compiler described in this thesis can fulfill all requirements, except for direct memory access, while removing the

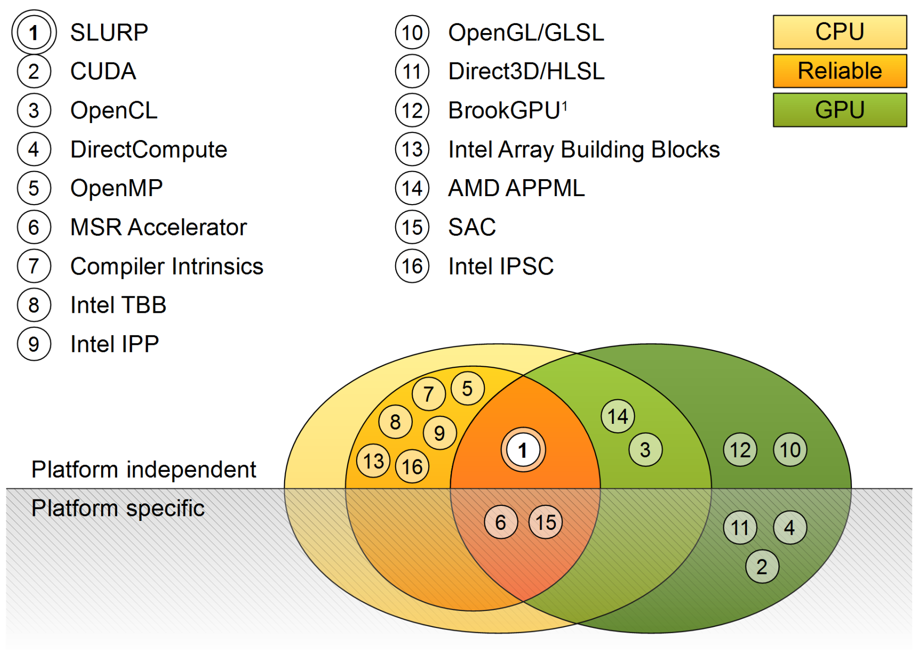


Figure 2.1.: Comparison of existing single-machine compute systems according to the key requirements ([1][BFH+04])

burden to implement and maintain the algorithms repeatedly. Direct memory access is in conflict with support for older GPU hardware and has thus not been included. Once the requirement for direct memory access becomes more important than supporting this type of GPU, the SLURP language can be extended with a full pointer type or with built-in functions to access memory freely.

3. The Language

The core part of the system is the language definition. Defining the supported language features is critical for being able to optimally parallelize the input program. Also, target languages such as GLSL may only support certain operations – so the lowest common denominator is the gauge to which the language needs to comply. For the rest of the thesis the language is referred to as the Stream Language Unified Runtime Programming (SLURP) language.

The general syntax of the language has been chosen based on the C family of languages (For details see [KR78] or [WG107]). Especially the expression syntax, the curly braces for blocks and the semicolon as the statement delimiter are taken from C. This choice is based on the fact that almost all popular languages in the stream language category (Cg [MGAK03], HLSL [PM03], GLSL [KBR08], C for CUDA [Nvi11] and OpenCL [SGS10]) have a similar syntax and users will be instantly familiar with the language.

Some features that are present in all or some of the mentioned languages have been left out to simplify the implementation and to allow some optimizations that would have been difficult or impossible otherwise. The three most important features in this category are global variables, pointers and free access to the destination data buffer. Due to the lack of these features, the language is conceptually purely functional. No function can have side effects and the result depends only on the function's parameters. However, inside of the function bodies, procedural-style statements and variables can be used. There is no language support for data pattern matching as found in some other functional languages, instead, procedural loop and control flow statements are supported.

```
alias point = float2;

function point subTileCoords(point pt, float tile_width )
{
  return pt % tile_width;
}

kernel float4 myKernel(
    point output_coordinate,
    sampler2 input_array
  )
{
  float2 coord = subTileCoords(output_coordinate, 100);
  return sample(input_array, coord);
}
```

Listing 3.1: A simple SLURP program

A program consists of a list of global declarations, where a global declaration can be a type definition, a function definition, or a kernel. A kernel is a function marked to be the main entry point for a filter. Kernel functions also must have the current output coordinate as their first argument, followed by an optional list of uniform parameters (see next section) that are later passed from the host application to the kernel. See Listing 3.1 for an example of a simple program with all three types of declarations. The complete language grammar is listed in Appendix A.

(For every kernel in a program, the compiler will create a public filter procedure in the generated code. This procedure is then called by the application to process a set of input arrays.)

Apart from the compiler-defined types and functions, there is a special file 'core.scl', which is parsed just before the input program. It contains additional functions and types that are available to the program. All vector and matrix types, as well as functions and operators dealing with these types, are defined in that file. It also allows to override functions that would otherwise be generated by the compiler in the SIMD back ends. The contents of the file can be considered

3.1. Parallelism model

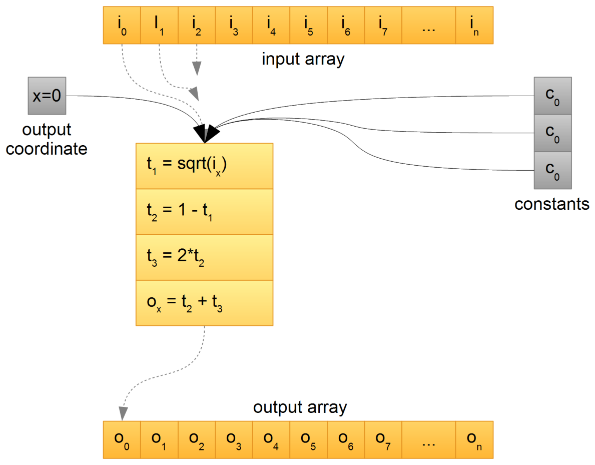


Figure 3.1.: Conceptual data flow in a stream program with one input array and several input constants

The programming model used for SLURP is the so called stream language model, also called "single program, multiple-data" (SPMD) [Dar01]. SPMD is a form of "multiple instruction, multiple data" (MIMD), where many elements are processed in parallel by a number of independent processors, but, in contrast to general MIMD, using the same program. Input and output data is handed to the stream program in the form of arrays or scalar values. For each element of the result array, the program is called with the input data, as well as the current output coordinate or output index of the currently computed element. The return value of the program is then stored at the corresponding position in the output array. Figure 3.1 shows the general data flow. A notable constraint here is that the output array is write-only and distinct from the read-only input arrays, so that there can be no dependencies on earlier output elements.

This dependency restriction allows the same program to be conceptually executed independently for every data element in the output data array. It thus allows for unrestricted parallelization because there are no data dependencies or concurrency of any kind. There can be hidden dependencies such as false sharing [BS93], where the distinct memory regions lie on the same cache line of the CPU's cache and thus are subject to synchronization between different CPU cores. However, these are hardware implementation details that can be handled by the compiler.

An obvious possibility here is to use multi-threading to compute parts of the result in parallel. This is a very important step on today's multi-core processors (and in similar form for server clusters) and will result in very good scalability because of the memory friendly nature of the program (lack of data dependencies, single write operation per data element). However, modern CPUs are not only parallel at the CPU core level but also have a growing number of parallel computation units in each core. Although the processor tries to use up as many of those computation units as possible by using means such as out-of-order execution or simultaneous multi-threading (SMT, also known as hyper-threading) [MBH$^+$02], to get the maximum performance, it is necessary to use a special single-instruction-multiple-data (SIMD) instruction set to fully exploit their possibilities.

The approach that is used here to exploit the performance gain of SIMD instructions is based on spatial coherence. Many algorithms have similar input for neighboring data elements, and the code path taken for those elements is either the same or similar in most cases. The idea is to exploit this fact by computing four or more array data elements in lock-step (see Figure 5.2). This way it is possible to guarantee full use of the SIMD computing power, as long as the code paths stay in sync. This is in contrast to auto-vectorization, which is heavily dependent on the sequence of computations that is defined by the compiled program (Section 7.1 provides a more detailed comparison of the two approaches).

Complications arise if the program contains conditional statements or loops where the execution of neighboring pixels takes different paths. In this case the compiler generates code that tries to compute the conditional or loop as efficiently as possible and then continues with the normal code execution in lock-step. Information from the type system can be used in many cases to keep the code executing in lock-step, even in the presence of such constructs. This is detailed in in Section 5.

3.2. Types

3.2.1. Primitive types

The type system is based on the three basic types *bool*, *int* and $float$. The semantics match C++'s types [ISO03], where *bool* can be either *true* or $false$, *int* is a two's complement integer and $float$ is an IEEE-754 32 bit floating point number. The *int* type is always 32 bit wide, in contrast to C/C++. The exact binary representation of the *bool* type in the final compilation result can vary from platform to platform. For example, the SIMD back end will represent *bool* as a 32 bit wide integer with the two possible values -1 and 0 for $true$ and $false$. This way, the *bool* type can be efficiently represented as part of an SSE register.

Finally, pointers are supported as an opaque handle inside of the language in the form of the *pointer* type. No operations apart from assignment are supported on this type. This includes the inability to construct a value of type pointer. The reason for having a pointer type in the language is simply to be able to define the data types that carry the input and output arrays for the program. Structures containing pointers can be passed to predefined functions that are implemented directly in C++ or LLVM code, which can in turn access the corresponding memory.

3.2.2. Aggregate types

Any type can be aggregated into an array of fixed dimensions, or multiple types can be combined as a structure, resulting in another (complex) type. The semantics are again compatible with C++. However, in addition to the one-dimensional arrays supported by C/C++, multi-dimensional arrays of fixed size are also supported. The predefined matrix types such as $float4x4$ are defined to alias two-dimensional array types (i.e. $float[4, 4]$, see the reference in Listing B.1). This feature allows a natural formulation of matrix types and definition of mathematical operators.

A special kind of complex type is the function type. This type is not syntactically constructible but is used for internally representing functions in the compiler. A function type consists of a return type and a list of parameter types.

3.2.3. The Uniform modifier

Any type can be tagged with the $uniform$ modifier. The uniform modifier tells the compiler that a certain variable will stay the same across all invocations of a kernel on a single data set. Any attempt to put a non-uniform value into a variable that is typed as uniform will result in a compile error. The knowledge that a particular value stays constant for the duration of the computation allows the compiler to perform a number of important optimizations. The SSE back end makes extensive use of this type modifier to generate less general but faster code in some situations (for an example, see Section 5.4 about conditionals that can be simplified). Another possibility, which is not implemented in the version of the compiler described here, is to move any computations that only depend on uniform or constant variables outside of the general data loop. The results are then only computed once and passed precomputed to the kernel for each computed data element.

3.3. Statements and Expressions

The organization of the program code in statements and expressions is very similar and forms a subset of the statements in C. The supported statement types are:

declaration statements, expression statements, compound statements, *return* statements, *if/else* statements, *do/while/for* statements

Declaration statements support only one variable declaration per statement. The remaining statements behave like their C/C++ counterparts [KR78]. Listing 3.2 shows an example, where all types of statements are used.

The expression syntax is also similar to C, except for some missing operators such as the element accessor '->' and the namespace separator '::'. The cast operator also has a *cast* prefix and the array index operator supports multiple arguments for indexing into multi-dimensional arrays. The operators have the same precedence order and associativity as in C. The supported operators are:

(), =, ||, &&, ==, !=, <=, >=, <, >, +, -, *, /, %, cast(), [], ., !, ++, –

```
function float test()
{
  float x; // declaration statement
  float y = 1.0; // declaration statement

  x = y * 2.0; // expression statement

  if( x > 1.0 ) // if/else statement
  { // compound statement
    do { // do-while statement
      x = x * 0.5;
    } while( x > 1.5 );

    while( x < 1.0 ) // while statement
      x = x + 0.1;

    for( uniform int i = 0; i < 10; i++ ) // for statement
      x = x * 0.5 + y * 0.5;
  } else x = 1.0;

  if( y > x ) // if statement
    x = y;

  return x; // return statement
}
```

Listing 3.2: Example code showing all statement types

3.4. Input array access

Accessing the input data arrays is done using the library function *sample*. The *sample* function takes a *sampler* parameter and a coordinate and returns the associated array element of the array corresponding to the sampler. An example kernel that performs a vertical edge detection on a 2-dimensional image is given in Listing 3.3. The reason for having a function instead of using a direct array access is that this abstraction provides a number of possible behaviors.

The first possibility is to allow fractional coordinates (non-integer coordinates) and perform automatic interpolation between neighboring array elements. The GPU usually has a dedicated texture filter unit, which is able to do this interpolation with no additional cost. Applications for this feature include lookup tables and scaling of images.

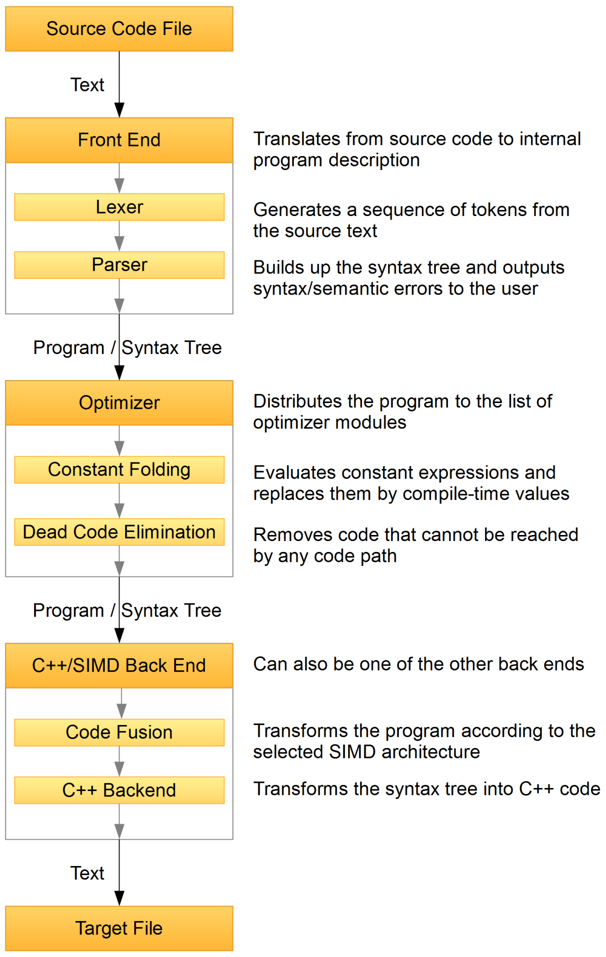
Another possible feature is the specification of a border-wrapping mode. In the current implementation the *sample* function will return the nearest element of the array if the coordinates lie outside of the array bounds. In future implementations wrapping modes such as repeating the array contents periodically could be added.

Finally, the sample function could provide an abstraction for the data type of the input array. For example a conversion from the input array element data type to a *float* based type could be done on-the-fly and thus make the kernel partially independent of the input data type.

```
kernel float4 verticalEdgeDetect(float2 coord, uniform sampler input_image)
{
  float4 current_pixel = sample(input_image, coord);
  float4 previous_pixel = sample(input_image, coord - float2(1.0, 0.0));
  return abs(current_pixel - previous_pixel);
}
```

Listing 3.3: Example kernel with *sampler* input argument

4. Compiler Architecture



Source Code File
Text
Front End
Translates from source code to internal program description
Lexer
Generates a sequence of tokens from the source text
Parser
Builds up the syntax tree and outputs syntax/semantic errors to the user
Program / Syntax Tree
Optimizer
Distributes the program to the list of optimizer modules
Constant Folding
Evaluates constant expressions and replaces them by compile-time values
Dead Code Elimination
Removes code that cannot be reached by any code path
Program / Syntax Tree
C++/SIMD Back End
Can also be one of the other back ends
Code Fusion
Transforms the program according to the selected SIMD architecture
C++ Backend
Transforms the syntax tree into C++ code
Text
Target File

Figure 4.1.: Data flow through the different compiler stages

The compiler framework is split up into a dynamic library that contains the actual compiler functions and a command line tool for invoking the compiler and outputting the compiled code into a file. The compiler library supports three modes of operation:

- Compiling to a text file – this is supported for the C++ and GLSL back ends.
- Compiling to a string that is returned to the calling application – also supported for the C++ and GLSL back ends.
- Compiling directly to a virtual machine for immediate execution – this method is supported for the LLVM back end.

