Improving Performance of OpenCL on CPUs

Ralf Karrenberg and Sebastian Hack

Saarland University, Germany
{karrenberg,hack}@cdl.uni-saarland.de

Abstract. Data-parallel languages like OpenCL and CUDA are an important means to exploit the computational power of today's computing devices. In this paper, we deal with two aspects of implementing such languages on CPUs: First, we present a static analysis and an accompanying optimization to exclude code regions from control-flow to data-flow conversion, which is the commonly used technique to leverage vector instruction sets. Second, we present a novel technique to implement barrier synchronization. We evaluate our techniques in a custom OpenCL CPU driver which is compared to itself in different configurations and to proprietary implementations by AMD and Intel. We achieve an average speedup factor of 1.21 compared to naïve vectorization and additional factors of 1.15–2.09 for suited kernels due to the optimizations enabled by our analysis. Our best configuration achieves an average speedup factor of over 2.5 against the Intel driver.

Keywords: OpenCL, SIMD, Vectorization, Data Parallelism, Code Generation, Synchronization, Divergent Control Flow.

1 Introduction

In this paper, we present two techniques to speed up data-parallel programs on machines with explicit SIMD operations (e.g. current CPUs). Although we focus on OpenCL in this paper, the presented techniques are also applicable to similar languages like CUDA. A data-parallel program is written in a scalar style. It is then executed in *n instances* (sometimes called threads, however this is not to be confused with an operating system thread) on a computing device. To a certain extent, the order of execution among all instances of the program is unspecified to allow for parallel or sequential execution as well as a mixture of both. Every instance is identified with a *thread ID* which is called `tid` in the following. Usually, the data-parallel program uses the `tid` to index data. Hence, every instance can process a different data item.

Since data-parallel semantics explicitly do not define an order of the instances, languages like OpenCL lend themselves to vector processing. In this paper, we consider machines that support SIMD instruction sets such as Intel's SSE and AVX, or ARM's NEON. SIMD instructions operate on a vector of W data items where W is the SIMD width (e.g. 4 32 bit values for SSE, 8 for AVX). To implement a data-parallel program on such a processor, one creates a vector

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 1–20, 2012.

program that executes W instances of the original program in parallel. The challenge in this setting is divergent control flow: at a conditional jump, it might be the case that instance i takes the branch, while instance j does not. Hence, the vector program must accommodate both situations. Usually this is solved by control-flow to data-flow conversion [1] where branches are replaced by predicate variables which express the control condition. Then, control flow is *linearized,* i.e. the vector program executes every instruction of the program and masks out the inactive computations using the predicates. The latter happens either by explicit *blending* of the values or by instruction predication provided by the hardware.

However, applying control-flow linearization naïvely leaves some potential unexploited: in many data-parallel programs, several branches do not diverge because they depend on values which are the same (*uniform*) for all instances. Consider the example in Figure 1:

```
__kernel void foo(int* data, int x) {
  /* A */ ...
  /* B */ if (expr(x))
  /* C */   data[tid] = ...;
            ...
          else
  /* D */   ...
  /* E */ ...
}
```

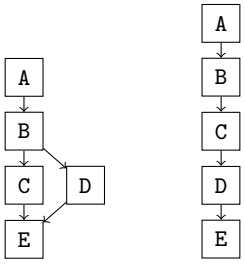

Fig. 1. An example kernel, the control-flow graph of its scalar version, and one possible linearization of the vector version after applying control-flow to data-flow conversion

The condition `expr(x)` only depends on uniform values.[1] Hence, the branch need not be linearized because either all instances take it or none. In addition, the evaluation of the condition can be hoisted outside the kernel. On the other hand, the computations inside the branch cannot be hoisted because they depend on the variable `tid` which is *not* uniform across all instances.

Previous work [22] injects code into the *linearized* vector program to *dynamically* test whether all instances evaluate a condition to the same value. If so, the corresponding code part can be bypassed by a branch, trading a reduction of the amount of executed code for some overhead for the dynamic test. While this reduces the number of instructions executed, it cannot compensate another drawback of control-flow to data-flow conversion: the increase of register pressure on architectures that do not support predication. Reconsider the example in Figure 1. When control flow is linearized, every variable that is live at the entrance of `D` will be live throughout `C`. Correspondingly, all variables live out at `C` will live throughout `D`. Our experiments have shown that the increase in register pressure often causes more spilling and reloading which deteriorates

[1] Note that the arguments passed to the kernel are the same for every instance, hence they are uniform.

performance. Keeping control flow for this code part prevents the increase of register pressure.

In this paper, we present a static analysis to identify branches that will never diverge and a code transformation that exploits these results to avoid control-flow linearization whenever possible. The result is a vector program in which only code parts of which we could not prove non-divergence are linearized.

Another important feature of languages like OpenCL is barrier synchronization. A kernel can use a *barrier* statement to enforce that no instance of a thread group executes the statement following the barrier before all instances of that group have reached the barrier. GPUs have dedicated hardware support to implement barrier synchronization. On CPUs, barriers need to be implemented in software. Simple implementations use the support of the runtime system and the operating system [24] which boils down to saving and restoring the complete state of the machine. More sophisticated techniques use loop fission on the abstract syntax tree to decompose the kernel into separate pieces that are executed in a way such that all barriers are respected [25]. However, this technique potentially introduces more synchronization points than needed. In this paper, we generalize the latter approach to work on control-flow graphs (CFGs, instead of abstract syntax trees) while not increasing the amount of synchronization points.

1.1 Contributions

To summarize, this paper makes three main contributions:

1. We present a static analysis that identifies blocks in the CFG of a data-parallel program that do not *diverge.* For these blocks, control flow can be retained and need not be replaced by data flow.
2. We present an SSA-based linearization algorithm that leverages the divergence analysis by retaining control flow, even in presence of irregular and arbitrarily nested control-flow structures (e.g. loops with multiple exits or jumps that exit multiple loops).
3. We present a novel technique to implement barrier synchronization for data-parallel languages on CPUs that splits the function into continuations. This does not require costly interaction with the OS, prevents introduction of additional synchronization points, and reduces overhead by only saving the live values.

