Whole-Function Vectorization

Ralf Karrenberg Sebastian Hack
Saarland University
{karrenberg,hack}@cs.uni-saarland.de

Abstract

Data-parallel programming languages are an important component in today's parallel computing landscape. Among those are domain-specific languages like shading languages in graphics (HLSL, GLSL, RenderMan, etc.) and "general-purpose" languages like CUDA or OpenCL. Current implementations of those languages on CPUs solely rely on multi-threading to implement parallelism and ignore the additional intra-core parallelism provided by the SIMD instruction set of those processors (like Intel's SSE and the upcoming AVX or Larrabee instruction sets).

In this paper, we discuss several aspects of implementing data-parallel languages on machines with SIMD instruction sets. Our main contribution is a language- and platform-independent code transformation that performs whole-function vectorization on low-level intermediate code given by a control flow graph in SSA form.

We evaluate our technique in two scenarios: First, incorporated in a compiler for a domain-specific language used in real-time ray tracing. Second, in a stand-alone OpenCL driver. We observe average speedup factors of 3.9 for the ray tracer and factors between 0.6 and 5.2 for different OpenCL kernels.

Categories and Subject Descriptors D.3.4 [*Processors*]: code generation, compilers, optimization; C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: Single-instruction-stream, multiple-data-stream processors (SIMD)

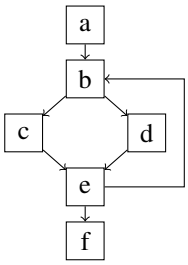
General Terms Parallelism, Performance, Algorithms, Optimization

Keywords SIMD, Vectorization, Data Parallelism, Code Generation, Ray Tracing, OpenCL

CGO '11 April 02-06, Chamonix, France.

1. Introduction

Data-parallel programming is an important concept in today's parallel computing landscape. A variety of programming languages have been created to support data-parallel programming over the last decades. Those cover domain-specific languages (DSLs) like shading languages in graphics (e.g. HLSL, GLSL, RenderMan) or, recently, general-purpose languages like CUDA or OpenCL. All those languages share the SIMD concept: A single piece of code is executed in parallel on different elements of arrays of data. This does not mean that every *instance* of that piece of code executes exactly the same instructions. Consider the example in Figure 1. The code given by the control flow graph (CFG) on the left is applied to four different inputs in parallel. This results in four different execution traces as shown in the table on the right: For example, instance 1 executes the blocks `a b c e f` while instance 3 executes the blocks `a b c e b c e f`.



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

Figure 1. Some piece of code and four execution traces.

Diverging control flow makes SIMD execution more complicated: Instance 1 executes block `c` but instance 3 executes block `d` instead. Assuming that block `c` and `d` contain different instructions, they cannot be executed in a SIMD fashion. The usual solution is that the SIMD program executes both branches and compensates for the unwanted effects: Instance 1 ignores the computations of block `d`, instance 2 ignores the effects of block `c`, and so on. This is also called *predicated execution* [23].

In practice, this behavior is implemented in different ways: GPUs perform predicated execution implicitly in hardware: Each instance of the data-parallel program is mapped to a scalar core in a multi-core processor. All in-

stances are executed in lock step. A central control unit stalls a processor on code regions where its instance is inactive.

This paper is concerned with the implementation of data-parallel languages on processors that require *explicit* vectorization. Such processors have vector register files and dedicated SIMD instructions (e.g. Intel's SSE and IBM's AltiVec instruction sets). The current register size is 16 bytes (SSE, AltiVec), with 32 bytes (AVX) and 64 bytes (LRBni, Larrabee new instructions) on the horizon. The ubiquity of vector instruction sets and the increase of the vector width in future architectures motivates the investigation of implementing data-parallel programs on these architectures.

To use SIMD instructions in the presence of diverging control flow, the programmer or the *compiler* has to replace control flow by data flow [2]. For code without loops, this is commonly called *if conversion*. For example, consider the function `f` in Figure 2 and its version without control flow `f'`. The function `select` chooses, dependent on the mask value, either `s` or `t`. The function `f_sse` in Figure 2 shows how a C/C++ programmer would implement a vectorized variant of the same code using intrinsics for Intel's SSE instruction set.

There exist several approaches that convert control flow into data flow to perform vectorization (Section 5 discusses related work in further detail). Those approaches typically originate from the parallel computing community, where parallelization is performed early in the compilation process; often already on the source level. This has two major disadvantages:

1. By transforming control flow to data flow too early in the compilation process, all analyses and optimizations in the compiler that make use of control flow information are rendered useless. These analyses (for example conditional constant propagation or loop invariant code motion) would need to be rewritten to be able to analyze the predicated vector code.
2. Modern compiler infrastructures use virtual instruction sets as a code exchange format (for example LLVM bitcode [18] or Nvidia's PTX) to decouple front ends from code generators. Those representations commonly use control flow graphs of instructions to represent the code. Performing control to data flow conversion already in the front end destroys the portability of the generated code for two reasons: First, the transformation would need to be undone to use the code for architectures that do not require control to data flow conversion. Second, even if the target architecture requires control flow conversion, the front end would need to address several architectural parameters like the vector width and details about the implementation of masking (see Section 2). Thus, the code is no longer portable.

1.1 Contributions

We argue that vectorization should be performed late in the compilation process. We present a code transformation that performs control to data flow conversion on code represented in nowadays common intermediate representations such as LLVM bitcode. More specifically, we cast vectorization as a program transformation from (and to) control-flow graphs in static single assignment (SSA) [4] form. In summary, we make the following contributions:

- We present a whole-function vectorization transformation of SSA-form control flow graphs for processors with SIMD instructions. SSA is particularly useful for vectorization for those processors because ϕ-functions give the locations where blending code (see the `select` instruction in Figure 2) has to be placed. Furthermore, we maintain the SSA form during our transformation.
- We present a pass that generates predicated code for *arbitrary* control flow on architectures without hardware support for predicated execution by carefully placing blend operations.
- We present a data flow analysis that identifies certain constraints on the values of variables in different parallel instances. This helps to respect the alignment and consecutiveness restrictions of the memory access instructions of today's SIMD architectures. Furthermore, computations that produce the same value in every instance do not need to be vectorized and can remain scalar.
- We implemented our algorithm in the LLVM compiler infrastructure and evaluated it in two practical scenarios. First, we added it to a compiler for the RenderMan shading language, a computer graphics DSL that is widely used for visual effects in the movie industry, and performed experiments with a real-time ray tracer. Second, we implemented a prototypical OpenCL driver that uses the presented vectorization algorithm. We show an average speedup of 3.9 for the ray tracer and speedups between 0.6 and 5.2 for different OpenCL applications.