The compiler library is split up into three main components: the front end, an intermediate representation optimizer and the back end. The front end contains the lexer and parser components and transforms the input text file into a semantically checked syntax tree. This syntax tree is then optionally processed by the intermediate representation optimizer. Finally, the back end performs the final transformation into the target language, which is output as a text file. Figure 4.1 shows the general data flow of the compiler components with the SIMD/C++ back end chosen in this example.

The size of the code base is comparatively small in relation to the functionality that the compiler offers. The software design was developed with attention to low redundancy and several properties of the D language allow for terse code. In particular, the module system without header files, the *auto* keyword for automatic type inference, the property syntax, compile-time reflection, and features such as the $foreach$ loop helped to get smaller and more readable source code – especially compared to the equivalent C++ code (Section 7.2 discusses the D language further). The following table lists some code metrics of the compiler library:

Number of files	35
Number of classes	40
Number of free functions	47
Lines of code	6.374
Number of bytes	370.156
Average number of characters per line	58

The next sections will give an overview over the individual components and their interfaces.

4.1. Lexer

The first stage of the compiler, the lexer, takes the input text file and then outputs a stream of so-called tokens. These tokens represent the basic atoms used by the parser to generate the high-level syntax tree. A token can be an identifier, a floating-point number or one of the predefined character sequences defined in Listing 4.1.

```
==, !=, <=, >=, <, >,
+=, -=, *=, /=, %=, =,
&&, ||,
++, --,
!, ~, &, |,
., ;, (, ), [, ], {, },
+, -, *, /, %
```

Listing 4.1: Special tokens

Identifiers, as in C, start with a letter or an underscore and can contain alphanumeric characters and underscores. Numbers can be in any standard floating-point form. Whitespace in the form of tabs, spaces and newlines is ignored. Also, C/C++-style comments are recognized and skipped.

The lexing is done by scanning the input string sequentially, ignoring whitespace and comments, until the beginning of a token is found. The token is then scanned for its end and a slice (see Section 7.2) of the string representing the token is returned. Since all strings are slices of the original input string, the memory required for internal identifiers during compilation will never exceed the size of the input file.

The lexer supports a look-ahead of one token to support the LL(1) grammar of the language (see Listing A.1 for the full grammar). This means that the token that was read last, can be put back into the token stream, so that it will be the next token that is returned.

4.2. Parser

The parser gets a lexer as its input and successively takes the token output, parses it and outputs a syntax tree. In addition to parsing and checking for correct syntax, all semantic rules are enforced during the same process.

Because all variable declarations, function definitions and type definitions must be done upstream in the source code before they can be used (no forward references are supported), parsing can happen in a single pass over the source code. This in turn allows for a fast and simple parser.

The implementation of the parser was done in native D code (see Section 7.2). In contrast to using a parser generator such as YACC [Joh79] or Bison [DS90], this requires no special adjustments to the build system and allows for a clean and obvious source code representation. This in turn simplifies debugging and makes handling ambiguous corner cases (such as the if/else statement, see [Wik11]) more explicit. An actual code snippet of the expression parsing code is shown in Listing 4.2. All operators, except for assignment, array indexing and the dot operator, are transformed into function calls, as seen in the code snippet in Listing 4.2.

```
// and_expr -> equal_expr ['&&' and_expr];
Expression parseAndExpr()
{
  auto left = parseEqualExpr();

   if( m_lexer.peek() != "&&" ) return left;
  m_lexer.scan();

  auto right = parseAndExpr();
  return createFunctionCall("opAnd", left, right);
}

// equal_expr -> add_expr [('==' | '!=' | '<' | '<=' | '>=' | '>') equal_expr]
Expression parseEqualExpr()
{
  auto leftExpr = parseAddExpr();


  string opname;
  switch(m_lexer.peek().text){
    default: return left;
    case "==": opname = "opEqual"; break;
    case "!=": opname = "opNotEqual"; break;
    case "<":  opname = "opLess"; break;
    case "<=": opname = "opLessEqual"; break;
    case ">=": opname = "opGreaterEqual"; break;
    case ">":  opname = "opGreater"; break;
  }

  m_lexer.scan();

  auto right = parseEqualExpr();
  return createFunctionCall(opname, left, right);
}

// add_expr -> mul_expr [('+' | '-') add_expr]
Expression parseAddExpr()
{
  auto left = parseMulExpr();

  string opname;
  switch(m_lexer.peek().text){
    default: return left;
    case "~": opname = "opConcat"; break;
    case "+": opname = "opAdd"; break;
    case "-": opname = "opSub"; break;
  }

  m_lexer.scan();

  auto right = parseAddExpr();
  return createFunctionCall(opname, left, right);
}
```

Listing 4.2: Code snippet of the parser implementation

4.3. Intermediate representation

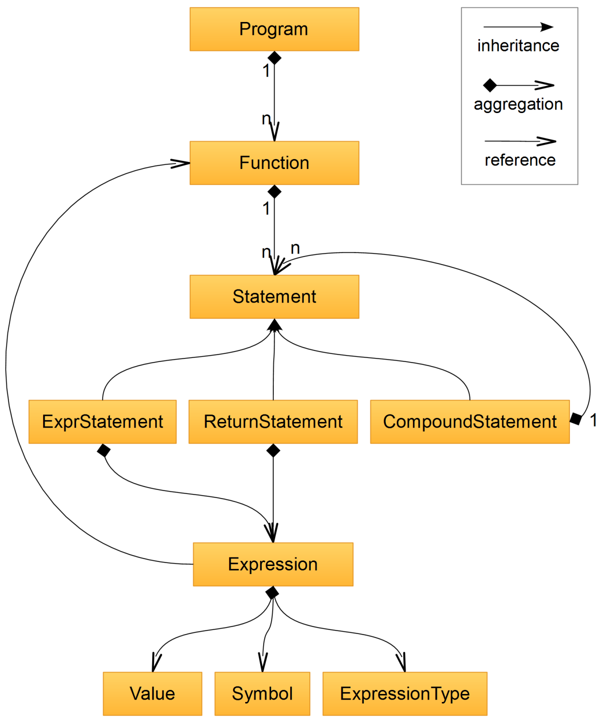


Figure 4.2.: Class diagram of the intermediate program representation

The intermediate representation used for communicating between the front end, the optimizer and the back end is implemented in the form of a syntax tree. The basic tree elements are the classes *Program*, *Function*, *Statement* and *Expression*. All of these classes derive from a common base class *AST*. The *AST* class provides abstract access to the syntax tree and allows generic traversal and transformations. The *Statement* class provides a number of additional abstract methods that are implemented in its subclasses. These methods allow traversal and modification of all contained expressions and statements. The optimizer (Section 4.4) and the SIMD code generator (Section 5) both make extensive use of these. The AST structure can also be used to serialize the syntax tree to disk. A pre-compiled binary format is an example application that can be realized with this. See Figure 4.2 for a breakdown of the class hierarchy.

Because the SLURP language is based on C, just as all supported high-level target languages are based on C (C++ and GLSL), the same syntax tree form can be used throughout the compiler all the way down to the back end, where it can then be easily transformed to the target language. Multiple transformations to and from different intermediate representations become unnecessary. One exception is the transformation of operators to function calls, which has to be reverted for the target languages.

Transforming operators in each language back end is a necessary step, as different languages sometimes have different semantics for the same operator. An example is the multiplication operator when used with matrix types. GLSL defines the operator to perform a matrix multiplication, while other languages such as HLSL (Direct 3D) use element-wise multiplication.

4.4. Optimizer

The optimizer consists of an ordered list of sub-modules that consecutively process the parsed program. Each sub-module is implemented as a simple function that gets a program object as its input and modifies this program object in-place. The order and type of optimizer modules is specified by the high-level application logic and can be extended by defining additional modules and inserting them into the optimization queue.

The following sections describe the optimizer modules that are already present in the compiler. For this thesis, only basic optimizers have been implemented. Modern compilers in general have a plethora of different optimization algorithms ([Muc97] gives a solid overview). But since all target languages of this compiler are again compiled using an existing, typically more advanced compiler, the gain of aggressive optimization at this stage is quite low. An important exception is the SIMD back end, which does some important optimizations outside of the general optimization queue that are specific to the back end (see Section 5.3.

4.4.1. Constant folding

The very first optimization that is done to the program is constant folding. During constant folding, the program is recursively searched for all contained expressions. Every expression is then searched for sub-expressions which only have constants as their arguments. These expressions are then evaluated if possible and replaced by a constant. This is done until no such replacements are possible any more.

This process works only for expressions which have a compile-time version of their operation available. Most of the built-in functions have a compile-time implementation. Additionally, user-defined functions can be evaluated using a small interpreter that is built into the compiler, as long as they use only data available at compile-time.

Constant folding is an important foundation for other optimizations that can make assumptions if certain values are constants. Examples are dead code elimination (Section 4.4.2) and uniform loop unfusion (Section 5.5).

4.4.2. Dead code elimination

Dead code elimination attempts to remove those parts of the code that can never be reached. The degree to which these parts can be detected depends on the refinement of the control flow analysis employed (where perfect detection is impossible due to Turing's undecidability theorem).

The method used here tries to catch the most simple cases where a constant expression of a conditional or loop statement yields either true or false. In this case the conditional statement can be replaced by either the true-branch or the false-branch, and loops with zero iterations can be eliminated.

Note that although this is a very simple detection strategy, it covers most real-world cases in stream programs and arguably almost all cases that matter. The primary reason for this optimization is to allow conditional statements together with compile-time constants to disable certain code paths without overhead.

4.5. Back end

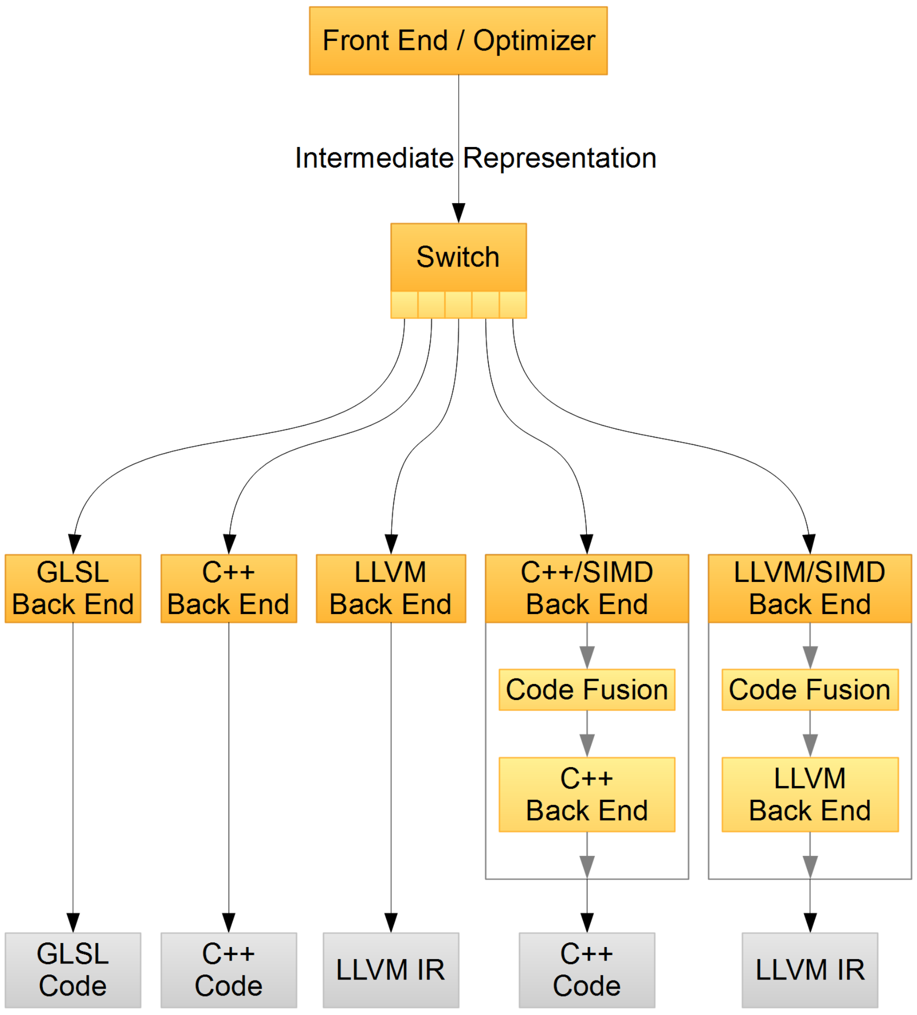


Figure 4.3.: Structure of the compiler back end

The compiler has five switchable back ends for generating its output, a GLSL back end, two C++ based back ends and two LLVM based back ends. The most basic back end is the generic C++ back end, which will output C++ source code that is in general very similar to the input program code. This generic output provides a target that

```
kernel float test(float2 pos)
{
  float z = pos[0] + pos[1];
  z = z + 0.5;

  if( pos[0] >= 0.5 ){
    z = z - 0.5;
  }

  return z * 0.5;
}
```

Listing 4.3: An example kernel function

can be further compiled using any C++ compiler. That way, for every platform with a C++ compiler there is always a robust kernel compilation available that provides correct output, assuming that there are no defects in the CPUs functionality.

The GLSL back end derives from the C++ back end and modifies the output by overriding certain functions and providing a mapping between functions and operators specific for the target language. Finally, there is a back ends based on LLVM [LA04]. This back ends does not output source code but generate a program representation in LLVM intermediate code. This code allows the stream program to be executed at run-time using a virtual machine.

The remaining two back ends combine the C++ or the LLVM back end with a code transformation for exploiting the SIMD functionality of the CPU. Figure 4.3 shows a diagram of the full back end structure including the SIMD back ends.

The following sub sections will use the example kernel in Listing 4.3 to illustrate the output of the respective back ends. The kernel performs a simple computation based only on the output coordinate of the result array. An if statement shows a simple example of conditional control flow.

4.5.1. SIMD back ends

The SIMD back ends are meta back ends in the sense that they first transform the scalar program to a program that uses SIMD operations in the form of operations on arrays of a small fixed size. This is done by logically computing multiple invocations of the kernel function in lock-step. The result is then fed into either the C++ or the LLVM back end, where the array operations are translated to actual SIMD instructions. Some modifications for the main loop are necessary in these back ends to be able to call the modified kernel function. In particular, the modified main loop needs to iterate over tuples of multiple data elements instead of single elements after the transformation.

The actual SIMD instruction set can be configured using a translation table (see next section). However, at the current stage, only an SSE3 table has been implemented. Section 5 describes the code transformation process from scalar to vector code, termed “code fusion” in this thesis.

4.5.2. C++ back end

The C++ back end performs a direct translation of the syntax tree that is given as the input to C++. Since the syntax of SLURP and C++ differs only for function declarations and type definitions, the largest part of the translation is quite simple. However, to make use of the arithmetic operators available in C++, the back end needs to translate back function calls into operators and obey the operator precedence rules of C++. (All operators are translated to function calls during the parsing process, as described in Section 4.2).