1.2 Structure of This Paper

In the next section, we give an overview of our OpenCL driver and present our implementation of barrier synchronization. Section 3 describes the divergence analysis and how it can be used to increase the efficiency of vectorized kernels. Section 4 discusses related work and Section 5 presents our experimental evaluation.

2 OpenCL Driver Implementation

In this section, we describe code-generation techniques to improve the efficiency of an OpenCL driver. The compilation scheme of our driver looks like this:

1. Perform SIMD vectorization → Section 2.1
2. Implement barriers → Section 2.3
3. Create loops over local instances → Sections 2.2, 2.3
4. Remove API callbacks (such as `get_global_id()`) → Section 2.2
5. Create wrapper for driver interface

The interface wrapper allows the driver to call the kernel with a static signature that receives only a pointer to a structure with all parameters. Pseudo-code for the modified kernel is shown in Figure 2 (before inlining and the callback optimizations described in Section 2.2).

2.1 SIMD Vectorization

We employ a modified algorithm for "Whole-Function Vectorization" (WFV) [12] to exploit SIMD instruction sets by transforming the kernel such that it computes W instances of the original kernel.

Enabling WFV in OpenCL can be summarized as follows: Vectorization is based upon changing the callback functions `get_global_id()` and `get_local_id()` to return a vector of the W IDs whose instances are executed by the vectorized kernel. From there, all uses of these indices are vectorized, mask and blend operations are created as necessary, and control flow is linearized [12]. However, we enhance these phases by additional analyses and optimizations that are described in Section 3.

2.2 Runtime Callbacks

OpenCL allows the user to organize threads in multiple dimensions (each thread is identified by an n-tuple of IDs for n dimensions). Furthermore, it allows to create groups of threads that are executed together and can be synchronized (see Section 2.3).

Given a kernel and a global number of threads $N_x \times N_y$ organized in a two-dimensional grid with groups of size $G_x \times G_y$, the driver is responsible for calling the kernel $N_x \times N_y$ times and for making sure that calls to `get_local_id()` etc. return the appropriate thread ID of the given dimension. The most natural iteration scheme for this employs nested "outer" loops that iterate over the number of groups of each dimension (N_x/G_x and N_y/G_y) and nested "inner" loops that iterate over the size of each group (G_x and G_y). Consider Figure 2 for some pseudo-code.

If the application uses more than one dimension for its input data, the driver has to choose one *SIMD dimension* for vectorization. This means that only

```
clEnqueueNDRangeKernel(Kernel kernelWrapper, TA arg_struct,
        int* globalSizes, int* localSizes) {
    int iter_0 = globalSizes[0] / localSizes[0];
    int iter_1 = globalSizes[1] / localSizes[1];
    for (int i=0; i<iter_0; ++i) {
        for (int j=0; j<iter_1; ++j) {
            int groupIDs[2] = { i, j };
            kernelWrapper(arg_struct, groupIDs, globalSizes, localSizes);
}   }   }

void kernelWrapper(TA arg_struct, int* groupIDs,
        int* globalSizes, int* localSizes) {
    T0      param0 = arg_struct.p0;
    ...
    TN      paramN = arg_struct.pN;
    int     base0  = groupIDs[0] * localSizes[0];
    int     base1  = groupIDs[1] * localSizes[1];
    __m128i base0V = <base0, base0, base0, base0>;
    for (int i=0; i<localSizes[1]; ++i) {
        int lid1 = i;                          // local id   (dim 1)
        int tid1 = base1 + lid1;               // global id  (dim 1)
        for (int j=0; j<localSizes[0]; j+=4) {
            __m128i lid0 = <j, j+1, j+2, j+3>; // local ids  (dim 0)
            __m128i tid0 = base0V + lid0;      // global ids (dim 0)
            simdKernel(param0, ..., paramN, lid0, lid1, tid0, tid1,
                       groupIDs, globalSizes, localSizes);
}   }   }
```

Fig. 2. Pseudo-code implementation of clEnqueueNDRangeKernel and the kernel wrapper before inlining and optimization (2D case, $W = 4$). The outer loops iterate over the number of groups, which can easily be parallelized across multiple threads. The inner loops iterate over all instances of a group (step size 4 for the SIMD dimension 0).

queries for instance IDs of this dimension will return a vector, queries for other dimensions return a single ID. Because it is the natural choice for the kernels we have analyzed so far, our driver currently always uses the first dimension. However, it would be easy to implement a heuristic that chooses the best dimension, e.g. by comparing the number of memory operations that can be vectorized in either case. The inner loop that iterates over the dimension chosen for vectorization is incremented by W in each iteration as depicted in Figure 2.

We automatically generate a wrapper around the original kernel that includes the inner loops while only the outer loops are implemented directly in the driver (to allow multi-threading, e.g. via OpenMP). This allows us to remove all overhead of the callback functions: All these calls query information that is either statically fixed (e.g. `get_global_size()`) or only depends on the state of the inner loop's iteration (e.g. for one dimension, `get_global_id()` is the local size multiplied with the group ID plus the local ID). The static values are supplied as arguments to the wrapper, the others are computed directly in the inner loops. After the original kernel has been inlined into the wrapper, we can remove all overhead of callbacks to the driver by replacing each call by a direct access to a value. Generation of the inner loops "behind" the driver-kernel barrier also exposes additional optimization potential of the kernel code together with the

surrounding loops and the callback values. For example, loop-invariant code motion moves computations that only depend on group IDs out of the innermost loop (reconsider the example in Figure 1).

2.3 Continuation-Based Synchronization

OpenCL provides the `barrier()` statement to implement barrier synchronization of all threads in a group. A barrier enforces all threads of the group to reach it before the threads in the group can continue executing instructions behind the barrier. This means that the current context of a thread needs to be saved when it reaches the barrier and restored when it continues execution. Instead of relying on costly interaction with the operating system, we use the following code transformation to implement barrier synchronization.