1.2 Structure of this Paper

In the next section, we discuss the design features of current SIMD architectures that are relevant for this paper. In Section 3 we briefly outline the data-parallel programs we consider. Section 4 presents the core contribution of this paper, the whole-function vectorization for SSA-form programs. Section 5 discusses related work and Section 6 presents our experimental evaluation.

2. SIMD Instruction Sets

In this section, we discuss the implications of the design of SIMD instruction sets on the implementation of data-parallel languages. Usually, the SIMD unit has SIMD registers of a certain bit width (e.g. 128 for SSE). One such register holds a vector of values of the same type. Different instruction sets

```
float f(float a, float b) {
  float r;
  if (a > b) r = a + 1;
  else       r = a - 1;
  return r;
}
```

```
float f'(float a, float b) {
  bool mask = a > b;
  float s = a + 1;
  float t = a - 1;
  float r = select(mask, s, t);
  return r;
}
```

```
__m128 f_sse(__m128 a, __m128 b) {
  __m128 mask = _mm_cmpgt_ps(a, b);
  __m128 s    = _mm_add_ps(a, _mm_one);
  __m128 t    = _mm_sub_ps(a, _mm_one);
  __m128 r    = _mm_blendv_ps(mask, s, t);
  return r;
}
```

Figure 2. A scalar program with control flow (f). The same program with control flow replaced by data flow (f'). An implementation of the latter using SSE intrinsics (f_sse).

provide support for multiple data types. For example, SSE has comprehensive support to hold either four floats, four ints, or two doubles. Sometimes, other data types (smaller ints) are supported as well. For the ease of presentation, we only regard data types of a fixed, equal size (four bytes). Throughout the paper, we will refer to the number of elements of that type that fit into a single register by the *SIMD width* W.

Memory Access. Commonly, a vector value is loaded (stored) by reading (writing) the elements *consecutively* from (to) an *aligned* address. Some instruction sets also allow accessing unaligned addresses. Those accesses are typically slower because they might involve touching multiple cache lines. Furthermore, an access to a potentially unaligned address is usually signalled by using a special instruction, because the aligned load/store instructions raise an exception upon an unaligned access.

Upcoming architectures [26] will also feature gather (scatter) operations that can load (store) from (to) a base address using a vector of (possibly non-consecutive) offsets. On architectures without scatter/gather instructions such non-consecutive accesses are very slow. This is because the vector elements have to be loaded one by one to different registers and incrementally moved into the target register. The *efficient* consecutive access can be seen as a special case of gathering with the index vector $\langle\, 0, 1, \ldots, W-1 \,\rangle$. Hence, it will be important for our vectorization pass to track the *consecutiveness* and *alignment* of indices that are used in loads and stores.

Pointers. In general it is not portable to store pointers in SIMD registers. In the case of SSE, one could put 4 pointers into a SIMD register in 32 bit mode. If the processor executes in 64 bit mode, only 2 pointers could be put into a SSE register. Therefore, instructions that use pointers in SIMD registers are uncommon.

Masking. As presented in the introduction, vectorization translates control flow to data flow. An instance potentially executes code that it would not have executed in the scalar version of the program using control flow. Thus, the effects of these computations have to be undone. This is done by masking. In the vectorized program, the control condition is made explicit as a vector of masks. The `select` function that is introduced by the control to data flow conversion uses that mask to *blend* two values at former control flow joins. Consider again the SSE code in Figure 2. Assume that the comparison (`_mm_cmpgt_ps`) produces the `mask`

$$\langle\, \texttt{FFFFFFFF}_{16}\ \texttt{00000000}_{16}\ \texttt{00000000}_{16}\ \texttt{FFFFFFFF}_{16} \,\rangle$$

for some program execution. By performing

```
(mask & s) | (~mask & t)
```

implemented by `_mm_blendv_ps` the variable `r` contains the correct values for every instance: Instances 0 and 3 contain the value of `a` and instances 1 and 2 the value of `b`.

Some architectures, e.g. Larrabee, have a dedicated mask register file to store those masks. Furthermore, every instruction can be given a mask register that controls which vector elements of the target registers shall be affected by the instruction. This makes explicit blending using `select` instructions unnecessary.

For operations that can produce side-effects, masking out results afterwards is not enough: their execution for inactive instances has to be prevented. This is achieved by splitting up the operation into scalar, sequential executions guarded by `if`-statements (see Section 4.7).

3. Data-Parallel Programs

Our model of a data-parallel program is close to the one of CUDA and OpenCL. In our setting, a data-parallel program is given by a function f. Executing the data-parallel program f means executing N instances of f. The temporal order of those instances is unspecified. Moreover, some of them can run in parallel. Every instance of f obtains the number of the instance as a parameter called the *instance ID* `tid`. The values of `tid` are pairwise different for every instance of f and range from 0 to N.