The result is a string or a text file containing standard C++ code with a function for each computational kernel. The text can then be fed into a C++ compiler, such as g++ [GRS] or the Microsoft C++ compiler, to generate the final machine code for any platform that is to be supported. The compile result for the example in Listing 4.3 is shown in Listing 4.4. The include file 'scl_builtins.h' contains C++ implementations of those built-in functions that have no implementation in 'core.scl' (See Section 3). All functions except for the main kernel function are defined in the 'slurp_filter' name space to avoid name collisions with external code. $PixelWriteCursor$ is a simple abstraction to access individual elements of an array. This abstraction can be used to transparently access contents of an array which is not stored as a continuous block of memory (e.g. this may not be possible for large arrays).

The function *kernel_test_generic* is generated by the compiler. It contains the main loop that loops over the output array and calls the kernel function for each output element. The return value of the kernel function is then written to the output array. For the case where the C++ back end is used as part of the C++/SIMD back end (see Figure 4.3), this function will have the SIMD instruction set architecture as a suffix in its identifier (e.g. 'sse3').

To facilitate code reuse, the back end works using a set of translation tables. These tables can be filled with target language specific information to allow translation to any C-like language. This is used in the GLSL back end, which uses the C++ back end with GLSL specific type and function translations. The maps are in particular:

Operator map The operator map maps functions to operators. An operator is defines as a symbol, a precedence value and a fixness (prefix, infix or suffix). This table is used to invert the transformation from operators to functions in the parser. It is needed because different languages may define some operators differently (e.g. matrix-matrix multiplication is defined as element-wise for HLSL, a regular matrix multiplication for GLSL and possibly undefined in C++).

Function map Allows to map functions to functions with the same type signature but with a different name. This map is used for the GLSL back end to translate some function names (e.g. the *lerp* function for linear interpolation is called *mix* in GLSL).

Type map The type map translates type names to different type names. Examples are vector and matrix types that can be different: vec2i in GLSL corresponds to int2 in OpenCL and SLURP.

```
// generated by SCL - do not modify!
#include "scl_builtins.h"

namespace slurp_filter {


  static inline float test(const float2& pos)
  {
    float z = pos[0] + pos[1];
    z = z + 5.000000e-01f;
    if( pos[0] >= 5.000000e-01f )
      {
        z = z - 5.000000e-01f;
      }
    return z * 5.000000e-01f;
  }

} // namespace slurp_filter

void kernel_test_generic(slurp_filter::PixelWriteCursor dest, slurp_filter::Recti rect)
{
  using namespace slurp_filter;

  for( int y = rect.mins[1]; y < rect.maxs[1]; y++ ){
    dest.moveTo(0, y);
    float2 pos;
    pos[1] = (float)y;
    for( int x = rect.mins[0]; x < rect.maxs[0]; x++ ){
      pos[0] = (float)x;
      dest.r() = slurp_filter::test(pos);
      dest.stepRight();
    }
  }
}
```

Listing 4.4: Example kernel compiled to C++

Constructor map Constructors have a very different syntax in different languages. Especially the construction of arrays varies: Some languages support array literals of the form "[1, 2, 3]" (D) or "{1, 2, 3}" (C/C++), others require a C++-like constructor call: "int3(1, 2, 3)" (GLSL/OpenCL). The constructor map translates values that have an array type to an expression that has this value as its result.

SIMD field map Inside the SLURP language, a SIMD tuple is defined as a struct with one field named v. The v field is defined as an array of scalars with the length of the SIMD with of target instruction set. In contrast, the compiler intrinsics used in C++ use a union of multiple such arrays for all supported SIMD types (e.g. $4 \times float32$, $4 \times int32$, $8 \times int16$ and $16 \times int8$). Each of these fields is named differently. The SIMD field map allows to map the v field to a field name for the corresponding C++ type.

4.5.3. GLSL back end

The GLSL back end is targeting the OpenGL shading language, which will in turn be compiled at application run-time by an OpenGL driver for the GPU. The compiled shader is then executed as a fragment shader, generating a texture that contains the results of the kernel computation. Several other languages such as HLSL as used by Direct3D or OpenCL exist as possible alternatives for GPU computations. Their syntax is in general very similar and analogous back ends can be implemented with low effort.

The GLSL back end derives from the C++ back end and overrides those functions that are responsible for generating type definitions and function headers and fills the translation tables (as described in Section 4.5.2) with entries matching GLSL. Apart from some boilerplate code which is necessary to connect the compute kernel to the GLSL shader entry point, all other code is syntactically identical with C++, so that the main part of the C++ back end can be reused without further modification.

The example kernel translated to GLSL is shown in Listing 4.5. The boilerplate code consists of declarations for some global input variables and the main function for the fragment shader. The main function then simply calls the kernel function and writes the result to the output variable corresponding to the current pixel in the destination texture.

Running the kernel consists of the following steps in OpenGL:

1. Create the shader using the output GLSL code
2. Create a texture for each input array and upload the array data
3. Create a texture for the output array
4. Create a framebuffer object and bind the output texture
5. Bind the framebuffer, the shader and the input textures
6. Draw a rectangle covering the full framebuffer
7. Read back the data from the output texture

The recommended approach is to keep the data as a texture as long as possible and only read it back as necessary. This means steps 2 and 7 only have to be done at times when the data is needed in system RAM (or needs to be stored on disk). Also, steps 1 and 4 have to be done only once, leaving steps 5, 6 and possibly step 3 to be done for each execution.

4.5.4. LLVM back end

The last back end is based on the LLVM compiler framework. The syntax tree is translated into LLVM's intermediate representation, which is an assembler-like language with a static-single-assignment (SSA) syntax (see Section 6.2). This intermediate representation is then compiled by the LLVM library at run-time and can be executed directly.

```
#version 140

float test(in vec2 pos)
{
  float z = pos[0] + pos[1];
  z = z + 5.000000e-01;
  if( pos[0] >= 5.000000e-01 )
    {
      z = z - 5.000000e-01;
    }
  return z * 5.000000e-01;
}

varying vec2 _position;

void main()
{
  gl_FragColor.r = test(_position);
}
```

Listing 4.5: Example kernel compiled to GLSL

This allows to use the CPU for performing kernel computations without needing another compiler or modifying the build chain for the project, as is the case for the C++ back end. The LLVM back end has a dedicated description in Section 6.

5. SIMD Code Generation

The SIMD back ends generate vectorized code that is aimed at performing each operation of a kernel in lock-step for four neighboring output data elements. A set of compiler defined types is used to represent SIMD tuples in a hardware independent way (see Listing 5.1). Only the word size of these tuples has to match the word size of the target instruction set. Finally, the generic operations are translated to either compiler intrinsics[1] embedded in C++ code (see Section 7.3), or by using the generic vector operations offered by LLVM [LA04], depending on which target back end is used.

The main SIMD instruction sets supported by the recent generations of x86 CPUs are MMX and Streaming SIMD Extensions (SSE). MMX provides simultaneous operations on either eight 8-bit integers, four 16-bit integers, two 32-bit integers or one 64-bit integer. Since most mathematical algorithms (e.g. physical simulations, equation system solving, image processing) require or are more easily expressed using floating-point operations, MMX is not sufficient as a target for the SLURP language. SSE on the other hand allows the computation of four simultaneous 32-bit floating point operations, which is sufficient precision for many algorithms. Later versions of the SSE instruction set allow two 64-bit integers to be processed at the same time, among other things. However, since the emphasis in this project is on speed, it uses only 32-bit floating-point numbers.

Another SIMD instruction set, which is only available on Intel's latest CPU architectures, is the Advanced Vector Extensions (AVX) set. This instruction set doubles the word size compared to SSE and is able to compute eight 32-bit floating-point operations at once.

Figure 5.1 shows the execution of a kernel on the CPU using a scalar execution model. For each data element in the output array, the kernel function is executed. After it has finished, the next element is processed. Figure 5.2 shows the same kernel using SIMD instructions to execute multiple output elements in lock-step. In this example, four input elements are processed in parallel and four output elements are stored at once. After that, the next four elements will be processed. Using this concept, the computation can in theory be sped up by a factor of four by using all of the CPU's arithmetic computation units instead of only one.

Complications arise if the kernel function gets more complex and contains conditional control flow. The next sections describe the process of transforming a program to the SIMD form and how these complications are handled.

[1] `http://en.wikipedia.org/wiki/Intrinsic_function`

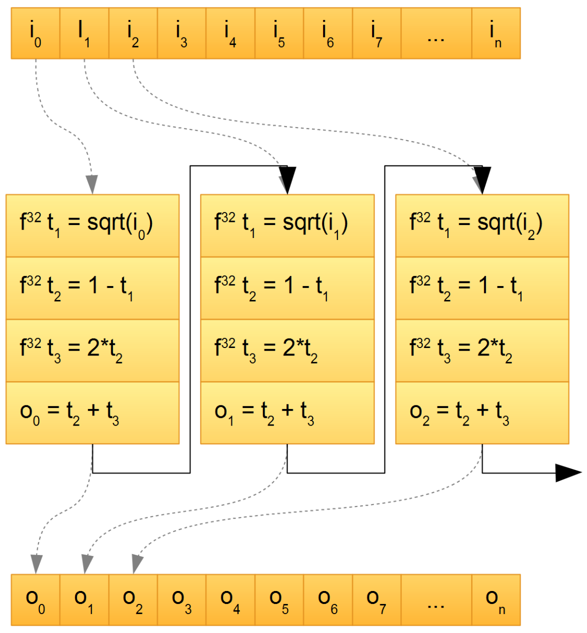


Figure 5.1.: Scalar execution of a stream program

5.1. Code fusion

The term "code fusion" is used throughout this thesis to describe the process of transforming a linear program into vectorized form, where all data types are replaced by vectorized versions of the type. Special handling is required for conditional program flow. The result of the fusion is a program that computes a fixed number output data elements in parallel, while at the same time retaining the semantics of the original program. The fusion process happens on all levels of the syntax tree – types, functions, statements, expressions and values all have to be transformed.

Fusing a type works by replacing all primitive data elements in a type (*bool*, *int*, *float*) by their fused SIMD equivalent, which is a compiler defined struct containing an array of the primitive type with the size of a SIMD word of the target instruction set (see Listing 5.1). This struct is later replaced by the SIMD type used in C++ (___m128 or ___m256) or by a vector type in the case of the LLVM back end. The fused type is then added as a type alias suffixed with an "_f". For an example of some types with their fused equivalents see Listing 5.2.

The code fusion of a program starts by fusing all functions in the program and adding a definition of the fused function with an "_f" suffix. The fused function has all parameters and the return type replaced by the fused equivalents. Then, all statements contained in the function are fused. If a function with the same signature is already defined, the existing function is used instead of performing the code fusion. This allows to specify hand optimized fused versions of certain functions.

For most of the statements, fusion means simply replacing any variable references by the new fused variables/parameters and fusing any expression or type contained in the

```
// type definitions of SIMD tuples for an SIMD architecture
// with 4 elements per word (e.g. SSE):
alias bool_f = struct { bool[4] v; }
alias int_f = struct { int[4] v; }
alias float_f = struct { float[4] v; }

// the equivalent definitions for AVX are:
alias bool_f = struct { bool[8] v; }
alias int_f = struct { int[8] v; }
alias float_f = struct { float[8] v; }
```

Listing 5.1: Compiler defined types used to represent SIMD tuples

```
// original types
alias myvec = float[2];
alias mystruct = struct {
  int[4] u
  myvec v;
  float w;
};

// fused types
alias myvec_f = float_f[2];
alias mystruct_f = struct {
  int_f[4] u;
  myvec_f v;
  float_f w;
};

// Compiler internal types:
//   alias float_f = struct { float[4] v; };
//   alias int_f = struct { int[4] v; };
```

Listing 5.2: Type definitions along with their fused equivalents

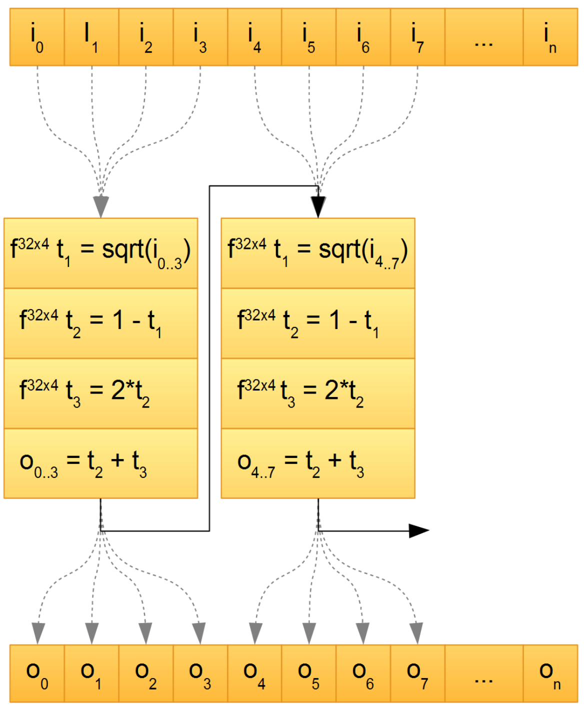


Figure 5.2.: Execution of a stream program using SIMD instructions

statement. Special handling is required for conditional statements and loops; these are covered separately in Sections 5.4 and 5.5.

Expressions are fused by first recursively fusing all sub-expressions until either a function call expression or an array index expression is reached. For function call expressions (note that operators are also mapped to functions during the parsing process) the function that is called is replaced by the fused function. Array accesses require some special handling because the array indices may differ across the SIMD elements. Section 5.3.1 goes into more detail about this.

After this process has finished, the resulting function will now generate the same result as the original scalar function, just for multiple values in parallel. The proof for the semantic equality is simple because of the functional nature of the stream program. It has not been done here explicitly though because the proof is expected to be quite long if all cases are handled.

Listing 5.4 shows a simple function without loops and conditional control flow along with its fused equivalent. Note that the multiplication in the first line of the function is translated to a call to $opMul$ which in turn results in a call to $opMul_f$ in the fused code. $opMul_f$ is then translated to an SIMD instruction in the target back end (e.g. the C++ back end). Also note how the $uniform$ parameter a is not affected by the fusion process. Since the compiler knows that a does not change across multiple invocations of the function, it is sufficient to keep one scalar version of a for all computations that are done in parallel in the fused code.

```
// original function (compiler defined):
function float sin(float x);

// fused function:
function float_f sin_f(float_f x)
{
  float_f ret;
  ret.v[0] = sin(x.v[0]);
  ret.v[1] = sin(x.v[1]);
  ret.v[2] = sin(x.v[2]);
  ret.v[3] = sin(x.v[3]);
  return ret;
}
```

Listing 5.3: The result of code fusion on a built-in function

5.1.1. Built-in functions

Built-in functions for which there is only a run-time implementation of the scalar function available but no implementation is known at compile-time have to be treated in a special way. These functions are transformed to a function that takes SIMD tuples as input and also outputs the result as an SIMD tuple. However, inside it will invoke the original, scalar version of the function for each SIMD element separately and sequentially. It thus preserves the semantics, albeit without introducing any parallelism. However, later in the process, the target back end can replace the whole function with a single SIMD instruction in many cases. For example, this is the case for all basic arithmetic operations such as multiplication or addition.

Conceptually, this operation is equivalent to transforming a hypothetic fused version of the function back to a scalar version while keeping the types of the parameters and variables fused (equals code splicing, see Section 5.2). The result is a function that computes each SIMD component of the result separately in a scalar fashion. Listing 5.3 shows an example of such a function.

5.2. Code splicing

Code splicing is in some sense the reverse operation of code fusion. It is needed in parts where the parallel program flow of the fused program diverges. In these cases the individual data elements again have to be computed in a scalar fashion, just as in the original code. However, since the variables inside a fused program block have a fused type, the original code cannot be used as is.

