FPGA-aware Transformations of LLVM-IR

Franz Richter-Gottfried, Sebastian Hain, and Dietmar Fey
Chair of Computer Science 3 (Computer Architecture)
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
91058 Erlangen, Germany
email: {franz.richter-gottfried, sebastian.hain, dietmar.fey}@fau.de

***Abstract*—The paper presents hardware-aware optimizations of the assembly language used by LLVM to optimize resource usage when an algorithm written in the Open Computing Language (OpenCL) is translated into a design for a field programmable gate array (FPGA) by the tool OCLAcc. In signal processing, latency and throughput of a solution are important, but also its efficiency. FPGAs offers high performance and low energy consumption for many applications, at the cost of a complex development. With high-level synthesis (HLS) the design process can be simplified significantly. We introduce our transformation of the control flow and how we minimize the bitwidth of data and operations performed. In contrast to existing work, we focus on the applicability for FPGAs and HLS from OpenCL. Both optimizations allow the generation of simpler hardware. We present metrics to rate the results with estimations of FPGA resources needed and demonstrate them using the Sobel operator, which is part of many image processing applications. Our results show that we can completely eliminate branches and reduce the total amount of bits by 16 % for a typical input configuration.**

Keywords–OpenCL; LLVM; high-level synthesis; FPGA; if-conversion; bitwidth reduction

I. INTRODUCTION

Modern FPGAs are popular for fast signal processing, because they offer a high degree of parallelism and low power consumption. This is proven, e.g., by integrated DSP blocks or hardwired floating point units on newer devices. However, it is more complex to create a custom hardware design than to optimize an algorithm for a fixed CPU. High-level synthesis (HLS) promises to fill this gap by deriving a hardware design from an algorithmic description. Inherently parallel source languages like OpenCL have the advantage that the mapping to FPGA resources is easier, compared to sequential languages like C, leading to more efficient designs. We first give a short introduction to OpenCL and the HLS-tool OCLAcc.

OpenCL: is a freely available standard created and supported by Apple, Intel and other companies. Its purpose is to describe a parallel problem and solve it on a variety of devices, managed by a host, which is usually a normal CPU. Common devices include CPUs and GPUs, but due to abstraction, host and device may even be the same physical CPU.

OpenCL defines a library interface for the host to control devices. The algorithm itself, referred to as kernel in the following, is written in the C-style language OpenCL-C. Devices offer compute units (CU), and each of them consists of processing elements (PE) to execute the kernel in parallel. Work is distributed among the PEs according to the OpenCL execution model. A work item, which is an entity in the problem space (NDRange) represents a single instance of the kernel (see Figure 1). Multiple work items share the same work group, which allows synchronization and data exchange using fast local memory. In contrast, communication among work groups does not allow synchronization and relies on slower global memory, as there is no guarantee, when and on which CU work groups are scheduled.

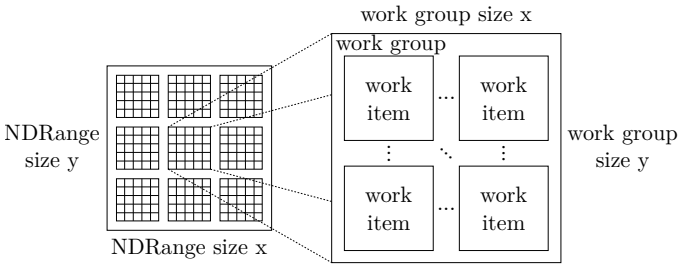


Figure 1. OpenCL Execution Model

LLVM-IR/SPIR: is a machine independent pseudo assembly language generated from the OpenCL kernel by Clang, a C-frontend for LLVM [1]. Standard Portable Intermediate Representation (SPIR) is a standardized version of LLVM-IR. Though our transformations presented target SPIR, they can also be used to optimize LLVM-IR. LLVM itself includes several passes to analyze and modify IR, e.g., alias analysis or vectorization, but most of them cannot be directly used to optimize IR for hardware generation.