A straightforward implementation that already exploits some parallelism subdivides the instance ID range into T equally-sized parts and distributes those among T threads (cores). Inside a thread, N/T instances of f run sequentially. In this paper, we also want to exploit intra-core parallelism by vector instructions. Hence, we are going to transform f into a function $\vec{f}$ that executes W instances simultaneously, where W is the width of the processor's SIMD registers: we vectorize the *whole function* f. A multi-threaded implementation then applies $\vec{f}$ $(N/W)/T$ times sequentially per thread:

```
void apply(vec_func f, input d, int thread, int N) {
  int simd_instances = N / W;
  int inst_per_th    = simd_instances / T;
  int start          = thread * inst_per_th;
  int stop           = start + inst_per_th;
  for (int tid = start; tid < stop; tid += W)
    f(tid, d);
  // run rest, if N % W != 0
}
```

Note that multi-threading among different cores is orthogonal to SIMD vectorization and is disregarded from now on.

3.1 Program Representation

We consider f to be given in a *typed* low-level representation. A function is represented as a control flow graph of instructions. Furthermore, we require that f is in SSA form, that is every variable has a single static assignment and every use of a variable is dominated by its definition. A prominent example of such a program representation is LLVM bitcode [18] which we also use in our evaluation (Section 6). We will restrict ourselves to a subset of a language that contains only the relevant elements for this paper. Figure 3 shows its types and instructions. Other instructions, such as jumps or arithmetic and comparison operators are straightforward and omitted for the sake of brevity.

Types

$$\begin{array}{rcl} \tau & = & \texttt{unit} \mid \beta \mid \nu \mid \pi \\ \beta & = & \texttt{bool} \mid \overrightarrow{\texttt{bool}} \\ \pi & = & \nu \mid \pi* \\ \nu & = & \sigma \mid \vec{\sigma} \\ \sigma & = & \texttt{int} \mid \texttt{float} \end{array}$$

Instructions

$$\begin{array}{rcl} \texttt{load} & : & \pi* \to \pi \\ \texttt{store} & : & \pi* \times \pi \to \texttt{unit} \\ \texttt{gep} & : & \pi* \times \texttt{int} \to \pi* \\ \texttt{arg} & : & \texttt{int} \to \pi \\ \texttt{phi} & : & \pi \times \pi \to \pi \\ \texttt{tid} & : & \texttt{int} \end{array}$$

Figure 3. Program representation types and instructions

This program representation reflects today's consensus of instruction set architectures well. Scalar data types (`int` and `float`) also exist in a vectorized form ($\vec{\sigma}$). However, there are *no vectors of pointers.* The `gep` instruction ("get element pointer") performs address arithmetic and can thus not be vectorized. `load` (`store`) takes a base address and reads (writes) the elements of the vector consecutively from (to) this address. The `bool` type is special (we do not allow taking pointers of it) because its representation differs on SIMD architectures with implicit and explicit blending (see Section 2).

The function `tid` returns the instance ID of the running instance (see above). `phi` represents the usual ϕ-function from SSA. $\arg(i)$ accesses the i-th argument to f. We assume that all pointer arguments to f are aligned to the SIMD register size.

4. Whole-Function Vectorization

The whole-function vectorization algorithm (short: the *vectorizer*) consists of six phases:

1. Preparatory transformations (Section 4.1).
2. Vectorization analysis (Section 4.2).
3. Mask generation (Section 4.3).
4. Select generation (Section 4.4).
5. CFG linearization (Section 4.5).
6. Instruction vectorization (Section 4.7).

4.1 Preparatory Transformations

Before vectorization, a few preparatory transformations are performed. Most notably, loops are simplified to ensure that each loop has exactly one incoming edge and one backedge. This guarantees the existence of a unique loop header, a unique loop pre-header (the block from which the loop is entered) and a unique loop latch (the block from which an edge leads back to the header). Section 4.6 describes how the algorithm works on graphs with irreducible control-flow.

4.2 Vectorization Analysis

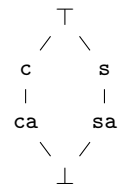
Most instructions can be simply vectorized by exchanging them with their vector counterparts (e.g. an `add` is turned into a vector `add`). However, current SIMD architectures and our language (see Section 3.1) have instructions for address arithmetic (`gep`) and memory access (`load`/`store`) that take vectors of addresses. A simple approach would split the vector containing the offsets apart, duplicate the scalar code W times, and insert the results back into a vector.

However, we can do better. Before vectorizing, we perform a simple forward data flow analysis to infer invariants on the *shapes* of the vectorized variables. For example, if we can prove that the base address plus the offset used by a `gep` contains the same value in every instance, we do not need to duplicate the `gep` and a possibly following memory access instruction. In that case we can load the value as a scalar and *broadcast* it to a vector when needed.

A more common example are vector elements with consecutive values. Assume we have a base address and know that the values of the offsets are consecutive for consecutive instance IDs. Putting the offsets in a vector yields $\langle\, n, n+1, \ldots, n+W-1 \,\rangle$. Using such an offset vector in a `gep` gives a vector of consecutive addresses. Thus, the vector load and store instructions, which load the vector elements consecutively, can be used directly.

Our analysis tracks the following information for each variable: Does the variable contain consecutive values for consecutive instances (`c`) and is it aligned to the SIMD width (`ca`)? The latter allows to use the faster, aligned memory instructions. The former still avoids splitting but needs unaligned memory accesses.

Furthermore, it is also important if a variable contains the same value (`s`) for every instance and if this value is a multiple of the SIMD width (`sa`). Such a variable can be kept



Element	Property
c	consecutive
ca	consecutive aligned
s	same value
sa	same value aligned

(a) Hasse diagram of the lattice $\mathbb{L}$

Element	Shape of Vector	Example
c	$\langle\, n, n+1, , \ldots, n+W-1 \,\rangle$	$\langle\, 3,4,5,6 \,\rangle$
ca	$\langle\, m, m+1, \ldots, m+W-1 \,\rangle$	$\langle\, 0,1,2,3 \,\rangle$
s	$\langle\, x, x, \ldots, x \,\rangle$	$\langle\, 7,7,7,7 \,\rangle$
sa	$\langle\, m, m, \ldots, m \,\rangle$	$\langle\, 4,4,4,4 \,\rangle$

$n \in \texttt{int}$, $m = n \cdot W$, $x \in \{\texttt{int}, \texttt{float}\}$

(b) Concrete value examples

Figure 4. The vectorization analysis lattice $\mathbb{L}$. The lattice is lifted to a function space whose elements map variables to elements of $\mathbb{L}$. Note that our notation uses the join, not meet style, i.e. $\top$ is least informative.

scalar and broadcasted into a vector when needed. Furthermore, adding the same value to a vector containing consecutive values maintains consecutiveness (equally for alignment). If nothing can be said about the shape of a vector, the analysis value is $\top$. Figure 4 shows the corresponding lattice.