Instead, what code splicing does is to take the fused code and transform it so that only a certain SIMD components is computed. The fused variables will be referenced, but only one fixed component will be read/written. There will then typically be a loop inserted around the spliced code to compute components consecutively.

The example in Listing 5.4 shows how this applies to the fused variable x. Inside if the *test_f_spliced* function, x first has to be decomposed so that only the SIMD

```
// original function
function float3 test(float3 x, uniform float a)
{
  float3 y = a * x;
  y[0] = -y[1];
  return exp(y);
}

// fused version of test()
function float3_f test_f(float3_f x, uniform float a)
{
  float3_f y = opMul_f(float_f(a), x);
  y[0] = -y[1];
  return exp_f(y);
}

// spliced version of test_f(), computing just one SIMD component of the result.
function float3 test_f_spliced(float3_f x, uniform float a, int simd_idx)
{
  float3 y = a * float3(x[0].v[simd_idx], x[1].v[simd_idx], x[2].v[simd_idx]);
  y[0] = -y[1];
  return exp(y);
}
```

Listing 5.4: A simple function along with fused and a spliced versions

component specified by $simd_idx$ is accessed. The rest of the code equals the code of the original, scalar function.

5.3. Optimization

Apart from the optimizations already mentioned in Section 4.4, there are a number of additional optimizations for the code fusion and code splicing transformations that can be critical for the performance of programs with specific (but common) patterns.

5.3.1. Constant unfusion for array access and SSE register filling

Array accesses with SIMD tuples used to index the array normally require a gather operation, where first the data elements are collected for the indices corresponding to each SIMD element. For complex array element types the data will also have to be transposed into fused form (e.g. $4 \times float4$ to $1 \times float4\_f$). Figure 5.3 shows such a generic index read access. The inverse approach applies to write accesses on an array. A scatter operation is required here to store the results for each SIMD component. Gather/scatter operations are not directly supported in the current SIMD instruction sets which means that the operation has to be broken up into a sequence of scalar load/store operations. This can impose a considerable performance overhead in the fused code.

However, if it is known at compile-time that the array indices are the same for all SIMD components, a single array access can be made by accessing just the index of

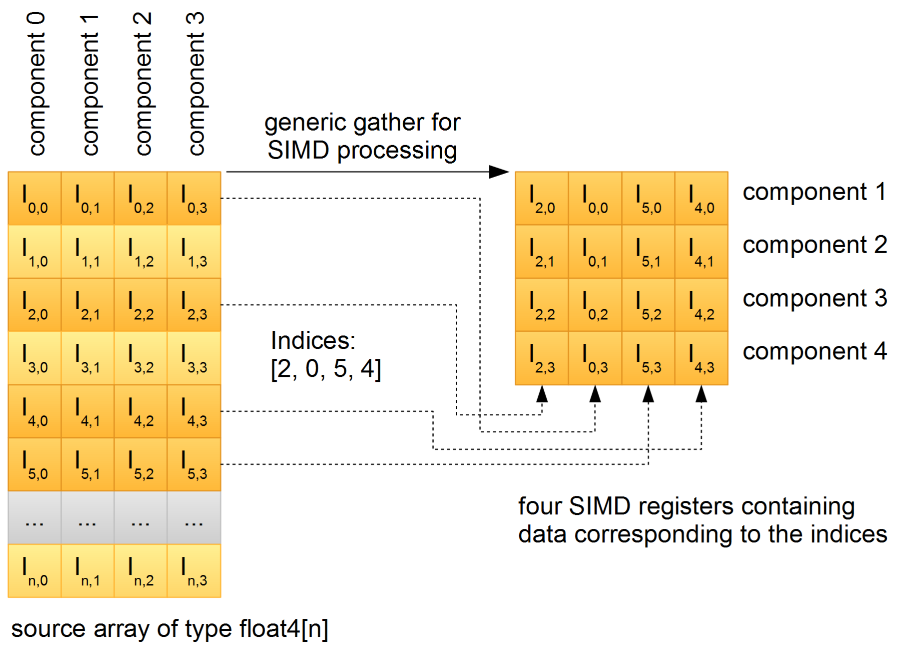

component 0
component 1
component 2
component 3
generic gather for SIMD processing
Indices: [2, 0, 5, 4]
component 1
component 2
component 3
component 4
four SIMD registers containing data corresponding to the indices
source array of type float4[n]

Figure 5.3.: Generic array access from SIMD code usind a component-wise gather operation

one SIMD element. The result is then replicated among the result SIMD register by using an efficient operation (e.g. the *_mm_set1_ps* intrinsic in case of SSE output). See Figure 5.4

The *uniform* type attribute guarantees that the value of a variable with the *uniform* attribute is the same for all invocations of a kernel. Thus, all array indices that are typed *uniform* are subject to this optimization. Since all constant values, such as numeric literals, are automatically uniform, these are also optimized.

5.3.2. Consecutive sampling detection

Very often, simple filter kernels map their input data elements one-to-one to the corresponding output elements at the same coordinates in the output array. For these filters the data sampling operation can be optimized by taking n consecutive data elements from the input array (where n equals the SIMD word size) and performing a matrix transpose to transform them to the fused form. Figure 5.5 shows such a transposition for an input array of $float4$ elements on a machine with SIMD words of size four. The generic approach would be a gather operation as in Figure 5.3.

This pattern is detected by simply looking at the arguments of every *sample* call (see Section 3.4) in a kernel and checking if the coordinate parameter is given exactly as the current output coordinate. More refined detection could be done by allowing constant additions and subtractions or passing the output coordinate to other functions that again pass the value unmodified to the *sample* function. However, the most common case is

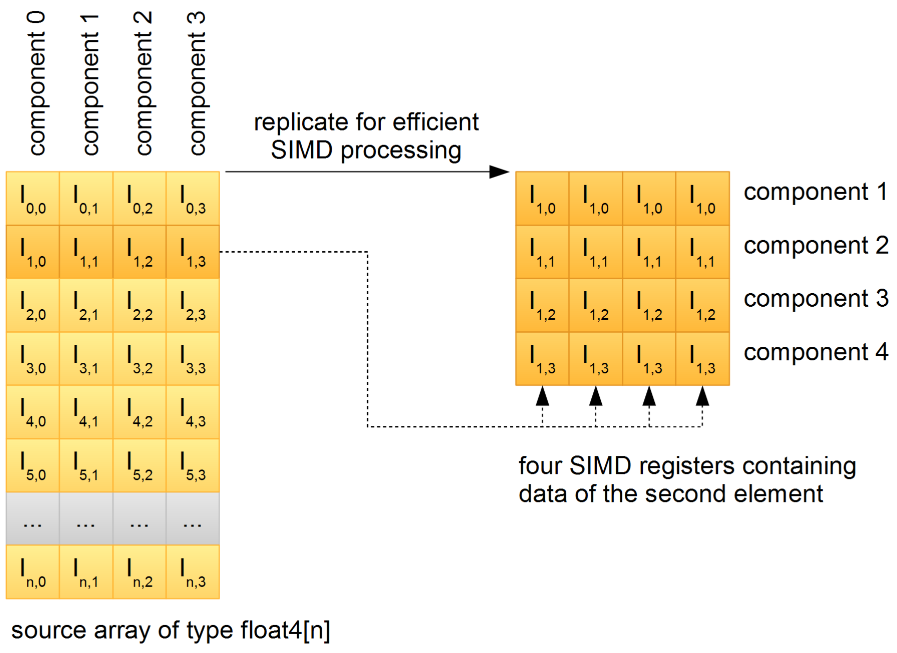

component 0
component 1
component 2
component 3
replicate for efficient SIMD processing
$I_{0,0}$ $I_{0,1}$ $I_{0,2}$ $I_{0,3}$
$I_{1,0}$ $I_{1,1}$ $I_{1,2}$ $I_{1,3}$
$I_{2,0}$ $I_{2,1}$ $I_{2,2}$ $I_{2,3}$
$I_{3,0}$ $I_{3,1}$ $I_{3,2}$ $I_{3,3}$
$I_{4,0}$ $I_{4,1}$ $I_{4,2}$ $I_{4,3}$
$I_{5,0}$ $I_{5,1}$ $I_{5,2}$ $I_{5,3}$
... ...
$I_{n,0}$ $I_{n,1}$ $I_{n,2}$ $I_{n,3}$
source array of type float4[n]
$I_{1,0}$ $I_{1,0}$ $I_{1,0}$ $I_{1,0}$ component 1
$I_{1,1}$ $I_{1,1}$ $I_{1,1}$ $I_{1,1}$ component 2
$I_{1,2}$ $I_{1,2}$ $I_{1,2}$ $I_{1,2}$ component 3
$I_{1,3}$ $I_{1,3}$ $I_{1,3}$ $I_{1,3}$ component 4
four SIMD registers containing data of the second element

Figure 5.4.: Simplified array access using replication in case of a uniform array index

to use exactly the same coordinate, so this was chosen as the only pattern for the first implementation.

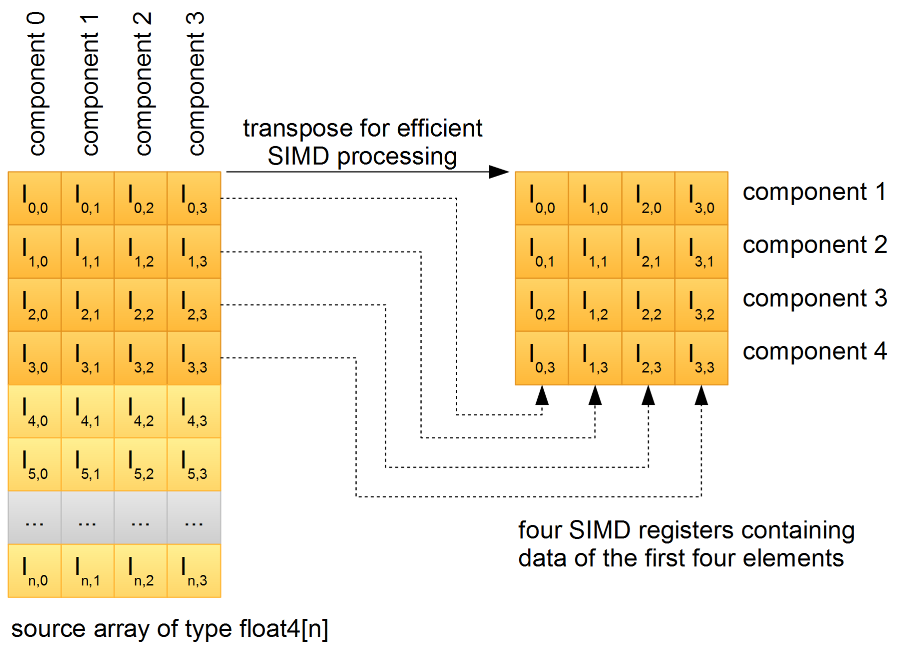
5.4. Conditional statements

Conditional statements create a problem for the code fusion process. If the result of the condition differs across neighboring data elements, the code paths differ, and it may become impossible to compute the result using the usual SIMD approach (See Figure 5.6). It now becomes necessary to fall back to scalar code and compute each data element on its own, depending on the corresponding outcome.

The general approach that is taken is as follows:

1. Compute the condition result for all SIMD components at once (fused code).
2. If the results are all equal, execute the true/false branch as normal fused code.
3. If the results differ, loop over all SIMD components, check the condition result each component and, depending on the result, execute the spliced code of the true/false branch corresponding to that SIMD component.

Listing 5.5 shows a small example with the compiled result in Listing 5.6. The result usually works acceptably fast because most algorithms have a relatively high spatial coherence (data elements that are near to each other have similar values and hence take similar code paths). If the generic case using spliced code needs to be executed anyway,



component 0
component 1
component 2
component 3
transpose for efficient SIMD processing
component 1
component 2
component 3
component 4
four SIMD registers containing data of the first four elements
source array of type float4[n]

Figure 5.5.: Simplified array access using a transpose operation in case of consecutive array indices

the C++ compiler can again use its auto-vectorization capabilities to speed up the code to the usual performance level of scalar code (except for the gather/scatter overhead that is introduced by the code splicing).

Another possibility to handle conditional branches is to always execute both the true and the false branch, record all results, and then select the right value for each SIMD component using the corresponding condition result. Code hand generated using this approach is shown in Listing 5.7. The compiler in its current state has no implementation of this type of conditional translation as it is only efficient if the conditional branches are short. Also, it requires temporary copies of all mutated state, which can be complex to implement in an optimal way. However, for short branches, it can actually be more efficient than the first approach, even if both branches are always computed.

In the future, the compiler could choose between both methods by some criterion, such as length of the branches. It could also execute the kernel on a test data set once for each method and choose the faster one. In the case of the LLVM back end it would even be possible to switch between the methods at run-time by employing a live benchmark while the kernel is executing on the actual data.

A special case are uniform conditions, that is, the condition uses only uniform or constant values or variables (as defined in Section 3.2) to compute its result. The conditional statement can be simplified in this case to compute the condition in a scalar way and to always use fused code for both branches. This is a small but important optimization for the cases where certain code-paths are enabled or disabled using a uniform parameter handed in to the kernel function. Uniform conditions that are also

```
kernel float4 conditionalKernel( float2 coord )
{
  float4 result;
  result[0] = 0.0;
  result[1] = 0.0;
  result[2] = 0.0;
  result[3] = 1.0;
  if( coord[0] > 100 ){
    result[0] = 1.0;
  } else {
    result[1] = 2.0;
  }
  return result;
}
```

Listing 5.5: Example kernel with a conditional

```
static inline float4_f conditionalKernel_f( vec2_f coord )
{
  float4_f result;
  result[0] = _mm_set1_ps(0.000000f);
  result[1] = _mm_set1_ps(0.000000f);
  result[2] = _mm_set1_ps(0.000000f);
  result[3] = _mm_set1_ps(1.000000f);
  {
    __m128 ___tmp_0 = _mm_cmpgt_ps(coord[0], _mm_set1_ps(100.000000f));
    int ___tmp_1 = 0;
    if( allTrue(___tmp_0) )
      result[0] = _mm_set1_ps(1.000000f);
    else if( allFalse(___tmp_0) )
      result[1] = _mm_set1_ps(2.000000f);
    else
      for( ; ___tmp_1 < 4; opInc(___tmp_1) )
        if( ___tmp_0.m128_i32[___tmp_1] )
          result[0].m128_f32[___tmp_1] = 1.000000f;
        else
          result[1].m128_f32[___tmp_1] = 2.000000f;
  }
  return result;
}
```

Listing 5.6: Example kernel with conditional translated to SIMD/C++

```
static inline float4_f conditionalKernel_f( vec2_f coord )
{
  float4_f result;
  result[0] = _mm_set1_ps(0.000000f);
  result[1] = _mm_set1_ps(0.000000f);
  result[2] = _mm_set1_ps(0.000000f);
  result[3] = _mm_set1_ps(1.000000f);
  {
    // copy all modified state into ___tmp_1
    float4_f ___tmp_1;
    ___tmp_1[0] = result[0];
    ___tmp_1[1] = result[1];
    // execute the 'true' branch
    ___tmp_1[0] = _mm_set1_ps(1.000000f);

    // copy all modified state into ___tmp_2
    float4_f ___tmp_2;
    ___tmp_2[0] = result[0];
    ___tmp_2[1] = result[1];
    // execute the 'false' branch
    ___tmp_2[1] = _mm_set1_ps(2.000000f);

    // evaluate the condition
    ___m128 ___tmp_0 = _mm_cmpgt_ps(coord[0], _mm_set1_ps(100.000000f));

    // select the components from ___tmp_1 and ___tmp_2 to store into result
    // note that this a predefined inline function using multiple
    // _mm_andnot_si128 intrinsics internally.
    result[0] = select_f(___tmp_0, ___tmp_1[0], ___tmp_2[0]);
    result[1] = select_f(___tmp_0, ___tmp_1[1], ___tmp_2[1]);
  }
  return result;
}
```

Listing 5.7: Example kernel with conditional translated to SIMD/C++ with delayed selection

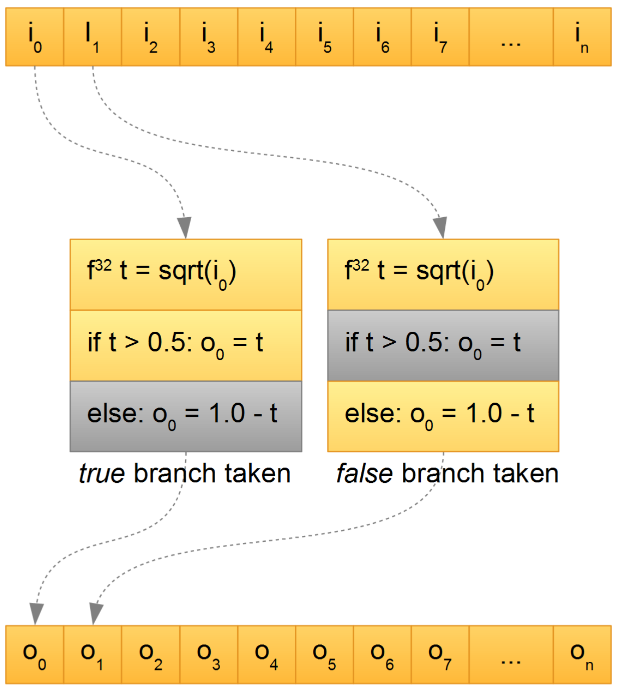


Figure 5.6.: Differing conditional control flow for two data elements

constant are completely replaced by the true or false branch by the dead code elimination logic at an earlier stage (Section 4.4.2).