Let the set $\{b_1, \ldots, b_n\}$ be the set of all barriers in the kernel. We apply the following recursive scheme: From the start node of the CFG, start a depth-first search (DFS) which does not traverse barriers. All nodes reached by this DFS are by construction barrier free. The search furthermore returns a set of barriers $B = \{b_{i_1}, \ldots, b_{i_m}\}$ which it hit. At each hit barrier, we determine the live variables and generate code to store them into a structure. For every instance in the group, such a structure is allocated by the driver. The last instruction generated is a return with the ID of the hit barrier. Now, the instructions $b_{i_1}, \ldots, b_{i_m}$ are taken as start points for m different kernels. For each one, we apply the same scheme until there are no more kernels containing barriers. Figure 3 gives an example for this transformation.

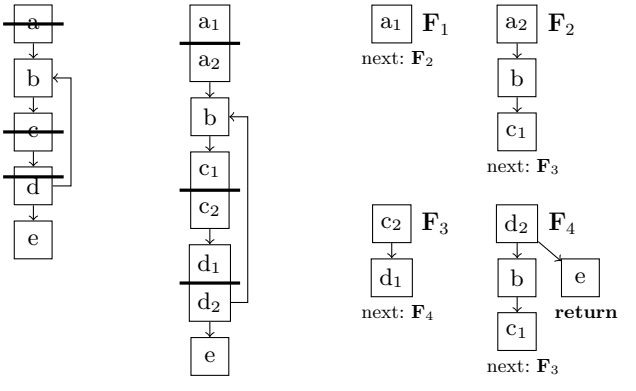

Fig. 3. Example CFG of a kernel which requires synchronization (the barriers are indicated by the bars crossing the blocks), the CFG after splitting blocks with barriers, and the resulting set of new functions $\{F_1, \ldots, F_4\}$

Then, we generate a wrapper that switches over the generated functions dependent on the last returned barrier ID (see Figure 4).

```
void newKernel(T0 param0, ..., TN paramN, int localSize, ...) {
    void* data[localSize/W] = alloc((localSize/W) * liveValSize);
    int next = BARRIER_BEGIN;
    while (true) {
        switch (next) {
            case BARRIER_BEGIN:
                for (int i=0; i<localSize; i+=W)
                    next = F1(param0, ..., paramN, tid, ..., &data[i/W]);
                break;
            case B2:
                for (int i=0; i<localSize; i+=W)
                    next = F2(tid, ..., &data[i/W]);
                break;
            ...
            case B4:
                for (int i=0; i<localSize; i+=W)
                    next = F4(tid, ..., &data[i/W]);
                break;
            case BARRIER_END: return;
}   }   }
```

Fig. 4. Pseudo code for the kernel of Figure 3 after implementation of barriers and before inlining and optimization (1D, computations of `tid` etc. are omitted). The value of `liveValSize` is the maximum size required for any continuation, `data` is the storage space for the live variables of all instances.

Note that the semantics of OpenCL require all instances to hit the *same* barrier, otherwise the program's behavior is undefined. Hence, if not all instances return the same ID, the kernel is in a bad state anyways, so we simply use the ID returned by the last instance.

3 Exploiting Uniform Computations

In this section, we describe our main contribution, a static analysis of control-flow divergence and its application in the context of whole-function vectorization. The analysis is based on a value analysis presented by Karrenberg and Hack [12] which identifies *uniform* computations.

3.1 Uniform Value Analysis

The uniform value analysis determines whether an operation produces the same value for all instances of a kernel. If a value is not uniform, we call it *varying*. If a branch depends on a varying condition, we call it a *varying branch*. Input arguments to OpenCL kernels are always uniform because all instances are called with the same arguments. The OpenCL functions `get_global_id()` and `get_local_id()` produce varying values if called with the SIMD dimension as parameter. If called with another dimension, they produce uniform values because we only vectorize one dimension (see Section 2.2).

3.2 Divergence Analysis

As described in the introduction, our goal is to retain as much control flow during vectorization as possible. To exclude a certain code part from control-flow to data-flow conversion, we must prove that the instances implemented by the vector program never diverge in this code part. In general, this is a dynamic property that can change for different kernel inputs. The following definition describes an overapproximation of *divergence* which can be statically proven.

Definition 1 (Static Divergence). *Let b be a block that can be reached from another block v that ends with a varying branch. b is marked as divergent if*

1. *b is a direct successor of v, or*
2. *there exist two* disjoint *paths from v to b, or*
3. *b is an exit block of a loop which includes v, and there exist two* disjoint *paths from v to b and from v to the loop's latch ℓ.*[2]

Figure 5 illustrates these conditions, Figures 6 and 7 depict some more involved examples.

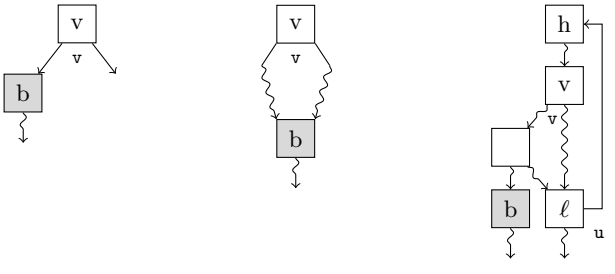

Fig. 5. Illustration of the three possibilities for divergence of a block b (Definition 1). Note that in the second case, b is neither required to post-dominate v, nor is v required to dominate b.

Informally, the second condition means that in the kernel execution under consideration, b can be reached by some instances from *both* edges that leave v. Hence, b is subject to control-flow to data-flow conversion.

The third condition is required because loops behave differently in terms of divergence: If b is a loop exit and v is inside the loop, there might be a path from one edge of v to an exit block and another, disjoint path from the other edge to a back edge. Even if all exit branches are uniform, this would still make it possible that some instances are still active in the loop when an exit edge is taken. Therefore, b is divergent under this condition.

[2] We assume that every loop has a latch which is the only basic block in the loop that has a back edge to the header.