Figure 5 shows the transfer functions $[\![\cdot]\!]^\sharp : (\mathit{Vars} \to \mathbb{L}) \to (\mathit{Vars} \to \mathbb{L})$ of our analysis. The function a is the analysis information that maps variables to lattice elements. The notation $a \mid v \mapsto l$ stands for

$$\lambda w. \begin{cases} l & \text{if } v = w \\ a(w) & \text{otherwise} \end{cases}$$

The function `tid` produces values that are consecutive and aligned. This is because $\vec{f}$ does the work of W consecutive instances of f. Furthermore, the ID of the first instance is always divisible by W. Assigning constants creates `s` values; an integer constant divisible by W creates an `sa` value. Because the arguments to f are the same for every instance, `arg` returns `s` or `sa` in the case of pointer arguments. This implies that the caller of the data-parallel program has to align data to which pointers are passed on SIMD register boundaries. A load is `s` if the address is the same for all instances (`s` or `sa`). Additive and multiplicative operators propagate the information accordingly. The `phi`-operation joins the data flow values. All other operators only keep the same-value information if possible to avoid premature broadcasting.

Manual inspection yields that $[\![\cdot]\!]^\sharp$ is monotone. Hence, the usual fixpoint algorithms converge. The initial value on each program point is $\bot$.

Transfer functions for miscellaneous instructions:

$$\begin{aligned}
[\![v \leftarrow \texttt{tid}]\!]^\sharp a &= a \mid v \mapsto \texttt{ca} \\
[\![v \leftarrow x \in \texttt{float}]\!]^\sharp a &= a \mid v \mapsto \texttt{s} \\
[\![v \leftarrow \texttt{phi}(x,y)]\!]^\sharp a &= a \mid v \mapsto x \sqcup y \\
[\![v \leftarrow n \in \texttt{int}]\!]^\sharp a &= a \mid v \mapsto \begin{cases} \texttt{sa} & \text{if } n = mW,\ m \in \mathbb{N} \\ \texttt{s} & \text{otherwise} \end{cases} \\
[\![v \leftarrow \texttt{load}(d)]\!]^\sharp a &= a \mid v \mapsto \begin{cases} \texttt{s} & \text{if } f(d) \in \{\texttt{s}, \texttt{sa}\} \\ \top & \text{otherwise} \end{cases} \\
[\![v \leftarrow \texttt{arg}(i)]\!]^\sharp a &= a \mid v \mapsto \begin{cases} \texttt{sa} & \text{if } i\text{-th arg is pointer} \\ \texttt{s} & \text{otherwise} \end{cases}
\end{aligned}$$

Additive operator $\oplus \in \{\texttt{add}, \texttt{sub}, \texttt{gep}\}$:

$[\![v \leftarrow \oplus(x,y)]\!]^\sharp a = a \mid v \mapsto$

$a(x), a(y)$	sa	s	ca	c	$\top$
sa	sa	s	ca	c	$\top$
s		s	c	c	$\top$
ca			$\top$	$\top$	$\top$
c				$\top$	$\top$
$\top$					$\top$

Multiplication:

$[\![v \leftarrow \texttt{mul}(x,y)]\!]^\sharp a = a \mid v \mapsto$

$a(x), a(y)$	sa	s
sa	sa	sa
s		s

else $\top$

Other arithmetic and comparison operators:

$[\![v \leftarrow \texttt{op}(x,y)]\!]^\sharp a = a \mid v \mapsto$

$a(x), a(y)$	sa	s
sa	s	s
s		s

else $\top$

For the sake of simplicity, we skipped function values with $\bot$ arguments. They all yield $\bot$.

Figure 5. Transfer Functions.

4.3 Mask Generation

As already mentioned, control flow may diverge because a condition might be `true` for some scalar instances and `false` for others. Consequently, *all* code is executed. The explicit transfer of control is modeled by mask variables (short: masks, also often called *predicates*) on control-flow edges. If a mask of a CFG edge $B \to B'$ is set to `true` at position i, then the i-th instance of the code took the branch from B to B'. Thus, the mask denotes which elements in a vector contain valid data on the corresponding control-flow edge.

The edge masks implicitly define entry masks on blocks: The entry mask of a block is either the disjunction of the masks of all incoming edges or—in case of a loop header—a ϕ-function with incoming values from the loop's pre-header and latch. The masks of the control-flow edges leaving a block are given by the block entry mask and a potential

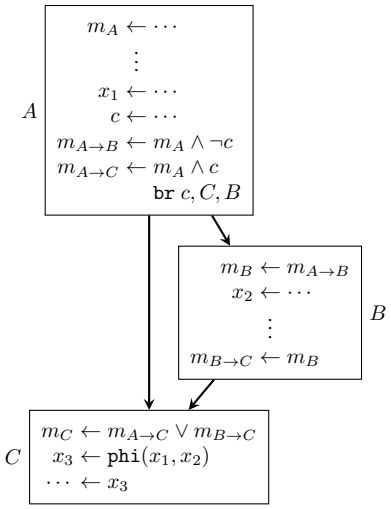


Figure 6. Edge and block entry masks. m_A, m_B, and m_C are the entry masks of the corresponding blocks A, B, and C. $m_{A\to B}$, $m_{A\to C}$, and $m_{B\to C}$ are the block exit masks connected to the edges $A \to B$, $A \to C$, and $B \to C$.

conditional. If a block exits with an unconditional branch, the mask of its single exit-edge is equal to the entry mask. If the exit branch is conditional, the exit mask of the "`true` edge" of the block is the conjunction of its entry mask and the branch condition. The exit mask of the "`false` edge" is the conjunction of the entry mask and the negated branch condition. Figure 6 shows three basic blocks A, B, C with corresponding block entry masks ($m_A, \dots$) and edge masks ($m_{A\to B}, \dots$).