5.5. Loop statements

Loops, as a generalization of conditional statements, have the same issue that the execution paths of neighboring data elements may diverge because the number of loop iterations may differ between neighboring data elements. A similar approach to the one used for conditional statements is used here to handle this case:

1. Initialize a condition mask variable to true for each SIMD component.
2. Compute the loop condition result for all components of the current SIMD group.
3. Perform a logical AND operation of the condition result with the condition mask and place the result in the condition mask.
4. If the condition mask is all true, execute the loop body as normal fused code.
5. If the condition mask is all false, exit the loop.
6. If the condition mask contains mixed results, loop over all SIMD components where the condition mask is still true and execute the spliced loop body for each of those components.

Note that ‘continue’ and ‘break’ statements are not currently supported, as they severely complicate the algorithm in the case of nested loops. Also, supporting these statements requires storing additional state and performing additional tests, which can impact performance. Support for these statements may be added in a later version along with a proper code analyzer to conditionally use the simple algorithm for the cases with no such statements. For the time being, the same effect can be achieved using boolean variables.

Loops with a uniform condition can be translated to a simple scalar loop with a fused loop body similar to the translation of uniform conditional statements. Note that it is a very common case to use loops with a constant number of iterations, so this is a very important optimization for many algorithms.

6. The LLVM Back End

The LLVM back end provides an additional CPU target for the SLURP compiler. It uses the LLVM compiler framework [LA04] to compile the SLURP program to machine code. It is able to do the full compilation at run-time using the virtual machine provided by LLVM. Together with the GLSL back end, which also compiles the code at application run-time, it allows to use the CPU and the GPU using run-time compiled code. This in turn allows dynamic code generation and fast changing and reloading of modified filter kernels. When doing filter development, this can be a significant productivity boost. The C++ back end, however, requires a separate invocation of the C++ compiler and linker tool chain. The application then has to be relinked and run again, resulting in long turnaround times and no direct possibility of doing dynamic kernel code generation.

An important property, which is shared with the CPU back end, is that the LLVM back end can be considered as a reliable target. The machine code output does not depend on a vendor driver that changes over time and the LLVM library is deployed together with the SLURP compiler and thus is always available.

The back end is at the same time the only back end that differs significantly from the C++ back end in that it compiles down to an assembler-like intermediate language instead of high-level C++ or GLSL code that is very similar to SLURP code.

6.1. The Low Level Virtual Machine

The LLVM (Low Level Virtual Machine) project [LA04] was initially started in 2000 as a research platform for just-in-time (JIT) compilation techniques at the University of Illinois. The project has grown since then to a full compiler framework with many front and back ends available, including a full C++ front end. Two modes of operation are available. First, by using one of the regular compiler back ends, code can be compiled down to machine code and linked into object or executable files for native execution. Second, LLVM supports two different virtual machine implementations. The code can either be interpreted or – and this was the primary purpose of the project – can be compiled into machine code on-the-fly and executed natively without having to write out an executable file. This mode is especially interesting for this thesis, because it allows for several interesting features.

One advantage of the JIT model is that it is possible to generate machine code for both types of hardware, the CPU and the GPU, at application run-time. This means that the compiler can be used as a library instead of a standalone executable and the kernel functions can be developed and recompiled while the application is running. This allows great savings in development time when kernels are being debugged or tuned, and advanced applications using generated kernel code become possible.

Compiling the code just-in-time also opens up the possibility to perform architecture-specific optimizations for the target machine. Typical compilers will compile for a certain architecture (e.g. Intel 80586) and use a certain extended feature set (e.g. SSE) but can only optimize for a certain type of processor. Since different processors can vary widely in their performance characteristics (in-order vs. out-of-order execution, pipeline depth, caches etc.), the processor-specific optimization is always a trade-off between the targeted CPUs. When using JIT compilation, however, the code can be optimized specifically for the current CPU to obtain optimal performance.

6.2. SSA representation

LLVM uses an intermediate representation known as static-single-assignment (SSA) [CFR+89]. This representation models the data flow of the program in a functional style, where each computation yields a new symbolic constant that can be used in later computation but can never be modified. This kind of intermediate representation allows for efficient formulation of many optimization and control flow analysis algorithms, as the program data flow is readily available as a cycle-free graph.

Special handling is required for places where the control flow diverges by means of conditional branching. In these cases, to be able to represent alternative values for a certain variable that is changed differently in each code path, a special kind of node is defined in the SSA graph, the so-called phi function. A phi function node is a node that selects a value from a set of inputs depending on the control-flow branch that lead to the phi node. Specifically, it has exactly one value assigned to it for every possible branch or jump instruction that jumps to the place where the phi node is defined. Listing 6.1 and Listing 6.2 show a simple example program along with its SSA representation, including a phi node.

The main purpose of the LLVM back end of the SLURP compiler is to transform the syntax tree representation with its procedural variable assignments into SSA form. However, since LLVM also allows memory load and store operations, the back end does not have to explicitly insert phi nodes in loops and after conditional statements. Instead, it emits instructions to store each variable that has been modified in a conditional branch to memory. The next block will then have to load the value back from memory before using it. LLVM then has an optimization pass built-in that recognizes the load/store instructions and converts them into phi nodes if possible. This approach substantially simplifies the code generator and moves the phi node generation to a stage where it can be done efficiently because all the necessary information is available. In particular, all blocks that may lead to a different value for a variable are known only after the code generator has generated the whole function – defining the phi variable may not even be possible at the point where it is first used. An example for this case is a loop, where the value of a variable may be read and changed inside of the loop. The phi node has to be defined at the beginning of the loop, when the loop body has not yet been generated.

```
function float test(float x, float y)
{
  float z = x + y;
  z = z + 0.5;

  if( x >= 0.5 ){
    z = z - 0.5;
  }

  return z * 0.5;
}
```

Listing 6.1: A simple function for SSA decomposition

```
define float @test(float %x, float %y) alignstack(16) {
EntryBlock:
  %z1 = fadd float %x, %y
  %z2 = fadd float %z1, 5.000000e-01
  %0 = fcmp oge float %x, 5.000000e-01
  br i1 %0, label %trueBranch, label %tail

trueBranch:                                        ; preds = %EntryBlock
  %z3 = fsub float %z2, 5.000000e-01
  br label %tail

tail:                                              ; preds = %trueBranch, %EntryBlock
  %z.0 = phi float [ %z3, %trueBranch ], [ %z2, %EntryBlock ]
  %1 = fmul float %z.0, 5.000000e-01
  ret float %1
}
```

Listing 6.2: The simple function transformed into LLVM SSA

6.3. Virtual machine interface

Once the program is handed over to LLVM in the form of an SSA immediate representation, a so-called execution engine is used to execute the code. The execution engine can either be an interpreter, which is available on all platforms, or a JIT compiler specific to the current platform. In both cases, a simple API is used to feed the program with the input data in the form of primitive values or pointers to complex data. In the case of pointers, the virtual machine works directly on the memory pointed to – which means that there is no memory isolation and the compiler has to make sure that there are no potentially invalid memory accesses in the intermediate code if security is an issue.

6.4. SIMD code generation

Vectorized code is handled in a generic way inside of the LLVM intermediate language. Most operations can work either on scalar data or on vectors of arbitrary but fixed size. Such operations are then transformed into SIMD instructions when the program is compiled down to machine code. The LLVM back end of SLURP supports vectorized code on two levels. On the first level, all vector data types (e.g. $float4$) are also translated to vectors in the LLVM representation. This allows the LLVM back end to generate efficient SIMD code for the directly compiled program without a need for auto-vectorization.

Secondly, the code fusion transformation that is done in the C++/SIMD back end (Section 5) can also be applied to the LLVM back end. Just like in the C++ version, instead of computing one data element at a time, a tuple of pixels will then be computed in parallel by single SIMD instructions.

7. Discussion

7.1. The Auto-Vectorization approach compared to code fusion

The approaches taken for vectorization in almost all compiler implementations for imperative languages are either explicit vectorization using compiler intrinsics [Int06], special loop statements [DM98], or array operations [1] and similar means. Another main approach is called auto-vectorization [AK87] where the compiler tries to turn inner loops into SIMD operations. Auto-vectorization is found in many major compiler implementations, such as GCC, LLVM, the Microsoft C++ compiler and the Intel compilers. Early research in the area of automatic parallelization of linear programs was done for Fortran in 1987 [AK87] to be able to use the vector capabilities of the Cray super computer at that time. Similar approaches are now used to target the MMX, SSE and AVX instructions of modern x86 CPUs.

Auto-vectorization tries to detect certain patterns in the program that can be replaced by SIMD instructions. Obvious examples include loops with simple loop bodies. Such loops can be broken up into chunks of the SIMD word size (e.g. four in the case of SSE) and then replaced by SIMD instructions. Listing 7.1 shows a simplified example. Even this example requires considerable work in the optimizer to get to the final vectorized result. The compiler first has to make sure that the loop body does not have any side effects, where one iteration influences the execution of later iterations; note that this can be non-trivial in the presence of data aliasing. Then it has to match the unrolled loop contents against some known patterns with vectorizable code. Also, the vectorized result shown in the example code is not even necessarily correct because the ___m128 data type has to be memory aligned at a 128-bit boundary. Additional code has to be added to handle possible misalignments of the input arrays. Alternatively, SSE has load and store instructions which are slower but are allowed to access non-aligned memory.

A thorough overview of the topic of vectorization can be found in [AK01]. The topic is complex, and the fact that there are only a few compilers on the market that are able to do meaningful auto-vectorization gives a hint about this complexity.

The obvious advantage of auto-vectorization is that the program developer does not need to know anything about vectorization and still may get benefits. In reality, however, even with the best compilers it is often vital to understand how the compiler detects vectorizable code and how to express the code in a vectorizable way to obtain optimal

[1] D array operations: `http://d-programming-language.org/arrays.html#array-operations`

```
// The original function taking the square root
// for every element of an array
void sqrtArray(float* dst, float* src)
{
  for( int i = 0; i < 64; i++ )
    dst[i] = sqrt(src[i]);
}

// Step one is to unroll the loop to multiples
// of the SIMD vector size
void sqrtArrayUnrolled(float* dst, float* src)
{
  for( int i = 0; i < 64; i += 4 ){
    dst[i+0] = sqrt(src[i+0]);
    dst[i+1] = sqrt(src[i+1]);
    dst[i+2] = sqrt(src[i+2]);
    dst[i+3] = sqrt(src[i+3]);
  }
}

// Finally, the four unrolled expressions can be replaced
// by SIMD instructions. In this case, SSE intrinsics
// are used.
void sqrtArrayVectorized(float* dst, float* src)
{
  for( int i = 0; i < 64; i += 4 )
    *(__m128*)&dst[i] = _mm_sqrt_ps(*(__m128*)&src[i]);
}
```

Listing 7.1: Code snippet along with a possible auto-vectorization result

results. Another problem is that a lot of code is not vectorizable at all with such a simple local transformation.

The approach taken for this thesis is based on the concept of loops with loop bodies that have no side effects apart from the output that each iteration computes (the stream data model). The loop is generated by the compiler itself, and the developer only writes the loop body, which is constrained to return the output data for a given data index without any side effects. The compiler will also make sure that the input and output data arrays are properly aligned in memory. These conditions permit the complex code transformation described in Section 5.1 without aliasing or alignment issues.

The big advantage is that there are no patterns to match in some sequential unrolled loop code but that every single instruction of the original code can now be conceptually transformed into an SIMD instruction. This guarantees full usage of the SIMD units as long as the operations are supported by the SIMD instruction set and as long as no dynamic conditionals are used.

In case of conditionals or loops where the code has to be spliced and executed sequentially (see Section 5.2), the compiler of the target language (C++) can again use its auto-vectorization capabilities to optimize this sequential code. So in essence, by using the stream data computation model, it is possible to provide transformed code that has a much better starting point regarding performance than the equivalent scalar code. The target compiler can then optimize this code further.

7.2. Using D for the implementation

The compiler implementing the SLURP language and the compilation concepts of this thesis is written in the D programming language [Ale11]. The language development was initially developed by Walter Bright of Digital Mars in 1999 as the "Mars" programming language and was later renamed to "D". In 2007, the second version (D2) was started, with an emphasis on a powerful type system that is strict enough to allow to actually make static inferences and verifications. The result is that a lot of programming errors are detected at compile-time instead of later at run-time. Today D2 has a well defined specification[2] and is in its final stages where the feature set is fixed and the development focus is on bug fixing. Several compiler implementations exist, including the reference compiler based on the Digital Mars back end, a version based on GCC and one based on LLVM.

The most important requirement for the implementation language was link compatibility with C and the ability to generate shared libraries (DLLs). D allows this in addition to limited link compatibility with C++. Together with its other properties, this made D a very adequate choice as the implementation language.

The main goal of D is to provide a systems language, which means that it is compiled down to machine code and that it provides all the low-level facilities which are necessary to access the hardware directly (e.g. for writing drivers or operating system kernels). At

[2]D language reference: `http://d-programming-language.org/lex.html`

the same time it tries to stay close to the C++ syntax in many places but at the same time improve on C++ by removing a lot of historically grown dead weight that causes the language and compiler implementations to be overly complex. It also adds several features that help in terms of programming productivity, compile-time computations and verifiability.

One notable feature is the support of array slices[3]. In D, every array can be indexed using a pair of integers and then yields a view on the corresponding sub-range of the array. String processing can greatly benefit from this feature, as the memory use and cost for memory allocations is much lower than for the typical string implementations in C++. The implementation done for this thesis uses array slices to avoid allocating any memory for identifiers used in the input programs.