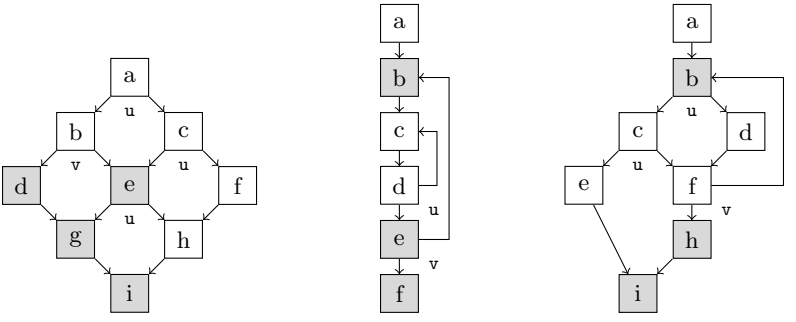

Fig. 6. Example CFGs showing our analysis results. Uniformity of conditional branches is shown with a lowercase letter below the block, divergent blocks are shaded. Our analysis determines that significant parts of these CFGs are non-divergent and therefore do not have to be linearized (see Figure 10).

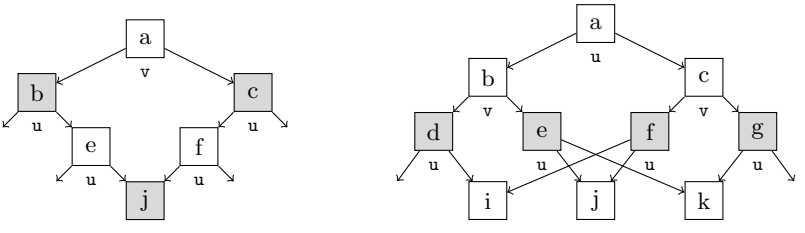

Fig. 7. More complex examples. In the left CFG, j is neither always executed by all instances that were active in a nor is it only executed by instances that took the left *or* right branch in a. Therefore, the linearized CFG has to make sure that j has to be executed regardless of the branch decision in a. In the right CFG, i, j, and k are non-divergent because none can be reached from both edges of the same varying branch. However, linearization requires duplication (see Section 3.3).

We compute uniformity and divergence using a data-flow analysis. Both properties influence each other mutually: First, as can be seen from Definition 1, divergence depends on uniformity. If a branch is labelled varying, control flow diverges. Second, divergence also influences uniformity. Consider a ϕ-function over uniform arguments. If that ϕ-function resides in a divergent block, the ϕ's value is *not* uniform itself, because not all instances enter the ϕ's block from the same predecessor in every execution. However, if we can prove the ϕ's block non-divergent, the ϕ's value is uniform, too. Hence, the data-flow analysis computes divergence and uniformity together.

The analysis uses join (not meet) lattices and employs the common perspective that instructions reside on edges not nodes. Program points thus sit between the instructions (see Figure 8). This has the advantage that the join and the update of the flow facts are cleanly separated.

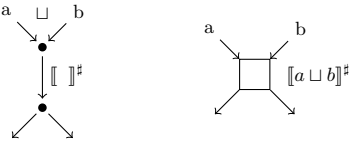

Fig. 8. Left: Our analysis setup with separated join and update of flow facts. Right: Classic setup with mixed join/update.

The analysis lattice uses two simple lattices for uniformity and divergence:

$$\begin{array}{lcll} \text{varying} & \mathtt{v} & \mathtt{d} & \text{divergent} \\ & | & | & \\ \text{uniform} & \mathtt{u} & \mathtt{n} & \text{non-divergent} \\ & \mathbb{U} & \mathbb{D} & \end{array}$$

The analysis lattice itself is defined as

$$\mathbb{L} := \mathcal{P}(V \times V) \times (V \to \mathbb{U}) \times \mathbb{D}$$

We record in every flow fact a set A of control-flow edges, a mapping u from variables to uniformity information, and information about divergence d of that program point. The join is defined component-wise:

$$(A, u, d) \sqcup (A', u', d') = (A \cup A', u \sqcup_{\mathbb{U}} u', \lambda w.\, (d(w) \sqcup_{\mathbb{D}} d'(w)))$$

For a program point x, the update function

$$(A', u', d') = [\![w]\!]^\sharp(A, u, d), \qquad [\![\cdot]\!]^\sharp : \mathbb{L} \to \mathbb{L}$$

is defined as follows. First, consider the divergence update which reflects Definition 1:

$$d' = \begin{cases} \mathtt{d} & \text{if } \exists p \in \mathit{pred}(x).\, u(p) = \mathtt{v} \\ \mathtt{d} & \text{if } \exists p_1, p_2 \in \mathit{pred}(x).\, \mathit{divergedPaths}_A(p_1, p_2) \\ \mathtt{d} & \text{if } x \in E_L \wedge \mathit{divergedPaths}_A(x, \ell_L) \\ \mathtt{n} & \text{otherwise} \end{cases}$$

where E_L is the set of program points behind the exits of a loop L and ℓ_L is the program point at the loop's back edge. Further, $\mathit{pred}(x)$ is the set of predecessor program points of x. The Information whether two paths are disjoint and divergent is given by the helper function

$$\mathit{divergedPaths}_A(p_1, p_2) = \exists e_t, e_f \in A(p_1) \cup A(p_2).\, (e_t \notin A(p_1) \vee e_f \notin A(p_2)).$$

divergedPaths uses the set A which captures information about edges leaving blocks with varying branch conditions. The control-flow edge is inserted into A if the branch of a block was detected varying by the analysis. To this end, every

branch instruction gets two extra program points (one for the `true` and one for the `false` program point) on which transfer functions can be placed that add the corresponding edges to A. For each branch node v, let v_t and v_f be these program points. For some $v_\square$, the update function for A is

$$A' = A \cup \begin{cases} \{v_\square\} & \text{if } u(v) = \mathtt{v} \\ \emptyset & \text{otherwise,} \end{cases}$$

otherwise we define $A' = A$.