Loop Masks. Each loop has to maintain a mask that is `true` for all instances that are still active in the loop. The loop can only be exited when this mask is `false` for *all* instances. Therefore, a special ϕ-function—the *loop mask phi*—is generated in the loop header (m_B in Figure 7). Its first incoming value is the mask of the incoming edge from the pre-header, the second value is the mask of the loop-backedge.

Loops with multiple exits require additional masks for *each* exit. These *loop-exit masks* store which instances left the loop over the corresponding edge and replace the former mask of that exit-edge. They are maintained by two instructions: a mask update operation (m_{exit}) and a ϕ-function in the loop header (m_{phi}). The update operation is the disjunction of the corresponding edge's mask and the ϕ-function. The ϕ-function has one incoming value from the pre-header and one from the latch. The value coming from the latch is the result of the update operation. The value coming from the pre-header is an empty mask (all elements set to `false`) if the loop is a top-level loop. If the loop is nested, it uses the corresponding loop mask ϕ-function of the parent loop.

After mask generation, each loop exit has exactly one update operation located right before the exit branch and one ϕ-function in the header of each loop that is left.

Finally, a *combined loop-exit mask* is required for select generation (see Section 4.4). This mask combines all information about instances that left the loop in the *current* iteration. In case of a loop that contains more nested loops, the current iteration of the parent includes all iterations of all nested loops. Thus, the combined loop-exit mask is a disjunction of all accumulated loop-exit masks of exits from nested loops and the exit conditions of exits from the current loop.

4.4 Select Generation

Linearization of control-flow is only possible if results of inactive instances are discarded. This is achieved by inserting *blend operations* at control-flow join points and loop latches. Each ϕ-function in the original CFG that is not in a loop header is replaced by a `select` instruction. ϕ-functions with n incoming values are transformed into series of $n - 1$ connected select instructions.

Loop Blending. Additionally, each loop requires *result vectors* in order to conserve the *loop live values* of instances that leave the loop early.

Loop live values are those values that are *live across loop boundaries*. A value is defined as *live across loop boundaries* if it is used either in a subsequent iteration or outside the loop.

For each loop live value, a result vector that is maintained by two instructions—a ϕ-function and an update operation—is introduced. The result update operation is a `select` which blends the result-ϕ-function and the corresponding live value depending on the combined loop-exit mask. This way, the result vector is updated only if one or more instances left the loop in the *current* iteration. All blend operations are performed in the loop latch (right before branching back to the loop header) and use the combined loop-exit mask as their condition. This enables us to only insert one select instruction for each loop live value per loop, regardless of the number of loop-exits. The result-ϕ-function is placed in the header of each nested loop and has two incoming values. The incoming value from the loop pre-header is undefined for top-level loops (there is no result until the loop has iterated at least once). For nested loops, the incoming value is the parent loop's result-ϕ-function of the current loop live value. The incoming value from the loop latch is the result update operation.

The usage of result vectors enables us to vectorize all kinds of loops. This especially includes control-flow with multiple nesting levels, multiple exits and edges exiting multiple loops. Figure 7 shows an example of mask and select operations in a loop.

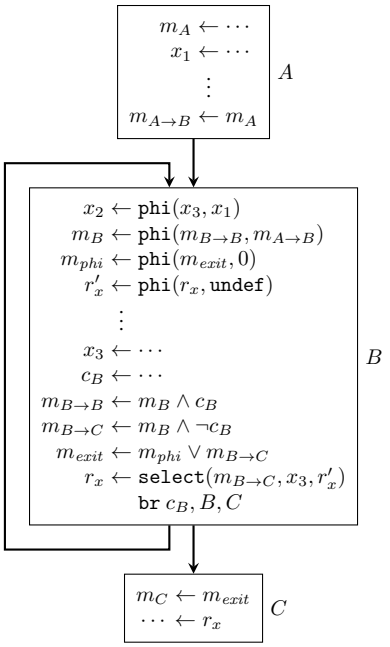


Figure 7. Mask and select generation for a loop. In general, each exit is assigned a mask operation m_{exit} and a ϕ-function m_{phi}. The mask operation updates the exit mask by setting elements of instances that leave the loop in the current iteration to `true`. The ϕ-function holds the current exit mask. Note that, after this pass, the mask of the edge $B \to C$ is m_{exit} instead of $m_{B\to C}$. The `select` r'_x in the latch and the ϕ-function r_x form the *result vector* of loop live value x. Each time an instance leaves the loop, the corresponding element of x is blended into the result vector.

4.5 CFG Linearization

After all mask and select operations are inserted, all control flow, except for loop backedges, is effectively encoded by data flow and can thus be removed. To this end, the basic blocks have to be put into a sequence that preserves the execution order of the original CFG G: If a block A executed before B in every possible execution of G, then A has to be in front of B in the flattened CFG G'. If the CFG splits up into two paths, one path is chosen to be executed entirely before the other.

The decision which path to execute first is currently nondeterministic. Employing a special heuristic instead is likely to improve the generated code. Such a heuristic can e.g. take into account the code size or register pressure of the different paths.

The ordering is determined by topologically sorting the blocks recursively over the loop tree of G. The result is a CFG that only has conditional branches remaining at loop exits and unconditional branches at loop entries. All other branches can be removed. Figure 8 shows the flattened CFG of the example in Figure 6.