OCLAcc: derives an FPGA design from OpenCL-C [2]. Figure 2 shows the steps of the transformation. Before running OCLAcc, the OpenCL kernel is translated to SPIR by a modified version of Clang maintained by the Khronos-Group. Translation in OCLAcc happens in two steps. First, SPIR is used to generate OCLAccHW, an internal representation of the data flow, optimized to derive hardware from. It works on basic blocks, which are instruction sequences always executed sequentially from the first to the last instruction. Inputs and outputs are analyzed to identify ports of the later design and streams from and to memory with their static and dynamic indices. Furthermore, the OpenCL standard includes built-in functions callable by a kernel, including functions for organization, synchronization and data access, which are mapped to specific components and control inputs. OCLAccHW also is used for hardware-specific optimization. HWMap, the second step in OCLAcc, depends on the actual hardware used, i.e., vendor and type of FPGA boards. OCLAcc either directly instantiates components, generates IP-cores, or relies on inference by the vendor tools. Scheduling of components is tightly coupled with their generation, because for many parts of the system, parameters like latency or maximum clock frequency are only available when they have been implemented and cannot be used for optimization before. Instead, metrics are

 ISBN: 978-1-61208-487-9

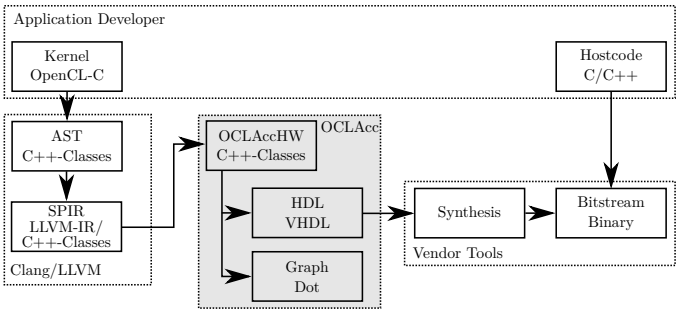


Figure 2. Components of OCLAcc

used (see Section IV). Clock synchronization of components is only done inside of basic blocks, while between blocks, each input and output carries an additional valid-bit. Blocks have to wait for their inputs to become valid before they can start computations. This minimizes synchronization overhead for the block scheduler.

Hardware generation issues: LLVM-IR is designed to be translated into code that will be executed on common CPUs or similar platforms, but an FPGA does not provide any of the capabilities of those platforms: branches cannot be directly mapped to an FPGA as there is no program counter or memory to load instructions from. Instead, the data flow defined by an instruction sequence is translated into functional units. As branches result in several basic blocks they lead to an increased synchronization overhead in form of additional registers and valid bits between those units.

OpenCL does not offer data types with variable bitwidth but only allows conventional types like `char` or `int`. In LLVM-IR and SPIR, types are mapped to integers of arbitrary bitwidth (`i1` for bool, `i32`) or floating point numbers (`float`), but Clang only uses those representing common types. This is a drawback since it is not possible to exploit the flexibility of an FPGA.

The remaining paper is structured as follows. In Section II, we present similar optimizations already published. We then introduce the example application in Section III. Sections IV and V present the transformations used to simplify the control flow and minimize the bitwidth of data paths and discuss their impact, respectively. Section VI summarizes the paper.

II. Related Work

This section gives several examples of control flow optimizations, however, most of them are designed to minimize instructions executed by a CPU. In [3] superblocks are introduced: a superblock consists of several basic blocks to enable the compiler to create code with a higher instruction level parallelism. Therefore, a sequence of basic blocks expected to be often executed forms a superblock. Then, during tail duplication, the superblock is cloned and each branch leaving and reentering the original superblock is redirected to target the cloned superblock instead. The new superblock now only has a single entry at its root to simplify scheduling inside the block and to exploit ILP. The problem for hardware generation is the process of tail duplication: copies of basic blocks increase the hardware consumption on the FPGA and have to be avoided.