Finally, the uniformity part provides information to the other two components of the analysis. A common operation $s = \oplus(o_1, \ldots, o_n)$ is uniform if all operands o_i are uniform. Thus, using $f \mid x \mapsto y$ as an abbreviation for

$$\lambda w. \begin{cases} y & \text{if } w = x \\ f(w) & \text{otherwise,} \end{cases}$$

the uniformity update for non-ϕ instructions is given by:

$$u' = u \mid s \mapsto \bigsqcup_i u(o_i)$$

For ϕ-functions, this does not hold. Even if all parameters of a ϕ are uniform, the ϕ will *not* produce a uniform value if two instances can enter it via *different* predecessors. Hence, to produce a uniform value, the ϕ-function's program point has to be detected non-divergent.

$$u'_\phi = u \mid s \mapsto \begin{cases} \mathtt{u} & \text{if } d(s) = \mathtt{n} \wedge \bigsqcup_i u(o_i) = \mathtt{u} \\ \mathtt{v} & \text{otherwise} \end{cases}$$

We omit the proof of monotonicity due to space limitations.

3.3 Optimizations

In the following paragraphs we describe techniques to take advantage of the results of the presented uniformity and divergence analyses.

Retaining Scalar Computations. The uniformity analysis allows us to prevent vectorization of uniform values. The benefit of using uniform computations is straightforward: register pressure is taken from the vector units and scalar computations can be executed in parallel to the vector computations. Furthermore, if a uniform value is used by a varying instruction, the value is *broadcast* to a vector beforehand. As a byproduct, our analysis computes the program points where scalar values have to be broadcast to vector values. This is important because we observed that eager broadcasting often causes performance degradation.

Retaining Control Flow. As discussed in the introduction, the divergence analysis opens up the possibility to retain uniform control flow. In the following, we describe an algorithm for CFG linearization which allows to exclude arbitrary, non-divergent control-flow structures.

First, we build regions of divergent blocks using a depth-first search on the CFG. While traversing, we create a new divergent region whenever we see a varying branch and mark this region as *active*. If we encounter a block that post-dominates all blocks in this region, we finish the region and set the active region to the last unfinished one. Divergent blocks are always added to the active region. Regions that overlap or have common entry or exit blocks are merged.

Next, each region is linearized recursively (inner regions before outer regions) as follows: We first determine an order for the contained blocks by sorting them topologically by data dependencies (linearized, inner regions are treated like one block). Next, we schedule the blocks of the region by rewiring all edges that target a divergent block. A block is scheduled after all its edges have been visited. The new target of each edge is the first divergent block of the current region's order that has not yet been scheduled. Edges that target non-divergent blocks remain untouched.

The reason behind creating schedules of the blocks and rewiring edges is that no block of a divergent region must be skipped during execution because this may violate the semantics of the original kernel.

Figure 9 illustrates an example where the non-divergent block e has a neighbor d that is divergent and thus always has to be executed. If we linearize all divergent blocks and retain the incoming and outgoing edges of e, we end up with a graph where blocks b and d can be skipped (Figure 9(d)), although some instances might want to take the path $a \to b \to d \to f \to g$. Dependencies are maintained correctly by rewiring the edge $e \to f$ to $e \to b$ (Figure 9(e)).

Figure 10 shows linearizations of the examples of Figure 6. In the leftmost CFG, only d, e, g, and i have to be linearized due to the varying branch in b. Because only one path from a to h leads through a varying branch, h is non-divergent. In the middle CFG, the inner loop, although being nested in a loop with varying exit, does not require any mask updates or blending because all active instances always leave the loop together. The rightmost CFG shows a case where it is allowed to retain the uniform loop exit branch in c: there are only uniform branches inside the loop, so either all or no active instance will leave the loop at this exit. However, h must not be skipped because of instances that might have left the loop earlier.

Linearization of patterns such as in the second graph of Figure 7 requires additional logic. This is because there is no schedule where all edges leaving i, j, and k can be rewired to a single block that has not yet been scheduled without violating dependencies. One possibility to handle this is duplication of code, another one is inserting conditional branches that depend on the previously executed divergent block. Due to space constraints we omit linearization examples for Figure 7 and leave a more detailed discussion of this issue for future work.

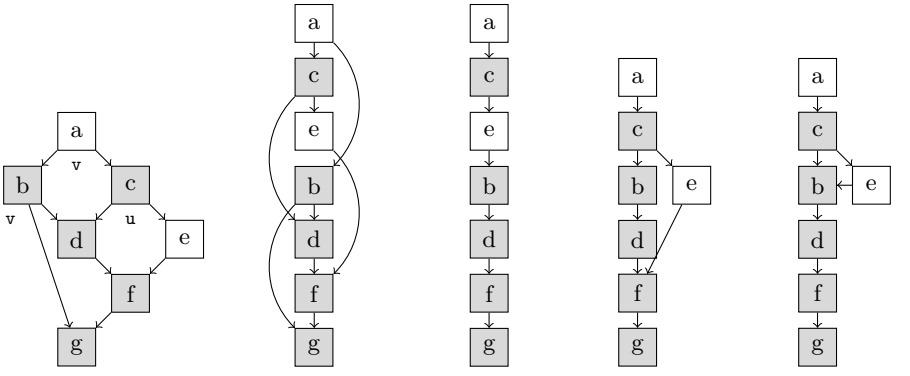

Fig. 9. CFG linearization example. In the original CFG (a), e is non-divergent because it can not be reached through different edges of varying branches. The topological sorting that is used in this linearization is shown in (b). The linearization (d) is invalid because it must not be possible to skip b and d. The graph (e) shows the correct linearization, which is likely to have better runtime than the naïve approach (c).

Reducing Mask Operations. During control-flow to data-flow conversion, each block is assigned a mask that is updated at each varying branch (conjunction and negation) and at each join block (disjunction).

If a conditional branch is uniform however, we use the incoming mask of the block for both outgoing edges instead of updating the mask with the comparison result. This implies that on paths with only non-divergent blocks, all edges have the same mask as the first block.

If our analysis found out that a block is always executed by *all* instances, the mask is set to `true`. At the end of regions with a single entry and exit block, the mask is reset to the one of the entry block. If a non-divergent block has multiple incoming edges, we generate a ϕ-operation instead of mask disjunctions. This is because only one of the incoming paths may have been executed.

In the rightmost CFG of Figure 10, blocks c and d can both use the entry mask of block b instead of performing conjunction-operations with the (negated) branch condition in b, and block f can use the same mask instead of the disjunction of both incoming masks.