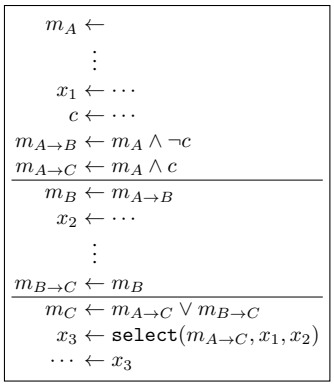


Figure 8. The flattened control-flow of Figure 6 with value blending.

Note that our vectorization analysis (Section 4.2) allows us to exclude regions of the CFG from linearization that are entirely marked as `s` ("same value"). For such regions, complete linearization is not likely to improve runtime performance. Analogously, we also do not have to generate select statements, so that such regions basically remain untouched.

For linearized regions, an optimization similar to *branch-on-superword-condition-codes (BOSCC)* [27] can be applied. Such a technique reintroduces branches after linearization to skip blocks of instructions if a mask is `true` or `false` for *all* instances.

4.6 Irreducible Control-Flow

If the CFG of a function is irreducible (e.g. if there is a loop that has more than one header), the commonly used technique for many program analysis algorithms is to apply node splitting [14] to transform the CFG into reducible code before the analysis. However, this can result in an exponential blowup of the code size [8].

Our algorithm is able to deal with irreducible control-flow without code duplication: During CFG linearization, one of the headers of an irreducible loop has to be chosen to be the *primary* header. This results in only the mask of the incoming edge of this header being updated in every iteration, entry masks from the other headers remain untouched. If control-flow during a later iteration reaches a join point with one of these headers, the incoming mask might falsely "reactivate" an instance that already left the loop.

In order to handle irreducible control-flow directly, we have to ensure that these masks are joined with the loop mask *in the first iteration only*. This is achieved by performing the blend operations at those join points with a modified mask: In the first iteration, the new active mask is given by a disjunction of the current active mask with the incoming mask from the other header. In all subsequent iterations, it is given by a disjunction with `false`, which means the current loop mask is not modified.

4.7 Instruction Vectorization

After linearization, the actual transformation into vector code is applied. Vectorizing a single instruction is basically a one-to-one translation from the scalar instruction to its SIMD counterpart. This holds for all instructions except for function calls and memory operations.

If f contains external calls that can not be vectorized, *native* vector-functions can be used. Such functions can either be built-in (such as vectorized variants of `sin`, `sqrt`, `floor`, etc.) or supplied by the user and can receive a mask if required.

If the current mask is not `true` for all instances, memory operations and calls to external, non-native functions have to be split into W guarded scalar operations. This is because we have to conservatively expect them to produce side effects that we do not want to occur for inactive instances.

For example, a shader might call a renderer to trace a ray (see Section 6) inside the `true`-part of an `if`-statement that is not executed by all instances. Similarly, we have to prevent the execution of a store operation for inactive instances. Thus, we have to guard each scalar execution by an `if`-construct that skips the instruction if the mask of that instance is `false`. Unfortunately, this involves a lot of extract-, insert-, and branch-operations that reduce the overall benefit of vectorization.

However, we can optimize such a store operation by generating a *load-blend-store* sequence of vector operations that is faster than an if-cascade with scalar stores as described above. Another way to circumvent this issue is hardware support of conditional load/store instructions as e.g. AVX will provide.

Additionally, any memory operation marked as $\top$ by our data-flow analysis (Section 4.2) has to be split independent of the mask information. This is by reason that we can not guarantee that the addresses of the different instances point to consecutive memory locations. If the operation is marked as `c`, we can still use unaligned memory access operations.

5. Related Work

Generating code for parallel hardware architectures is being studied since the emergence of vector computers and array processors. A lot of research went into parallelizing scientific (Fortran) programs; especially the analysis and automatic transformation of loop nests [3, 10]. Allen et al. [2] pioneered control flow to data flow conversion to help the dependence analyses to cope with more complex control structures. In our setting, we do not have to perform dependence analysis or find any parallelism; it is implicit in the programming model we consider. We use control flow to data flow conversion as a technique to *implement* data-parallel programs on SIMD processors. Furthermore, Allen et al. perform their transformation on the syntax tree level. We however consider control flow to data flow conversion on arbitrary control flow graphs in SSA form.

Another strain of work bases on explicit *instruction-level parallelism (ILP)* for *automatic vectorization*. There, inner loops are unrolled several times such that multiple instances of the same instruction are generated and vectorized. Several authors discuss the issues of this approach with SIMD instruction sets [9, 16, 20, 29]. Since those techniques only consider inner loops, they only vectorize acyclic code regions.

In general, the control-flow conversion of Allen et al. is very similar to our mask generation pass, but it only targets vector machines that support *predicated execution* [23]. Predicated execution is a hardware feature that performs implicit blending of results of operations. For machines without predication, we are the first to show how masking of arbitrary control flow can be implemented using blend operations.

Superword-level parallelism (SLP) [17] describes the occurrence of independent isomorphic statements (statements performing the same operations in the same order) inside a basic block, independent of loops. Such statements can be combined to SIMD instructions similar to instructions unrolled inside loops. Shin [28] extended the approach to also work in the presence of control-flow by using predicates. Unfortunately, this technique introduces overhead for the packing and unpacking of vectors that makes the approach unusable for smaller fractions of code. Also, it is restricted to code-regions without loops.

Our approach can be seen as a generalization of *outer-loop vectorization (OLV)* [19, 21, 25, 30] that unrolls outer instead of inner loops. However, OLV does not allow for diverging control flow inside the outer loop in contrast to our algorithm.

On the language side, there are many different *data-parallel languages* that automatically compile to parallel and/or vector code, like NESL [7], Ct [13], and CGiS [11]. Modern, GPGPU-oriented languages like CUDA [22] or OpenCL [15] execute code in SIMT (Single Instruction, Multiple Threads) fashion. On a GPU, a thread roughly corresponds to an element in a vector register. To our knowledge, no existing CPU driver for these languages currently employs a technique similar to whole-function vectorization. They solely rely on multi-threading to implement parallelism.