The authors of [4] combine superblocks with if-conversion whereby each instruction is predicated and only executed if that condition is true. They call these blocks hyperblocks. Hyperblocks can be larger than superblocks, allowing more efficient instruction scheduling and a reduction of branches to avoid performance penalties of branching overhead and misprediction. Besides the problems of tail duplication, this approach assumes that the target architecture is able to handle predicats and can skip instructions with violated predicates. As predicates are only available at runtime, a hardware design has to implement all instructions, leading to a waste of resources.

Allen *et al.* [5] present a transformation that converts much more control flow dependencies to data flow dependencies and creates a kind of predicated execution. Branches are categorized either as forward branch, exit branch or backward branch. The first are eliminated, and during this process condition variables are introduced for each statement. If such a variable is true, the associated instruction will be executed. This simplifies the control flow graph (CFG) and control dependencies are converted into data dependencies. In [6], another algorithm for if-conversion is presented that tries to assign predicates as early as possible. Furthermore, some optimizations are presented to keep those predicates simple. Both approaches assume that predicated instructions have an advantage at runtime, which cannot be applied to hardware generation.

The authors of [7] briefly mention if-conversion for FPGAs by using multiplexers and predicates, which is in general the preferred way in hardware design. However, they do not explain or discuss when it is possible or feasible, nor do they describe the transformation in detail.

LLVM itself already provides optimization passes for if-conversion, but those work on a machine-instruction level instead of IR. This means, they operate on a hardware-specific level and depend on details of the target architecture. As OCLAcc transforms LLVM-IR code into a hardware description and does not use any machine-instructions, those optimizers cannot be used. The only integrated if-conversion optimizer working on IR-level is too conservative and does not detect all the cases of our solution.

There are also several publications explaining approaches to minimize the bitwidth of integer data paths, including software-based approaches like FRIDGE [8], which simulates the execution to get run-time values. The user constrains the range of input values to allow an interpolation of the needed bitwidth for other operations and intermediate values.

Lee *et al.* [9] present MiniBit, a static bitwidth optimizer based on range and precision analysis. Like for FRIDGE, the range of the input values is supplied by the user. Range analysis is performed with Affine Arithmetic, while they use an error function to calculate the required fraction bitwidth. The authors demonstrate the results for different algorithms on a Xilinx Virtex-4 FPGA.

Our requirements to optimize bitwidths differ from the solutions available as we only can use the information provided by the author of an OpenCL kernel, and it must not break compatibility with the OpenCL standard. We use a static approach, because it cannot be assumed that the programmer of an OpenCL kernel has run-time information.

III. Reference code

To demonstrate our transformations, we use the Sobel operator, a simple convolution algorithm often used to preprocess

 ISBN: 978-1-61208-487-9

```
void kernel Sobel(global int *a, global int *b) {
    int idx = get_global_id(0), idy = get_global_id(1);
    int sx = get_global_size(0), sy = get_global_size(1);
    int v, c  = a[idy * sx + idx];
    if (idx!=0 && idx!=sx-1 && idy!=0 && idy!=sy-1) {
        int nw = a[(idy-1) * sx + idx - 1];
        int n  = a[(idy-1) * sx + idx];
        int ne = a[(idy-1) * sx + idx + 1];
        int w  = a[idy * sx + idx - 1];
        int e  = a[idy * sx + idx + 1];
        int sw = a[(idy+1) * sx + idx - 1];
        int s  = a[(idy+1) * sx + idx];
        int se = a[(idy+1) * sx + idx + 1];
        int vx = nw - ne + 2*w - 2*e + sw - se;
        int vy = nw + 2*n + ne - sw - 2*s - sw;
        v = (int) sqrt( (float)(vx*vx + vy*vy));
    } else v = c;
    b[idy * sx + idx] = v;
}
```