Loops require special *loop exit masks* in order to store the information which instances have left the loop through which exits [12]. However, if an exit block is non-divergent, we omit its loop exit mask because it is equal to the active mask. If all exit blocks are non-divergent no loop mask is required because all instances that enter the loop will exit together through one of the exits.

Reducing Blend Operations. If control flow is linearized, ϕ-operations in blocks with multiple predecessors have to be transformed to select-operations that conditionally blend together incoming values based on the active mask.

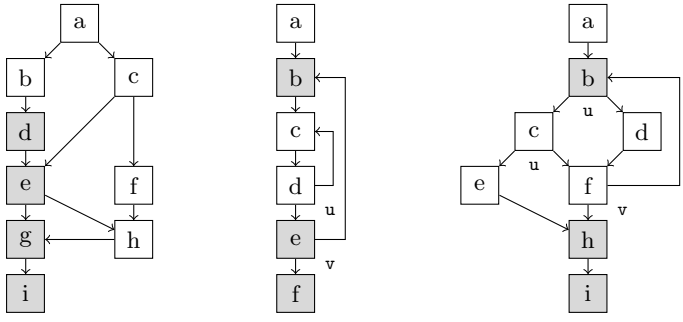

Fig. 10. Valid linearizations of the CFGs shown in Figure 6

However, if a block with multiple incoming edges is non-divergent, we can retain its ϕ-operations. Blending in such a case is not necessary because only one of the incoming paths may have been executed. For example, in the leftmost CFG of Figure 10, the ϕ's in block h remain untouched.

If an edge is rewired, the ϕ's from the old target block have to be modified (one direction will yield only dummy values that are later masked out) and moved to the new target block of the edge. For example, the ϕ's in block f of Figure 9 have to be moved to block b.

Loops require a blend operation for all values *live across loop boundaries* before the back branch of the loop [12]. If our analysis proves the loop to only exit to non-divergent blocks, we also do not require any blending because all instances will iterate equally often.

Optimizing Operations with Side-Effects. Operations with side-effects such as store operations and function calls have to be split up and each scalar operation must not be executed unless the corresponding mask element is `true`. If our analysis proves that a block is always executed by *all* instances, we prevent generation of such expensive code because all mask elements will be `true` when reaching the block.

Optimizing Loop Induction Variables. A more subtle optimization aims at values that are independent of any input of the function: values related to loop induction variables. Consider the Mandelbrot kernel in Figure 11. The exit condition of the main loop is varying because one of the two comparisons depends on varying values ($x2$, $y2$). Therefore, all values that are live across loop boundaries also have to be considered varying because they may differ between instances due to different loop trip counts. This means that `iter`—because it has a use after the loop—and all its uses (the increment and the comparison) have to be vectorized.

```
uint iter;
for(iter=0; (x2+y2 <= scaleSquare) && (iter < maxIter); ++iter) {
    y = 2 * x * y + y0;
    x = x2 - y2   + x0;
    x2 = x*x;
    y2 = y*y;
}
int tid = get_global_id(0);
image[tid] = 255*iter/maxIter;
```

Fig. 11. Main loop of the OpenCL Mandelbrot kernel

However, we can perform the following optimization: We introduce an additional vector variable that holds the "result" of `iter` which is updated after every iteration of the loop and is also used to determine when to exit the loop. The update is performed by a broadcast of `iter` (which remains scalar) followed by blend operation. This allows us to perform all computations inside the same loop iteration that only depend on input-independent or uniform values in scalar registers.

In the Mandelbrot example, this means that the increment of `iter` and the comparison `iter < maxIter` can remain scalar and `maxIter` does not require a broadcast. We observed a speedup of over 400% when applying this optimization (see Section 5). The reason for this is that the optimized operations are inside a frequently executed loop. One required vector register less or more in this critical part can affect performance significantly.

4 Related Work

Our SIMD vectorization technique stems from work of Allen [1] on the conversion of control flow to data flow. In classic loop vectorization [2,5,23,6,17], the innermost loop level is unrolled before combining isomorphic statements to vector operations ("unroll-and-jam"), either for vector machines or SIMD instruction sets. Target loops are usually restricted to either static iteration counts, specific data-dependency schemes, or straight-line code. Ngo [16] was the first to describe "Outer-Loop Vectorization" (OLV) [18]. In OLV, outer loops are unrolled to improve vectorization, e.g. due to longer trip counts of outer loops or better memory access schemes. Whole-Function Vectorization (WFV) [12] applied a scheme similar to OLV to data-parallel languages like OpenCL. By vectorizing an entire function, the new kernel computes W instances of the scalar code in parallel, effectively vectorizing the driver-level loop over the input data. Less generic approaches have been applied to various domain-specific languages [7,20,9,21,15].

The uniform value analysis was introduced as part of WFV. We are not aware of other related work that attempts to classify data-parallel code into uniform and varying instructions automatically. However, some data-parallel languages like ISPC [21] or the RenderMan Shading Language [4] have explicit keywords

to enforce specific behavior where required. In languages like OpenCL [13] and CUDA [19] this is not necessary because the semantics of the kernel are given by the execution model.

The only prior work directly related to our divergence analysis that we are aware of is a technique that employs "branches-on-superword-condition-code" (BOSCC) introduced by Shin [22]. This technique reintroduces control flow into vectorized code to exploit situations where the predicates (masks) for certain parts of the code are entirely `true` or `false` at runtime. While this prevents unnecessary execution of code, it suffers from some overhead for the dynamic test and does not solve the problem of increased register pressure: the code still has to account for situations where the predicate is not entirely `true` or `false`. Our analysis can do better by providing the necessary information statically, which allows to retain the original control flow that does not require any blending. However, it is possible to benefit from both our optimizations *and* BOSCCs. The next section will evaluate both approaches.