In graphics, there is a number of different *domain-specific languages (DSLs)* that allow the user to program scalar code that is executed in parallel for different input data. RTSL [24] for example lets the user write *shaders* (i.e. functions that describe material properties) in scalar code that is automatically transformed to vector code usable by packet ray tracing systems.

Our model of computation is inspired by the GPGPU-style languages. However, our pass comes so late in the compilation phase (immediately before machine-dependent code generation tasks) that the source language's influence is

negligible. We argue that all languages mentioned here can be mapped to the language core presented in Section 3 and thus can profit from our algorithm. In the following section, we show this for two languages: RenderMan and OpenCL.

6. Experimental Evaluation

We implemented the algorithm presented in this paper in the LLVM compiler framework and evaluated the runtime of the vectorized programs in two real-world scenarios: shading in real-time ray tracing and a custom OpenCL driver. All experiments were conducted on a Core 2 Quad at 2.8 GHz with 4 GB of RAM. The vector instruction set is Intel's SSE 4.1 with 4 `floats`/`ints` per register. The machine ran in 64 bit mode, thus 16 vector registers were available.

6.1 Vectorized Shaders for Ray Tracing

We integrated our vectorizer into a shading-language compiler for the real-time ray tracer *RTfact* [12] that uses LLVM to compile RenderMan shaders to x86 machine code. Shaders are well-suited for whole-function vectorization. They are very compute-intensive and only perform aligned and consecutive memory accesses. Hence, there is almost no overhead due to splitting vectors.

Table 1 shows the performance of each shader applied to a sphere lit by two point light sources at a resolution of 512×512 pixels. We compare the rendering performance of automatically vectorized shaders against scalar versions of the shader that are executed sequentially.

Ray Tracing Performance (fps)

Shader	Native	Scalar	Vectorized	Speedup
Brick	-	8.8	31.4	3.6x
Checker	34.5	8.8	31.8	3.6x
Glass	-	0.9	4.5	5.0x
Granite	-	7.2	24.6	3.4x
Parquet	-	4.3	18.6	4.3x
Phong	35.5	14.1	32.5	2.3x
Screen	-	4.6	22.7	4.9x
Starball	-	4.5	20.0	4.4x
Venus	-	7.6	25.7	3.4x
Wood	-	4.4	19.1	4.3x
Average	-	6.5	23.3	3.9x

Table 1. Ray tracer performance for ten different shaders [6] in frames per second. "Native" refers to highly optimized, internal SIMD shaders of the renderer.

The vectorized versions of the shaders outperform their scalar counterparts by an average factor of 3.9. For some shaders we observe super-linear speedups up to 5.0. This has two reasons: First, the ray tracer internally also works with vectors and needs to split them in the case of *sequential* shading. This overhead is not there in vectorized shading. Second, the data layout used by the ray tracer enables for a better cache locality in the vectorized case. The whole transformation and compilation process (including LLVM's JIT) takes less than 100ms for any shader. This allows for dynamic recompilation at the runtime of the renderer which is important in graphics. Compared to highly optimized SIMD shaders directly integrated into the renderer we are within 10% of the best possible performance.

6.2 OpenCL

In the second part of our evaluation we built a custom OpenCL [15] driver. We implemented a sufficiently complete fraction of the OpenCL API to run a set of diverse test applications. The driver internally uses LLVM for OpenCL-specific runtime code modification, vectorization, and code generation.

Vectorization. Enabling whole-function vectorization in OpenCL requires several program transformations, which we will briefly describe here. A more detailed description of the driver is beyond the scope of this paper.

If the application uses more than one dimension for its input data, the driver has to choose one *SIMD dimension* for vectorization. Depending on the chosen dimension, the kernel has to be adjusted to the modified data size which is now divided by the SIMD width W. Our driver currently always uses the first dimension, but a heuristic that analyses memory accesses via indices given by `get_global_id()` of the different dimensions could by applied instead.

The results of our data-flow analysis (Section 4.2) allow us to optimize frequent cases where consecutive and aligned elements of arrays are accessed using the local ID incremented by a multiple of the local size (which is divisible by W).

In order to support OpenCL's `barrier()`-statement we implemented a synchronization scheme inside our driver that is not thread-based but uses function splitting in a coroutine style. Synchronization is non-trivial because kernels are allowed to synchronize in-between loop iterations, which requires all threads of a group to execute loop iterations in lock-step. Our system avoids costly callbacks to the driver or the operating system and even enables inter-optimization of the required code with the kernel itself.

Benchmarks. All the benchmark applications except for AOBench [1] are taken directly from AMD's Stream Software Development Kit [5]. AOBench is a minimal ambient occlusion ray tracer. The SDK serves as the basis for all of our measurements: it is compiled with our OpenCL driver instead of AMD's. We further employ AMD's `clc` tool to generate LLVM bitcode files from OpenCL code.

Table 2 shows the runtime performance of a diverse set of applications ranging from compute-intensive kernels (e.g. BlackScholes) to kernels that are dominated by memory access operations (e.g. Histogram). This demonstrates that the vectorizer can handle real-world applications, but it also shows its limitations: Other than shaders, OpenCL kernels

OpenCL Kernel Performance (milliseconds)

Application	Input Size	AMD (4)	Scalar (1)	Scalar (4)	Vectorized (1)	Vectorized (4)	Speedup
AOBench	$1{,}024^2$	5880	37037	10000	24390	6250	1.5x
BlackScholes	65,536	70	13	3.7	2.4	0.8	5.2x
FastWalshTransform	1,048,576	80	80	33	100	30	0.8x
Histogram	$8{,}192^2$	120	410	310	710	430	0.6x
Mandelbrot	$8{,}192^2$	72800	4000	1700	1800	800	2.2x
NBody	4,096	50	160	50	57	16	2.8x
MatrixTranspose	$10{,}000^2$	4400	1220	470	900	340	1.4x

Table 2. Average kernel execution times of our OpenCL driver for different applications (100 iterations). Parentheses denote the number of threads used, the speedup refers to scalar against vectorized mode without multi-threading. The column labelled "AMD" denotes the performance of AMD's proprietary OpenCL CPU driver.

frequently modify arrays in a non-consecutive way, which incurs large penalties for splitting vectors and sequential loads and stores. Consequently, benchmarks like FastWalshTransform and Histogram, where memory access dominates computation, even slow down with vectorization enabled. On the other hand, vectorization can significantly improve the performance of suitable applications like NBody or Mandelbrot: These benchmarks are very compute-intensive and have no complicated control-flow and no scattering memory accesses patterns. The BlackScholes benchmark additionally is almost branch-free, its memory footprint is minimal, and almost no spilling occurs. This accumulates into a speedup factor of 5.2.