A garbage collector, which is used by default in D, simplifies the code in the compiler and helps to improve performance by reducing the amount of memory management during the compilation. Also it avoids memory leaks, which is especially important when the compiler is used as a library for run-time compilation. Finally, it makes sure that no dangling pointers exist, which can lead to hard-to-track crashes and and possibly dangerous misbehaviors. D additionally supports a subset known as SafeD. This subset allows only operations which are safe memory-wise. In particular, it is not allowed to use pointers or unsafe casting operations. Together this makes sure that the code is not vulnerable to typical security attacks using buffer overruns or similar program misbehavior.

Source code is organized in packages and modules, similar to languages such as Java and C#. D has no header files as they are used in C/C++. This has productivity advantages because there is less redundant code to write and at the same time it allows very fast compilation times. The D compiler processes all source code files at once and therefore avoids the most important reason for slow compile times of C/C++ code, the repeated loading and parsing of header files.

The syntax of D is simplified compared to C++ and leads to shorter and better readable code. Powerful compile-time meta programming features can be used to shorten the code further. This is used to build some tables in the compiler, where the signature of a list of D functions is automatically translated to the corresponding run-time representation by using compile-time reflection.

7.3. Choosing intrinsics as the SIMD target

The C++/SIMD back end generates C++ code with embedded calls to compiler intrinsic functions ("intrinsics") matching the SIMD instructions of the target instruction set. The obvious alternatives would be to use (inline) assembler or machine code as the target. However, there are several advantages to using C++ intrinsics instead:

- **Additional optimization**: The main advantage is that the C++ compiler usually has a powerful optimizer with machine-specific optimizations that can benefit

[3]D array slicing: `http://d-programming-language.org/arrays.html#slicing`

the end result. The compiler is free to perform instruction reordering and SIMD register allocation in a way that is advantageous for the target machine with intrinsics. Inline assembler forces the compiler to disable optimizations around the assembler block.

- **Implementation complexity**: Several low-level compiler routines are not needed in the back end that would be needed if assembler or machine code were targeted. Examples are register allocation, loop-to-jump transformations, jump label management, function calls and stack management. In the case of machine code, the full instruction set of each CPU architecture would need to be written and maintained.

- **Platform independence**: Since portable ISO C++ code is generated that can be compiled on any compiler which supports the target SIMD intrinsics, almost any machine and operating system is implicitly supported.

- **Limited architecture independence**: SSE and AVX intrinsics have the advantage over assembly code of not being specific to 32-bit or 64-bit x86 architectures. Also, the IA-64 architecture (Intel Itanium processor) supports SSE intrinsics, but only supports a subset of the SSE functionality present on x86 CPUs and has a different instruction encoding. Intrinsics provide a convenient abstraction to avoid having to account for this in the compiler.

The drawback of this approach is that a second compiler is needed to process the code. But since this compiler is needed anyway in most cases to compile the host application that uses the filter kernel, this is not a drawback in practice. The platform independence that is gained outweighs this disadvantage by far.

8. Measurements

To obtain a comprehensive understanding of the performance characteristics of the different back ends, a number of different kernels were run. Each kernel stresses a certain part of the processing framework so that differentiated conclusions can be made. The data is computed in four chunks, which are either computed sequentially or in parallel depending on the benchmark configuration.

The test platform is a laptop with an Intel Core i7 dual core CPU with Hyper-threading (simultaneous multi-threading, SMT) [MBH+02], clocked from 2.67 GHz up to 3.06 GHz using Turbo Boost, which allows dynamic over-clocking of the CPU when only some cores are used [CJA+09]. The memory consists of 4 GiB of DDR3 memory in dual-channel mode. The OpenGL/GLSL back end runs on an NVIDIA GeForce GT 330M. Certain CPU features, namely Hyper-threading and Turbo Boost, skew the results in favor of the single-threaded versions. But since these features are becoming mainstream across the major CPU vendors, the results are still a realistic measurement.

Five output types are generated. The first type is a direct translation of the kernel to C++ without explicit SIMD instructions and without multi-threading. The second output is the fused version with explicit SIMD usage but still without multi-threading. Then, the first two outputs are computed again but using 4 threads to split up the work. And finally, the kernel is compiled as GLSL.

Each of the CPU outputs is compiled using four different compilers. The C++ output is processed by the Visual Studio 2010 C++ compiler[1], once generating a 32-bit executable and once a 64-bit one. GCC 4.5.2[2] also processes the C++ output and compiles a 32-bit executable. LLVM 2.9 is targeted by the LLVM back end and will generate 32-bit machine code in memory for direct execution.

The GLSL output is benchmarked in two different modes. The first mode ("strict") computes the kernel result sequentially for each of the four input chunks and reads back each result directly. The second mode will only read back the result of the last chunk set that was computed. This mode was added to get a more realistic timing for environments where the result data stays in GPU memory as long as possible, which is recommended for intermediate results when working on the GPU. The overhead of reading back data from the GPU is still noticeable in the mode, although the PCIe bus of modern systems is much better in this regard than the older AGP bus. The loop overhead benchmark shows how much impact the additional read-back operations in the "strict" version have.

All tests were done four times, and the best result was chosen for each configuration. The GCC results for the SIMD version are strikingly worse than the rest of the config-

[1]Command line: cl /O2 /Oi /Ob2 /Ot /arch:SSE2 /fp:fast [input files]

[2]Command line: g++ -O3 -msse3 -ffast-math [input files]

urations because GCC does not perform function inlining for this configuration. The reason for this is unknown.

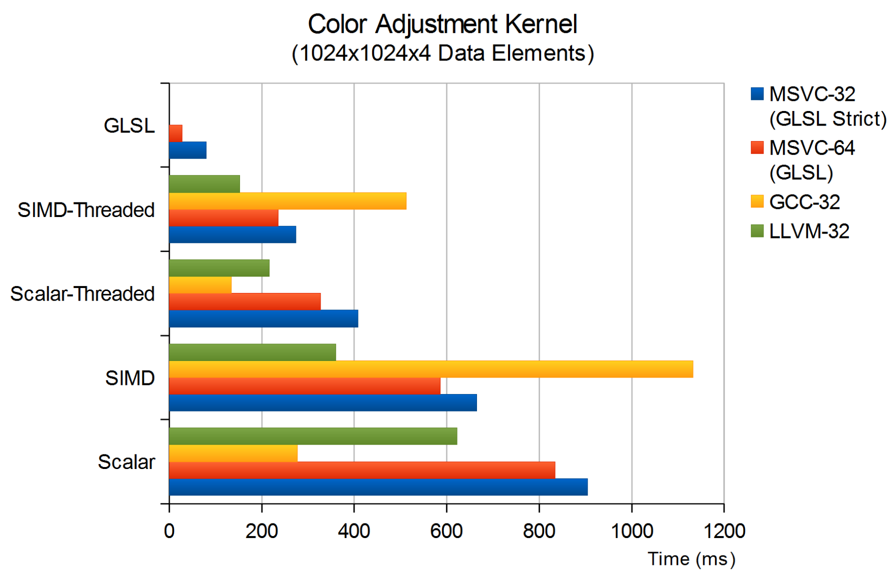


Figure 8.1.: Benchmark results for the color adjustment kernel

Arithmetic performance The first kernel (color adjustment kernel, see Listing C.1) performs a relatively complex calculation separately on each data element (nearly 10^3 arithmetic operations per data element) with some conditional statements and only one sampling operation of the input data array. This test provides a good measure of how much can be gained by the code fusion process for a typical computation-heavy kernel that operates on vectors (in this case RGBA pixels). The results are shown in Figure 8.1.

Both the LLVM back end and the C++ output compiled with Visual Studio have a performance gain of about 73 % when comparing the fused version against the scalar version. The theoretical gain of 300 % (a factor of four) is not reached because many of the operations done in the kernel work on 4-component vectors. The compilers are able to translate these operations to SIMD instructions and can already gain a lot even without code fusion. Multi-threading offers a further gain of about 135 % for all back ends, which is in the usual range for a dual-core CPU with Hyper-threading.

The GLSL versions are faster by a factor of 2 to 5 when compared to the multi-threaded SIMD version of the LLVM back end. This is approximately the theoretical ratio between the floating point performance of the GPU and the CPU, which means that the CPU version is about as efficient in terms of wasting computational resources as the GPU. (... TODO: why is GCC so fast? ...)

A second kernel was created with similar properties as the color adjustment kernel but with only scalar operations (see Listing C.2) and almost no conditional operations. This kernel is close to a best-case for the fused SIMD version. The speed up for the LLVM

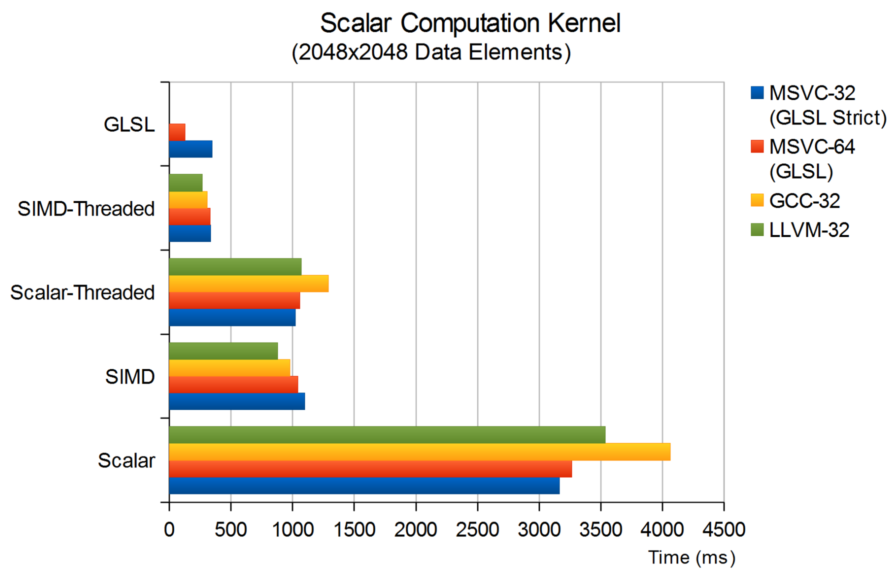


Figure 8.2.: Benchmark results for the scalar computation kernel

back end is about 4.01, which even better than the theoretical gain of SSE. The reason for this behavior is that the scalar version has an additional loop overhead, because an inner loop (see Listing C.2, line 18) has to be executed four times as often. The multi-threaded SIMD version of the LLVM back end is even faster than the GLSL version, which shows how optimized the GPU is for vector operations and that the CPU can actually compete against it for scalar compute kernels. Figure 8.2 lists the full results. The speedup factors for this kernel are very near to the theoretical values based on the number of CPU threads and SIMD components.

Sampling performance A simple convolution filter is used to measure the cost of reading data from an input array. The kernel (see Listing C.3) consists of a 10×10 Gaussian, meaning 100 input sampling operations per output data element. The number of arithmetic operations is lower than in the color adjustment kernel. The results are shown in Figure 8.3.

Some large performance differences are apparent in the different back ends. Both the LLVM and the 32-bit Visual Studio results have very little gain in the respective SIMD versions because the number of arithmetic computations is relatively small and the sampling is more expensive in the SIMD version. The expensiveness of the data sampling in the SIMD version is caused by the necessity to gather data elements scattered across the input array. This operation cannot be implemented efficiently with SSE operations.

The scalar 64-bit Visual Studio version is exceptionally slow, slower by a factor of about 2.5 compared to the corresponding 32-bit version. The reason seems to be that the optimizer is not working for this code for an unknown reason.

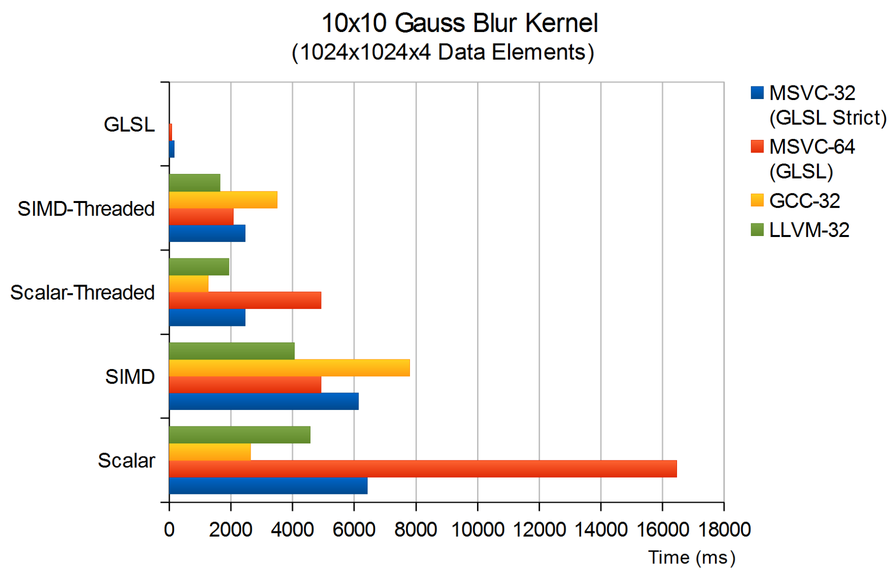


Figure 8.3.: Benchmark results for the 10 × 10 Gaussian blur kernel

Finally, the GLSL version provides a very high gain factor of about 10 to 18 compared to the multi-threaded SIMD LLVM version. The reason for this high discrepancy is that heavy use of sampling is a very common pattern in the 3D graphics area. For this reason, GPUs have the capacity to predict, prefetch and cache these accesses [HG97]. Section 9 lists some possible optimizations for the CPU back ends to make this performance gap smaller.

Another example of a kernel with a high number of sampling operations is one that computes one iteration of a Gauss-Jordan elimination algorithm to invert a matrix (see Listing C.4). The SIMD version is slower in general than the scalar version because of the overhead that sampling and condition statements have in the SIMD code. Again, the 64-bit scalar version compiled with Visual Studio is exceptionally slow because the code does not get fully optimized. Figure 8.4 shows the full results.

Loop overhead For very simple kernels, the overhead caused by computations outside of the actual kernel code can become important. Such computations are the loop that iterates over all data elements in the CPU version and kernel calls issued by the OpenGL driver in the GLSL version, as well as latencies in the driver communication. A short filter computing a vignetting effect is used to model this scenario. The kernel (see Listing C.5) performs just 7 multiplications and some subtractions.

The results in Figure 8.5 show that all back ends perform pretty equally. In particular, the GLSL version is now in the same performance range as the CPU versions. Except for the 64-bit Visual Studio version, the SIMD results are slightly worse than the scalar counterparts because of the memory gather/scatter operations needed for sampling and for writing to the output array (See Section 5.3.1).

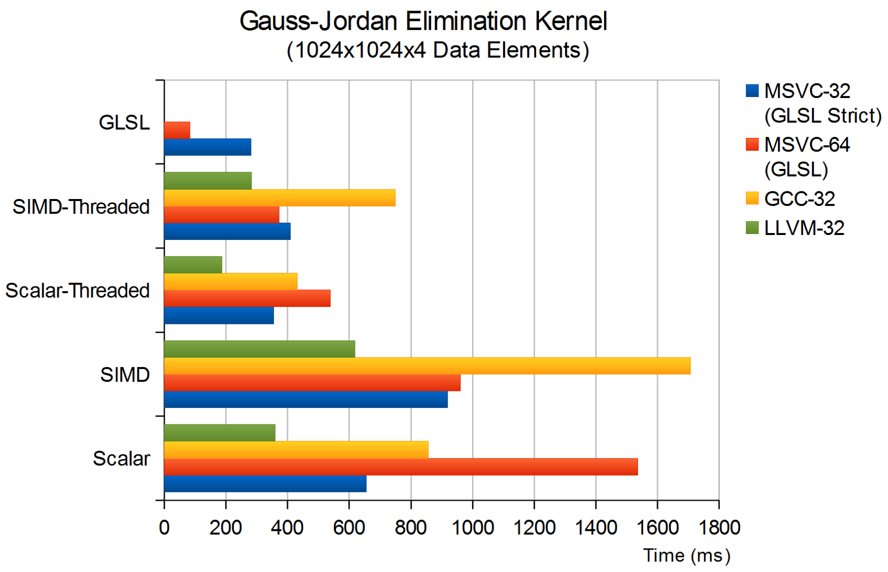


Figure 8.4.: Benchmark results for the Gauss-Jordan equation system solver kernel

Dynamic branching The most important weak spot of the SIMD versions is conditional control flow. There is a certain computational overhead, and it may be necessary to fall back to scalar computations (see Section 5.4). Two kernels are used to get an impression of the impact of conditional statements.