An increasing number of OpenCL drivers is being developed by different software vendors for all kinds of platforms from GPUs to mobile devices. For comparison purposes, the x86 CPU drivers by Intel [10] and AMD [3] are most interesting. However, there is not much detail on the underlying implementations. Both drivers have in common that they build on LLVM and exploit all available cores with some multi-threading scheme. The Intel driver also performs SIMD vectorization similar to our implementation[3]. However, to our knowledge, it lacks analyses to retain uniform computations and control flow, an important source of performance (see Section 5).

Recently, The Portland Group released an x86 CPU driver for CUDA [26] that also makes use of both multi-threading and whole-function vectorization, but no implementation details are publicly available. MCUDA [25] is another x86 CPU implementation of CUDA that introduced the first "thread loop"-based synchronization scheme. Their approach uses loop fission of the thread loop to remove barriers, which results in similar code as our approach if no barriers are inside loops. In that scenario however, MCUDA generates additional, artificial synchronization points at the loop header and before the back branch. This can impose significant overhead due to additional loading and storing of live variables. Jääskeläinen et al. [11] implemented a standalone OpenCL compiler that generates customized code for FPGAs and also uses this synchronization scheme. In contrast to our driver, they rely on the instruction-level parallelism of the FPGA design by duplicating kernel code W times instead of performing explicit SIMD vectorization. TwinPeaks [8] is an implementation of the OpenCL API that targets both CPUs and GPUs but does not perform aggressive code transformations. Their synchronization scheme uses custom implementations of `setjmp()/longjmp()` whereas our driver modifies the kernel code directly, storing only the live values instead of blindly saving registers. Clover [24] is an open

[3] We have no information about the AMD driver but suspect that no whole-function vectorization is used due to the inferior performance.

Table 1. Median kernel execution times of our OpenCL driver in different configurations for different applications (no multi-threading, 50 iterations). The row "speedup" shows the effect of our divergence optimizations, comparing "UniCF" to "UniVal" (95% confidence level).

OpenCL Kernel Performance (milliseconds)							
Application	Input Size	Scalar	Naïve	UniVal	BOSCC	UniCF	Speedup
BitonicSort	1,048,576	1,649	549	519	518	519	1.00×
BlackScholes	16,777,216	2,743	713	672	672	672	1.00×
DCT	$4{,}000^2$	732	1,100	857	857	411	2.09×
FastWalshTransform	134,217,728	9,852	13,317	13,450	13,451	13,458	1.00×
FloydWarshall	512	444	4,081	3,592	3,420	3,603	1.00×
Histogram	$15{,}000^2$	1,575	1,703	1,454	1,469	1,266	1.15×
Mandelbrot	$8{,}192^2$	4,136	8,114	1,724	1,727	1,725	1.00×
MatrixTranspose	$12{,}000^2$	2,295	2,378	1,599	1,599	1,600	1.00×
NBody	19,968	3,768	2,099	1,410	1,408	1,035	1.36×

source OpenCL driver on top of Gallium3D which implements synchronization with POSIX contexts. Both TwinPeaks and Clover do not employ WFV.

5 Experimental Evaluation

Our OpenCL driver is based on the LLVM compiler framework [14] and the AMD APP SDK [3]. We did not attempt to implement the full OpenCL 1.1 API rather than a sufficiently complete fraction to run benchmarks from the APP SDK. If necessary, the benchmarks were modified to only use scalar values instead of the OpenCL built-in vectors to allow for automatic vectorization. All experiments were conducted on a Core 2 Quad at 2.8 GHz with 4 GB of RAM running Windows 7. The vector instruction set is Intel's SSE 4.1, yielding a SIMD width of four 32 bit values. The machine ran in 64 bit mode, thus 16 vector registers were available.

We report kernel execution times of our driver in different configurations and compare to Intel's [10] and AMD's [3] CPU driver. Each measurement shows the median of 50 individual runs per configuration per benchmark without warm-up. Although the machine was not rebooted after every run, the numbers reported here are as realistic as possible for one cold-started, arbitrary run of the application. In addition, we conducted tests that ensure significance of our results using the "SpeedUp Test" [27].

5.1 Benchmarks

Table 1 shows the runtime performance of a diverse set of applications in different configurations: The first configuration ("scalar") performs no vectorization, so the kernel is executed sequentially. The "naïve" configuration performs vectorization without retaining uniform values and with complete linearization. The

Table 2. Median kernel execution times of our OpenCL driver (VecOCL, vectorized and multi-threaded) compared to the proprietary drivers of Intel and AMD. Values marked with an asterisk are execution times without WFV: for FloydWarshall, the Intel driver does not perform vectorization. We obtain an average speedup factor of the median of over 2.5 against the Intel driver at a confidence level of 95%.

OpenCL Kernel Performance (milliseconds)

Application	VecOCL	Intel	AMD	Speedup vs Intel
BitonicSort	164	1,170	47,271	7.13×
BlackScholes	241	329	717	1.37×
DCT	201	350	693	1.74×
FastWalshTransform	4,944	6,661	8,601	1.35×
FloydWarshall	934(148*)	525*	471	0.56×(3.55×*)
Histogram	387	1,178	527	3.07×
Mandelbrot	632	1,930	29,045	3.05×
MatrixTranspose	1,072	2,933	10,748	2.74×
NBody	343	676	1,253	1.97×

"UniVal" configuration linearizes all control flow, but retains uniform values where possible. The "UniCF" configuration additionally employs our analysis, leaving non-divergent control flow intact. "BOSCC" refers to the "UniVal" configuration with additional insertion of BOSCCs.

The overall observation is that performance increases with the addition of analyses and optimizations (from left to right in Table 1). Retaining uniform values proves to be effective for all of the benchmarks with an average speedup factor of 1.21. Retaining non-divergent control flow helps most in presence of loops with non-divergent exits as in DCT, Histogram, and NBody. These benchmarks profit from the reduced overhead of mask and blend operations and retained control flow, which results in speedup factors of 2.09, 1.15, and 1.36. As expected, there is no effect on benchmarks that do not have any non-divergent control flow, such as BitonicSort, BlackScholes, or MatrixTranspose.