The wide range of observed speedups motivates the development of a heuristic that applies vectorization only to code regions where it seems beneficial. We leave this for future work.

To be able to compare to AMD's reference driver we implemented a naive, unoptimized multi-threading scheme via OpenMP. We do not have any information about AMD's implementation except that it is multi-threaded and also uses LLVM. Nevertheless, note that our custom driver significantly outperforms their driver in all test-cases except Histogram and AOBench. The Mandelbrot benchmark was rewritten in SDK version 2.2 using OpenCL's `float4` SIMD datatypes in manually unrolled loops. Although performance improved by over twelve times (6 seconds), the code is hardly readable anymore and much more cumbersome to write (223 compared to 28 lines of code). More importantly, our driver still outperforms it by a factor of 7.5 without putting that strain on the programmer.

7. Conclusion

In this paper, we discussed the implementation of data-parallel programs on machines with SIMD instruction sets. We presented an algorithm that vectorizes a function given by an arbitrary control flow graph in SSA form. The algorithm is based on control to data flow conversion and generates efficient blending code for architectures without hardware support for predicated execution.

We furthermore presented a data-flow analysis that determines which code regions have constraints for vectorization concerning alignment and consecutiveness. The analysis yields two main optimizations: First, we can prevent costly vector splits that are due to the limitations of memory access instructions of current SIMD instruction set architectures. Second, our vectorization algorithm can ignore code regions that are proven invariant for all instances of the program, which makes any reconstruction steps unnecessary.

Case studies showcased the applicability of the system: Integrating our technique into a real-time ray tracer provides a linear average speedup of 3.9 over the scalar version. The speedup of an enhanced OpenCL framework depends largely on the target kernels and varies between 0.6 and 5.2.

Acknowledgments

Ralf Karrenberg is supported by the Cluster of Excellence on Multimodal Computing and Interaction at Saarland University. This project is supported by the Intel Visual Computing Institute Saarbrücken. The authors would like to thank Roland Leißa, Dmitri Rubinstein, and Philipp Slusallek for many interesting and insightful discussions. Furthermore, we thank Daniel Grund, Mark Lacey, Ingo Wald, Sven Woop and the anonymous reviewers for their helpful comments and remarks.

References

[1] AOBench. http://lucille.atso-net.jp/blog.

[2] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL*, pages 177–189. ACM, 1983.

[3] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, 1987.

[4] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL*, pages 1–11. ACM, 1988.

[5] AMD-ATI. ATI Stream Software Development Kit v2.1. http://developer.amd.com/gpu/atistreamsdk, March 2010.

[6] A. Apodaca and M. Mantle. RenderMan: Pursuing the Future of Graphics. *IEEE Computer Graphics & Applications*, 10(4):44–49, July 1990.

[7] Guy E. Blelloch et al. Implementation of a portable nested data-parallel language. In *PPOPP*, pages 102–111. ACM, 1993.

[8] Larry Carter, Jeanne Ferrante, and Clark Thomborson. Folklore confirmed: reducible flow graphs are exponentially larger. In *POPL*, pages 106–114. ACM, 2003.

[9] Gerald Cheong and Monica Lam. An Optimizer for Multimedia Instruction Sets. In *Second SUIF Compiler Workshop*, 1997.

[10] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000.

[11] Nicolas Fritz, Philipp Lucas, and Philipp Slusallek. CGiS, a New Language for Data-Parallel GPU Programming. In *VMV*, pages 241–248, 2004.

[12] Iliyan Georgiev and Philipp Slusallek. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2008*, pages 115–122, August 2008.

[13] Anwar Ghuloum et al. Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture. *Intel Technology Journal*, 11(04), November 2007.

[14] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031–1052, 1997.

[15] Khronos Group. *OpenCL 1.0 Specification*, 2009.

[16] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *Int. J. Parallel Program.*, 28(4):347–361, 2000.

[17] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *SIGPLAN Not.*, 35(5):145–156, 2000.

[18] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, Mar 2004.

[19] Viet Ngo. *Parallel loop transformation techniques for vector-based multiprocessor systems*. PhD thesis, May 1994.

[20] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *CGO*, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: revisited for short simd architectures. In *PACT*, pages 2–11. ACM, 2008.

[22] NVIDIA. *CUDA Programming Guide*, 2009.

[23] Joseph C. H. Park and Mike Schlansker. On Predicated Execution, 1991.

[24] Steven Parker, Solomon Boulos, James Bigler, and Austin Robison. RTSL: A Ray Tracing Shading Language. *IEEE Symposium on Interactive Ray Tracing*, 2007.

[25] Randolf G Scarborough and Harwood G Kolsky. A vectorizing fortran compiler. *IBM J. Res. Dev.*, 30(2):163–171, 1986.

[26] Larry Seiler et al. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH*, pages 1–15. ACM, 2008.

[27] Jaewook Shin. Introducing Control Flow into Vectorized Code. In *PACT*, pages 280–291. IEEE Computer Society, 2007.

[28] Jaewook Shin, Mary Hall, and Jacqueline Chame. Superword-Level Parallelism in the Presence of Control Flow. In *CGO*, pages 165–175. IEEE Computer Society, 2005.

[29] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *Int. J. Parallel Program.*, 28(4):363–400, 2000.

[30] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.