The first kernel (see Listing C.6) blurs an image horizontally but takes only pixels into account that lie in the same column as the current output pixel. The columns have a width of 100 pixels. The need for falling back to scalar code occurs at the edges between two such columns. This, however, means that most of the image can be executed as fused code, and the important factor is the overhead for detecting if the condition is equally true for all of the SIMD streams. Figure 8.6 shows that the overhead slightly outweighs the performance advantage of the fused code. Because of this, the SIMD versions are about 15 % slower than their scalar counterparts, except for the scalar 64-bit result for Visual Studio, which is again excessively slow.

The GLSL version is faster by a factor of 5 to 12 compared to the multi-threaded version which is caused by the faster sampling of the GPU, as with the 10×10 Gaussian blur kernel.

The second kernel (see Listing C.7) executes a sequence of multiplications and additions on the output coordinate. The sequence has a length of 100 iterations, and each iteration contains two nested if-statements. The kernel is built in a way that causes the condition to be different for most neighboring data elements, forcing a scalar path in almost all iterations. The result of the kernel is a pseudo-random pattern that is useful for detecting small differences in the precision of floating-point operations on different back ends or on different hardware.

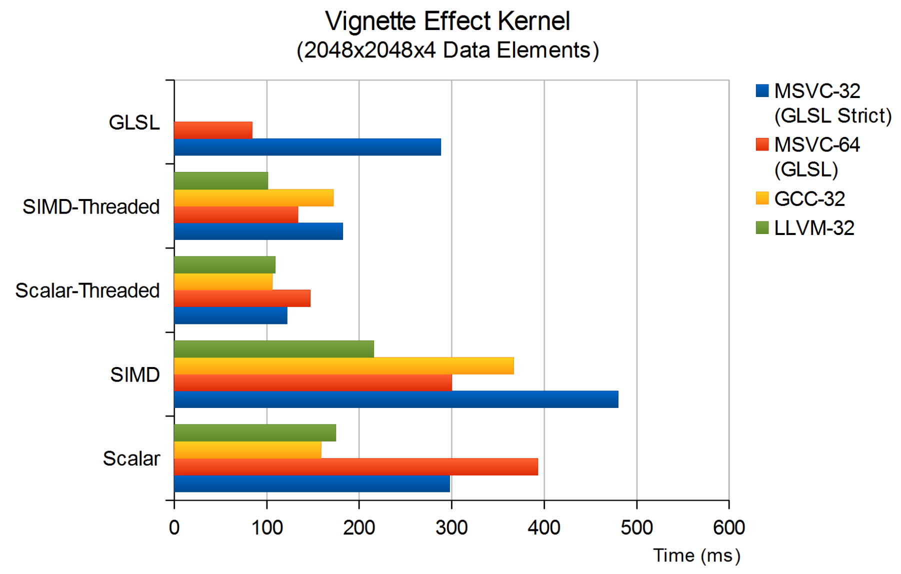


Figure 8.5.: Benchmark results for the vignette effect kernel

As shown in Figure 8.7, the fastest CPU version – the multi-threaded SIMD version compiled with Visual Studio – is slower than the GLSL versions by a factor of about 3 to 4. This shows that conditional control flow on the GPU has a comparable overhead to the CPU (or even slightly higher), assuming that the pure floating-point arithmetic performance of the GPU is about 3 to 5 times faster than the CPU, as the arithmetic benchmark results indicate.

TODO: why is SIMD-LLVM slower than scalar but not for VS?

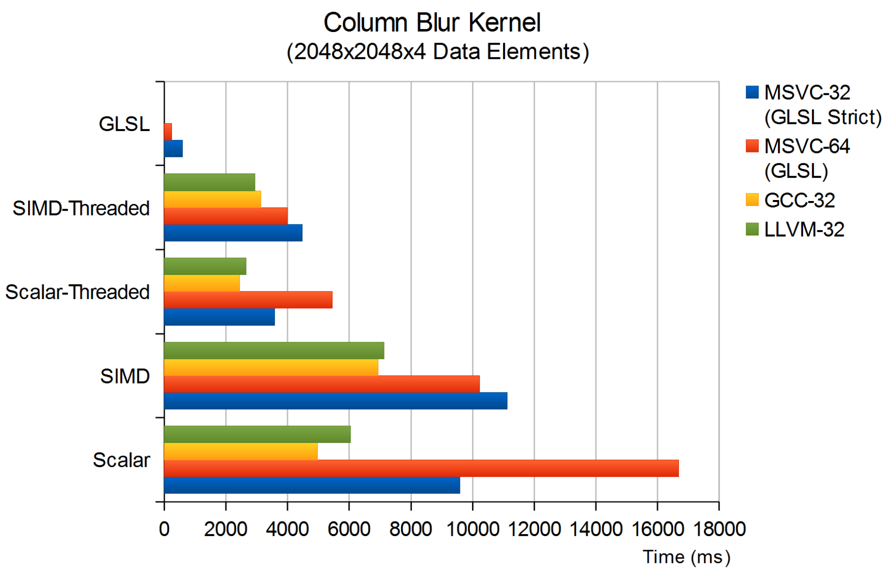


Figure 8.6.: Benchmark results for the horizontal column blur kernel

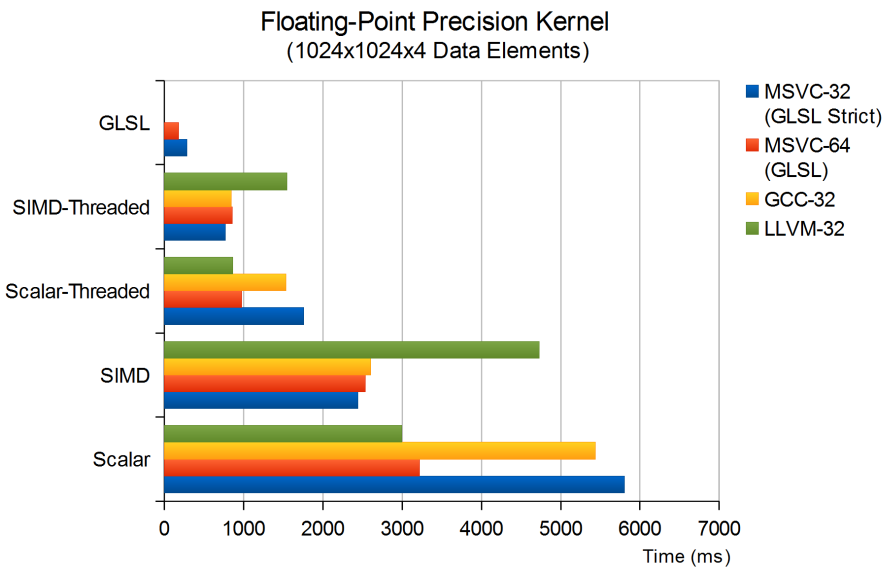


Figure 8.7.: Benchmark results for the floating point-precision test kernel

9. Conclusion and Outlook

The compiler developed for this thesis successfully solves the main problems with current cross-platform computation languages or APIs. It supports virtually all practically usable hardware by targeting the existing computation languages such as GLSL, while still providing a reliable code path by outputting C++ or LLVM code.

An additional SIMD code fusion transformation provides optimized output for systems with SSE support. While certain operations are such as conditional statements can be slower, this transformation provides a significant speedup for many practical kernels when comparing against the non-SIMD version without code fusion. This again saves the development time usually needed to make a hand-optimized version of each filter kernel.

The system also allows run-time compilation and thus supports run-time generated kernels and rapid development by avoiding compile-link cycles. The result is a practical and scalable stream language implementation for robust support of a wide range of heterogeneous hardware and software configurations.

While the current implementation of the SLURP compiler provides a working base for practical use, there are many areas in which the implementation can be extended. Improvements can be made in the areas of performance, programming convenience, platform support and expressibility.

Weakening the parallelism model A weaker parallelism model with support for global variables, aliasing using pointers or array slices, or allowing limited dependencies between different data elements can enable more flexible algorithm implementations.

Many of the target languages support such features, although with possible performance implications. However, the GLSL back end is an example where dependencies and aliasing are not possible or possible only by exploiting implementation-dependent behavior. Once the decision is made to support only the more powerful target languages, support for these features can be added.

Compile-time code interpretation An extension to the dead code elimination optimization is to not only evaluate constant expressions at compile-time but also to run whole functions through an interpreter if all of their input is constant. This allows for convenient specification of precomputation algorithms whose results are then used for the main algorithm. Without this support either the performance will suffer or the precomputation algorithm has to be written in C++ and the result is the fed into the kernel function as a uniform parameters, which is quite inconvenient in comparison.

Computing uniform expressions outside of the data loop Similar to the compile-time interpretation of constant code, uniform code – code that depends only in uniform or constant input – can be moved from inside the data loop to a place before the loop. The results of the computation are then either put into global variables for later access or passed as additional function parameters to the functions that need these values.

Additional optimizer stages Modern compilers support a multitude of optimizations [Muc97]. While many stages are already performed by the compiler of the target language, there are some targets with compilers that also do not optimize very aggressively. Examples include some GLSL compilers that are shipped with graphics drivers. Some of the more important optimizations could be added to the SLURP compiler.

OpenCL back end At the current stage, graphics cards are only supported as computation targets by using OpenGL/GLSL fragment shaders [KBR08]. OpenCL [SGS10] provides an API specifically for doing general purpose computations. Compared to OpenGL/GLSL it has a more flexible programming model (see "Weakening the parallelism model" at the beginning of this section). OpenCL is on its way to gaining widespread support across different platforms and becoming a valuable target. Since the syntax of GLSL and OpenCL is very similar, a simple modification of the GLSL back end suffices to obtain a functional OpenCL back end.

Full support for control flow statements Some statements such as "break" and "continue" have been left out of the current implementation because they considerably increase the implementation complexity. Although the language is able to express all algorithms without these statements, it can often be convenient to have them available.

Detection and optimization of certain data sampling patterns Most of the typical kernel functions have very regular patterns in the way they access the input arrays. Probably the most common pattern for image processing is to read the data element corresponding to the current output coordinate, followed by accessing a series of neighboring input elements (convolution).

Detecting such patterns and implementing an optimized replacement function for the *sample* function is expected to lead to a significant speed up. Several multiplications and additions as well as conditional branches can be avoided in the SIMD version. For simple kernels this would presumably be the most rewarding optimization in terms of performance.

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
AVX	Advanced Vector Extensions
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
JIT	Just-In-Time (compilation)
LLVM	Low Level Virtual Machine
LL(1)	Left to right parsing using Leftmost derivation with 1 lookahead
GCC	GNU Compiler Collection
GLSL	OpenGL Shading Language
GNU	GNU's Not Unix
GPU	Graphics Processing Unit
HLSL	High-Level Shading Language
MIMD	Multiple Instruction, Multiple Data
MMX	Marketing term that looks like an abbreviation
RAM	Random Access Memory
SIMD	Single Instruction, Multiple Data
SLURP	Stream Language Unified Runtime Programming, the programming language framework described in this thesis
SPMD	Single Program Multiple Data
SSA	Static Single Assignment
SSE	(Intel) Streaming SIMD Extensions
VM	Virtual Machine
YACC	Yet Another Compiler Compiler

List of Figures

Listings

Bibliography

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9:491–542, October 1987. URL: `http://doi.acm.org/10.1145/29873.29875`.

[AK01] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[Ale11] Andrei Alexandrescu. *The D Programming Language*. Addison Wesley, 2011.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. URL: `http://doi.acm.org/10.1145/1465482.1465560`.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23:777–786, August 2004. URL: `http://doi.acm.org/10.1145/1015706.1015800`.

[BS93] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems – Volume 4*, page 3, Berkeley, CA, USA, 1993. USENIX Association. URL: `http://portal.acm.org/citation.cfm?id=1295480.1295483`.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM. URL: `http://doi.acm.org/10.1145/75277.75280`.

[CJA+09] James Charles, Preet Jassi, Narayan S. Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the Intel Core i7 Turbo Boost feature. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 188–197, Washington, DC, USA, 2009. IEEE Computer Society. URL: `http://dx.doi.org/10.1109/IISWC.2009.5306782`.

[Dar01] Frederica Darema. The SPMD model: Past, present and future. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 1–, London, UK, 2001. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=648138.746808`.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science and Engineering*, 5:46–55, 1998. URL: `http://doi.ieeecomputersociety.org/10.1109/99.660313`.

[DS90] C. Donnelly and R. Stallman. BISON the YACC-compatible parser generator. 1990. URL: `http://www.gnu.org/software/bison/`.

[GDORT$^+$11] Jing Guo, Antonio Wendell De Oliveira Rodrigues, Jeararajan Thiyagalingam, Frédéric Guyomarch, Pierre Boulet, and Sven-Bodo Scholz. Harnessing the Power of GPUs without Losing Abstractions in SaC and ArrayOL: A Comparative Study. In *HIPS 2011, 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Anchorage (Alaska), États-Unis, May 2011. IEEE. URL: `http://hal.inria.fr/inria-00569100/en/`.

[GRS] Brian Gough, Foreword Richard, and M. Stallman. An introduction to GCC for the GNU compilers gcc and g++.

[GS06] Clemens Grelck and Sven-Bodo Scholz. SAC: a functional array language for efficient multi-threaded execution. *Int. J. Parallel Program.*, 34:383–427, August 2006. URL: `http://dl.acm.org/citation.cfm?id=1182756.1182760`, `doi:10.1007/s10766-006-0018-x`.

[HG97] Ziyad S. Hakura and Anoop Gupta. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th annual international symposium on computer architecture*, ISCA '97, pages 108–120, New York, NY, USA, 1997. ACM. URL: `http://doi.acm.org/10.1145/264107.264152`.

[Int06] Intel. Intel C++ compiler for linux intrinsics reference. 2006. URL: `http://software.intel.com/file/6373`.

[Int11] Intel. Intel spmd program compiler, 2011. URL: `http://ispc.github.com/`.

[ISO03] ISO. ISO/IEC 14882:2003: Programming languages – C++. *International Organization for Standardization*, 2003.

[Ive62] Kenneth E. Iverson. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, AIEE-IRE '62 (Spring), pages 345–351, New York, NY, USA, 1962. ACM. URL: http://doi.acm.org/10.1145/1460833.1460872.

[Joh79] S. C. Johnson. YACC – yet another compiler-compiler. *UNIX Programmer's Manual*, 2B, 1979. URL: http://invisible-island.net/byacc/byacc.html.

[KBR08] John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL shading language. 2008. URL: https://www.opengl.org/registry/doc/GLSLangSpec.Full.1.30.08.pdf.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[MBH+02] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):1–12, 2002. URL: http://download.intel.com/technology/itj/2002/volume06issue01/art01_hyper/vol6iss1_art01.pdf.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22:896–907, July 2003. URL: http://doi.acm.org/10.1145/882262.882362.