Table 1 also shows numbers of a configuration that does not use our divergence analysis but inserts BOSCCs after linearization. It can be observed that this technique does not impact performance largely (only FloydWarshall runs 5% faster) in contrast to the configuration which makes use of our divergence analysis. This is mostly due to the fact that BOSCCs do not help in presence of loops, which are the hot spots in most of the benchmarks: a loop always has to be executed as long as any instance is still iterating. Introducing BOSCCs does not help here, whereas our optimizations can remove blend and mask operations if all loop exits are non-divergent. When combining all techniques, we match the performance of the "BOSCC"-configuration for FloydWarshall, all other benchmark results remain unchanged from "UniCF".

The Mandelbrot benchmark additionally profits from the special optimization described in Section 3, which resulted in a reduction of the kernel execution time from 8.1 seconds to 1.8.

It is also important to note that naïve vectorization is often inferior to scalar execution (DCT, Histogram, and MatrixTranspose), which highlights the importance of additional optimizations. Despite our efforts, there are still benchmarks that are not suited for vectorization such as FastWalshTransform and Floyd-Warshall, which are dominated by random memory accesses.

For a fair comparison against Intel's and AMD's drivers we implemented a naïve, unoptimized multi-threading scheme that uses OpenMP. Table 2 shows that our custom driver significantly outperforms both drivers in all test-cases (statistically significant with a confidence level of 95%).

6 Conclusion

Whole-function vectorization of kernels is the technique of choice to achieve maximum performance of data-parallel languages on CPUs. However, naïvely vectorizing all code can greatly limit the benefits due to the possibly large overhead of control-flow to data-flow conversion. We presented key techniques to reduce this overhead based on the analysis of divergent control flow.

In addition, we described code generation techniques to reduce the overhead that is inherent to data-parallel languages like OpenCL and CUDA: we integrated parts of the driver code into the kernel and used a novel synchronization-scheme based on continuations to enable aggressive optimizations.

Our techniques have proven to be successful on a variety of different benchmarks, significantly outperforming proprietary drivers by Intel and AMD.

We are aware of the fact that we did not provide a formal proof of our transformations. However, proving correctness requires a formal semantics of a data-parallel language such as OpenCL which has not been developed yet. Such a semantics would also enable a more formal definition of divergence. We leave this for future work.

Acknowledgement. This work is part of the ECOUSS project and has been funded by the German Ministry for Education and Science (BMBF) and the Intel Visual Computing Institute Saarbrücken. The authors would like to thank Christoph Mallon and Daniel Grund for insightful discussions about control-flow divergence. Furthermore, we thank Roland Leißa and the anonymous reviewers for their helpful comments and remarks.

References

1. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: POPL, pp. 177–189. ACM (1983)
2. Allen, R., Kennedy, K.: Automatic translation of FORTRAN programs to vector form. ACM Trans. Program. Lang. Syst. 9(4), 491–542 (1987)
3. AMD: AMD APP SDK v2.5 (March 2011)
4. Apodaca, A., Mantle, M.: RenderMan: Pursuing the Future of Graphics. IEEE Computer Graphics & Applications 10(4), 44–49 (1990)

5. Cheong, G., Lam, M.: An Optimizer for Multimedia Instruction Sets. In: Second SUIF Compiler Workshop (1997)
6. Darte, A., Robert, Y., Vivien, F.: Scheduling and Automatic Parallelization. Birkhauser, Boston (2000)
7. Fritz, N., Lucas, P., Slusallek, P.: CGiS, a New Language for Data-Parallel GPU Programming. In: VMV, pp. 241–248 (2004)
8. Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B.R., Zheng, B.: Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In: PACT, pp. 205–216. ACM, New York (2010)
9. Hormati, A.H., Choi, Y., Woh, M., Kudlur, M., Rabbah, R., Mudge, T., Mahlke, S.: Macross: macro-simdization of streaming applications. In: ASPLOS, pp. 285–296. ACM, New York (2010)
10. Intel: Intel OpenCL SDK 1.1 (June 2011)
11. Jaskelainen, P.O., de La Lama, C.S., Huerta, P., Takala, J.: OpenCL-based design methodology for application-specific processors. In: SAMOS 2010, pp. 223–230 (July 2010)
12. Karrenberg, R., Hack, S.: Whole Function Vectorization. In: CGO, pp. 141–150 (2011)
13. Khronos Group: OpenCL 1.1 Specification (June 2011)
14. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO (March 2004)
15. Newburn, C.J., So, B., Liu, Z., McCool, M.D., Ghuloum, A.M., Toit, S.D., Wang, Z.G., Du, Z., Chen, Y., Wu, G., Guo, P., Liu, Z., Zhang, D.: Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In: CGO, pp. 224–235 (2011)
16. Ngo, V.: Parallel loop transformation techniques for vector-based multiprocessor systems. Ph.D. thesis, University of Minnesota-Twin Cities (May 1994)
17. Nuzman, D., Henderson, R.: Multi-platform auto-vectorization. In: CGO, pp. 281–294 (2006)
18. Nuzman, D., Zaks, A.: Outer-loop vectorization: revisited for short simd architectures. In: PACT, pp. 2–11. ACM (2008)
19. NVIDIA: CUDA Programming Guide (2009)
20. Parker, S., et al.: RTSL: A Ray Tracing Shading Language. In: IEEE Symposium on Interactive Ray Tracing (2007)
21. Pharr, M.: Intel SPMD Program Compiler (June 2011)
22. Shin, J.: Introducing Control Flow into Vectorized Code. In: PACT, pp. 280–291. IEEE Computer Society (2007)
23. Sreraman, N., Govindarajan, R.: A vectorizing compiler for multimedia extensions. Int. J. Parallel Program. 28(4), 363–400 (2000)
24. Steckelmacher, D.: An OpenCL State Tracker for Gallium based on Clover (August 2011), `http://people.freedesktop.org/~steckdenis/clover`
25. Stratton, J.A., Stone, S.S., Hwu, W.-m.W.: MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 16–30. Springer, Heidelberg (2008)
26. The Portland Group, Inc.: PGI CUDA-x86 (June 2011)
27. Touati, S.A.A., Worms, J., Briais, S.: The Speedup Test. Rapport de recherche (2010), `http://hal.inria.fr/inria-00443839/en/`