[MSA+83] J. MCGRAW, S. SKEDZIELEWSKI, S. ALLAN, D. GRIT, R. OLDEHOEFT, J. GLAUERT, I. DOBES, , and P. HOHENSEE. SISAL: streams and iteration in a single-assignment language. language reference manual version 1.1. 1983.

[Muc97] Steven S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[NSL+11] C.J. Newburn, Byoungro So, Zhenying Liu, M. McCool, A. Ghuloum, S.D. Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Code Generation and Optimization*, pages 224 – 235. IEEE/ACM, 2011.

[Nvi11] Nvidia. What is CUDA, 2011. URL: `http://developer.nvidia.com/what-cuda`.

[Phe08] Chuck Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23:298–298, April 2008. URL: `http://dl.acm.org/citation.cfm?id=1352079.1352134`.

[PM03] C. Peeper and J. L. Mitchell. Introduction to the DirectX 9 high level shading language. In *ShaderX2: Introduction and Tutorials with DirectX 9*. Wordware, 2003.

[SGS10] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12:66–73, 2010. URL: `http://doi.ieeecomputersociety.org/10.1109/MCSE.2010.69`.

[Tay07] Stewart Taylor. *Optimizing Applications for Multi-Core Processors: Using the Intel Integrated Performance Primitives.* Intel Press, 2007.

[TPO06] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. *SIGOPS Oper. Syst. Rev.*, 40:325–335, October 2006. URL: `http://doi.acm.org/10.1145/1168917.1168898`.

[WG107] WG14. The C99 language standard. (WG14 N1256), 2007. URL: `http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf`.

[Wik11] Wikipedia. Dangling else – Wikipedia, the free encyclopedia, 2011. [Online; accessed 04-July-2011]. URL: `http://en.wikipedia.org/w/index.php?title=Dangling_else&oldid=422212194`.

A. Language Grammar

```
global_decl →        typedef | function

typedef →            'alias' IDENT '=' type ';'

function →           ('kernel' | 'function') type IDENT
                      param_decl_list compound_stmt

param_decl_list →    '(' ( ε | param (',' param)* ) ')'

param →              type IDENT

stmt →               compound_stmt | non_comp_stmt

compound_stmt →      '{' stmt* '}'

non_comp_stmt →      decl_or_expr_stmt | return_stmt
                   | if_stmt | for_stmt | while_stmt
                   | do_stmt

decl_or_expr_stmt → type IDENT (ε | '=' expr) ';'
                   | expr ';'

return_stmt →        'return' expr ';'

if_stmt →            'if' '(' expr ')' stmt
                     (ε | 'else' stmt)

NOTE: if/else ambiguity is resolved by matching the 'else'
      with the closest 'if' that has no 'else' already
      matched

for_stmt →           'for' '(' (decl_or_expr_stmt | ';')
                     expr ';' expr ')' stmt

while_stmt →         'while' '(' expr ')' stmt

do_stmt →            'do' stmt 'while' '(' expr ')' ';'

expr →               assign_expr

assign_expr →        or_expr (ε | ('=' | '+=' | '-='
                   | '*=' | '/=' | '%=') assign_expr)

or_expr →            and_expr (ε | '||' or_expr)

and_expr →           equal_expr (ε | '&&' and_expr)

equal_expr →         add_expr (ε | ('==' | '!=' | '<' | '<='
                   | '>=' | '>') equal_expr)

add_expr →           mul_expr (ε |
                     ('+' | '-') add_expr)

mul_expr →           cast_expr (ε |
                    ('*' | '/' | '%') mul_expr)

cast_expr →         post_expr | 'cast' '(' type ')' cast_expr

post_expr →         unary_expr ('.' IDENT
                  | '[' expr (', ' expr)* ']'
                  | '(' (ϵ | expr (',' expr)*) ')')

unary_expr →        var_expr
                  | ('+' | '−' | '!' | '~' | '++' | '−−') post_expr

var_expr →          IDENT | NUMBER | 'true' | 'false'
                  | IDENT '(' (ϵ | expr (',' expr)*) ')'
                  | '(' expr ')'
                  | '[' (ϵ | expr (', ' expr)*) ']'

type →              base_type ('[' NUMBER ']')*

base_type →         type_attributes (struct_type | IDENT)

struct_type →       'struct' '{' (type IDENT ';')* '}'

type_attributes →   ('uniform')*
```

Listing A.1: The language grammar

B. Built-in Functions and Types

```
// opaque pointer type
pointer

// void type
void

// boolean values
bool
bool2 : bool[2]
bool3 : bool[3]
bool4 : bool[4]
bool2x2 : bool[2, 2]
bool3x3 : bool[3, 3]
bool4x4 : bool[4, 4]

// integer values
int
int2 : int[2]
int3 : int[3]
int4 : int[4]
int2x2 : int[2, 2]
int3x3 : int[3, 3]
int4x4 : int[4, 4]

// floating-point values
float
float2 : float[2]
float3 : float[3]
float4 : float[4]
float2x2 : float[2, 2]
float3x3 : float[3, 3]
float4x4 : float[4, 4]

// input array access
sampler2 : uniform struct
```

Listing B.1: Built-in types

```
// common functions
float[N] rcp(float[N] x)                                | computes the reciprocal of x
float[N] sqrt(float[N] x)                               | computes the square root of x
float[N] rsqrt(float[N] x)                              | computes the 1.0 / sqrt(x)
float[N] exp(float[N] x)                                | computes e to the power of x
float[N] log(float[N] x)                                | computes the natural logarithm of
     x
float[N] pow(float[N] x, float[N] y)                    | computes x to the power of y

// trigonometric functions
float[N] sin(float[N] x)                                | computes the sine of x
float[N] cos(float[N] x)                                | computes the cosine of x
float[N] tan(float[N] x)                                | computes the tangent of x
float[N] asin(float[N] x)                               | computes the inverse sine of x
float[N] acos(float[N] x)                               | computes the inverse cosine of x
float[N] atan(float[N] x)                               | computes the inverse tangent of x
float[N] atan2(float[N] x, float[N] y)                  | computes the inverse tangent of x
    /y, with valid values for y==0

// rounding functions
float[N] trunc(float[N] x)                              | rounds x to the next integer
    towards zero
float[N] floor(float[N] x)                              | rounds x to the largest integer
    that is not larger than x
float[N] ceil(float[N] x)                               | rounds x to the smallest integer
    that is not smaller than x
float[N] round(float[N] x)                              | rounds x to the nearest integer

// misc. functions
float[N] lerp(float[N] x, float[N] a, float[N] b)      | computes a linear interpolation
    between a and b
float[N] select(bool[N] cond, float[N] a, float[N] b) | returns a if cond==true and b
    otherwise
float[N] select(bool[N] cond, int[N] a, int[N] b)     | returns a if cond==true and b
    otherwise
float[N] clamp(float[N] x, float[N] a, float[N] b)     | clamps x to the range [a .. b]
float[N] step(float edge, float[N] x)                   | returns 1.0 if x>edge and 0.0
    otherwise
float[N] step(float[N] edge, float[N] x)                | returns 1.0 if x>edge and 0.0
    otherwise
float[N] sign(float[N] x)                               | returns 0.0 if x==0.0, 1.0 if x>0
     and -1.0 if x<0
float[N] abs(float[N] x)                                | returns the absolute value of x
float[N] sat(float[N] x)                                | clamps x to the range [0.0 ..
     1.0]

// vector functions
float dot(float[N] x, float[N] y)                       | computes the scalar product of x
     and y
float dot(float[N] x)                                   | computes the euclidean norm of x
     and y
float3 cross(float3 x, float3 y)                        | computes the cross product of x
     and y

// sampling
float4 sample(sampler2 source, float2 coord)            | returns the array element pf '
    source' at the specified coordinate
```

Listing B.2: Built-in functions

C. Benchmark Kernels

```
function float rgblum(float3 col) { return dot(col, float3(0.33, 0.34, 0.33)); }
function float avg(float3 v) { return dot(v, float3(1.0, 1.0, 1.0)/3.0); }
function float3 saturate(float3 v) { return clamp(v, 0.0, 1.0); }

function float my_exp(float x){ return 1.0 + x * (0.5 + x * (0.1667 + x * (0.0417 + x *
    0.0083))); }
function float3 my_exp(float3 x){ return 1.0 + x * (0.5 + x * (0.1667 + x * (0.0417 + x *
     0.0083))); }
function float my_ln(float x){ float y = x - 1.0; return y * (1.0 + y * (-0.5 + y *
    (0.3333 + y * -0.25))); }
function float3 my_ln(float3 x){ float3 y = x - 1.0; return y * (1.0 + y * (-0.5 + y *
    (0.3333 + y * -0.25))); }
function float my_pow(float x, float y){ return my_exp(y * my_ln(x)); }
function float3 my_pow(float3 x, float3 y){ return my_exp(y * my_ln(x)); }
function float my_sqrt(float x){ float y = x - 1.0; return 1.0 + y * (0.5 + y * (-0.125 +
     y * (0.0625 + y * -0.0391))); }
function float3 my_sqrt(float3 x){ float3 y = x - 1.0; return 1.0 + y * (0.5 + y *
    (-0.125 + y * (0.0625 + y * -0.0391))); }
function float my_rsqrt(float x){ return 1.0 / my_sqrt(x); }
function float3 my_rsqrt(float3 x){ return 1.0 / my_sqrt(x); }
function float my_norm(float3 x){ return my_rsqrt(dot(x, x)); }
function float my_atan(float x){ return x*my_rsqrt(abs(x)); }
function float3 my_atan(float3 x){ return x*my_rsqrt(abs(x)); }

function float3 adjustBrightness_gamma(float3 fc, float b)
{
  float bexp = my_exp(-2.0 * b);
  float3 f = my_pow(fc * min(1.0 + b*0.3, 1.0), float3(1.0,1.0,1.0)*bexp);
  float fbl = rgblum(f);
  return (f - fbl) * (1.0 + 0.5*max(b, 0.0)) + fbl;
}

function float3 adjustContrast_atan(float3 f, float c)
{
  float cexp = my_exp(3.0*c);
  float3 fcb = 0.5*my_atan((f-0.5)*cexp) / my_atan(0.5*max(cexp, 1.0)) + 0.5;
  float cs = min(abs(c), 1.0);
  return lerp(cs, f, fcb);
}

function float3 brightnessContrast(float3 fc, float b, float c, float3 refcolori)
{
  float3 f = adjustBrightness_gamma(fc, b);
  f = adjustContrast_atan(f, c);
  return f;
}

function float rgbmax(float3 rgb){ return max(max(rgb[0], rgb[1]), rgb[2]); }

function float3 applyTint(float3 c, float3 tintNormalized)
{
  float3 r = my_pow(c, my_exp(avg(3.0*tintNormalized)-4.0*tintNormalized));
  float s = my_exp(-my_norm(tintNormalized));
  return (r - avg(r)) * lerp(s, avg(c), 1.0) + avg(r);
}

kernel float4 colorAdjust(float2 pos, uniform sampler2 input0)
{
  // adjustment parameters
  float globalContrast = 0.3;
  float contrast = 0.3;
  float brightness = 0.3;
  float saturation = 0.3;
  float3 tintNormalized = float3(1.0, 1.0, 1.0);

  float4 cur = sample(input0, pos);

  float3 result = float3(cur[0], cur[1], cur[2]);

  float minmul = my_exp(-1.0);
  float satmul = my_exp(saturation);
  result = saturate((result - rgblum(result)) * (satmul - minmul) / (1.0 - minmul) +
      rgblum(result));

  result = applyTint(result, tintNormalized);

  float contr_l = (globalContrast + contrast);
  float contr_g = contr_l * sign(globalContrast);
  contr_g = clamp(contr_g, 0.0, abs(globalContrast)) * sign(globalContrast);
  contr_l = contr_l - contr_g;

  float3 contr_l_refcolor = float3(0.5, 0.5, 0.5);
  result = saturate((result - contr_l_refcolor) * my_exp(contr_l) + contr_l_refcolor);
  result = saturate(brightnessContrast(result, brightness, contr_g, float3(0.5, 0.5, 0.5)
      ));

  return float4(result, 1.0);
}
```

Listing C.1: Color adjustment kernel

```
kernel float4 scalarArithmeticTest(float2 pos)
{
  float a = pos[0] * 0.0001;
  float b = pos[1] * 0.0001;

  float c = a*a + b*b;
  float d = a / c;
  float e = b / c;

  float aa = 0.1;
  float ab = 0.9;
  float ba = 0.9;
  float bb = 0.1;

  c = 0.2 * c;
  float f = 1.0 - c;

  for( uniform int i = 0; i < 40; ++i ){
    float tmp = aa * e - ab * d;
    e = ba * e + bb * d;
    e = (e + c) * f;
    d = tmp;
  }

  return float4(e, d, d*e, 1.0);
}
```

Listing C.2: Scalar kernel with an emphasis on arithmetic

```
function float myexp(float x)
{
  float tmp = (25.0-x*x) / 25.0;
  return tmp * tmp;
}

kernel float4 gauss(float2 pos, uniform sampler2 input0)
{
  float4 sum = float4(0.0);
  float wsum = 0.0;

  for( uniform int i = -5; i < 5; ++i ){
    float2 xy = float2(0.0, cast(float)i);
    for( uniform int j = -5; j < 5; ++j ){
      xy[0] = cast(float)j;
      float w = myexp(-0.1*dot(xy, xy));
      wsum = wsum + w;
      sum = sum + w * sample(input0, pos + xy);
    }
  }

  return sum / wsum;
}
```

Listing C.3: "10 $\times$ 10 Gaussian blur approximation kernel

```
kernel float4 gaussjordan(float2 pos, uniform sampler2 input0)
{
  float pivot = 10.0;

  float4 v = sample(input0, pos) / sample(input0, float2(pivot, pos[1]));

  if( abs(pos[1] - pivot) > 0.5 )
    v = v - sample(input0, float2(pos[0], pivot)) / sample(input0, float2(pivot, pivot));

  return v;
}
```

Listing C.4: Gauss-Jordan elimination step kernel

```
kernel float4 vignette(float2 pos, uniform sampler2 input0)
{
  float4 s = sample(input0, pos);

  float ires = 1.0 / 4096.0;
  float2 posnorm = pos * ires;
  float mul = 1.0 - posnorm[0] * (1.0 - posnorm[0]) * posnorm[1] * (1.0 - posnorm[1]);
  mul = (1.0 - mul*mul*mul) * 4.0;
  s = s * mul;

  return s;
}
```

Listing C.5: Vignetting effect kernel

```
function float myexp(float x)
{
  float tmp = (25.0-x*x) / 25.0;
  return tmp * tmp;
}

kernel float4 columnblur(float2 pos, uniform sampler2 input0)
{
  float4 sum = float4(0.0, 0.0, 0.0, 0.0);
  float wsum = 0.0;

  float col_start = floor(pos[0] / 100.0) * 100.0 - pos[0];
  float col_end = col_start + 100.0;

  for( float x = max(-20.0, col_start); x < min(20.0, col_end); x = x + 1.0 ){
    float w = myexp(-0.0025*x*x);
    wsum = wsum + w;
    sum = sum + w * sample(input0, pos + float2(x, 0.0));
  }

  return sum / wsum;
}
```

Listing C.6: Column blur kernel

```
kernel float4 chaos(float2 pos, uniform sampler2 input0)
{
  float c = (pos[0] * 1024.0 + pos[1]) * 0.000001;
  for( uniform int i = 0; i < 100; ++i ){
    c = c - floor(c);
    if( c < 0.3 )
      c = 2.0 * c;
    else if( c < 0.7 )
      c = (c - 0.5) * 4.0 + 0.5;
    else
      c = 1.0 - 2.0 * c;
  }

  return float4(c, c, c, 1.0);
}
```

Listing C.7: Floating-point precision test kernel