Figure 3. Sobel Operator

```
define cc75 void @Sobel(i32 addrspace(1)* noalias ↩
    nocapture readonly %a, i32 addrspace(1)* noalias ↩
    nocapture %b) #0 {
  %1 = tail call cc75 i32 @_Z13get_global_idj(i32 0)
  ...
  %8 = icmp eq i32 %1, 0
  br i1 %8, label %62, label %9
; <label>:9                       ; preds = %0
  ...
  %or.cond3 = or i1 %or.cond.not, %13
  br i1 %or.cond3, label %62, label %14
; <label>:14                      ; preds = %9
  ...
  br label %62
; <label>:62                      ; preds = %0, %9, %14
  %v.0 = phi i32 [ %61, %14 ], [ %7, %9 ], [ %7, %0 ]
  %63 = getelementptr i32 addrspace(1)* %b,i32 %5
  store i32 %v.0, i32 addrspace(1)* %63, align 4
  ret void
}
```

Figure 4. PHI Node Insertion

images in computer vision. It consists of two separate filter kernels, and the combination of both gives the magnitude of the gradient.

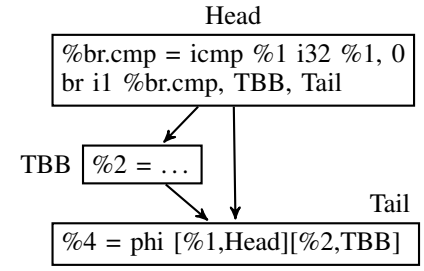
Each iteration loads eight values from memory to produce a single result. Updated values are stored in a separate image, so synchronization is only needed at the end. Since the kernel performs a single update, there is no need to explicitly synchronize at all. To update the whole image, no loop inside of the kernel is used, but instead each work item takes care of updating a single cell. Values at the border are preserved and copied into the new image. This diverging control flow is generated by an if-statement. Figure 3 shows the OpenCL kernel implementation of the Sobel operator.

Clang translates the if-statement into branches to different basic blocks, depending on which condition is met. For a CPU, this reduces the amount of instructions executed if one of the first conditions is false and the then-clause can be skipped. However, for reasonably large images, this is not the default case and the then-branch is executed far more often than the else-branch. After the if-statement, the diverged control flow is unified in IR by a PHI instruction in the last basic block. Its purpose is to select a single value from a list, depending on the block from which it was reached. This is common for single static assignment code (SSA), when the value of a variable depends on a condition, because reassigning is not allowed. See Figure 4 for the relevant parts of the generated LLVM-IR code.

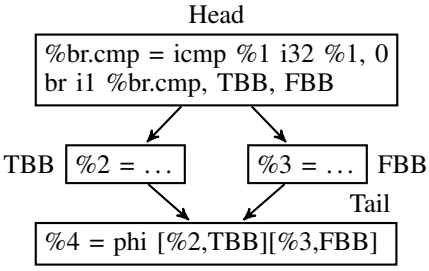
IV. IF-CONVERSION

This section presents our transformations of the control flow to eliminate branches and enlarge basic blocks. To simplify the control flow, all values are computed speculatively instead of jumping to different blocks. PHI-nodes are replaced by select-instructions, which choose one of two values, depending on a condition variable.

Our transformation recognizes the if-patterns shown in Figure 5, and switch-patterns which are not covered by this paper. In all examples a condition variable is computed in the head block and evaluated by a conditional branch at its end. That branch then jumps to the proper block. After the then– or else-clause, the tail block merges the control flow again. It



(a) Triangle for if-then



(b) Diamond for if-then-else

Figure 5. Supported patterns for if-conversion

usually contains one or more PHI-nodes to select the correct results of the branch blocks.

The goal of the transformation is to convert the shown patterns into a single basic block. Therefore, the head is merged with the branch blocks of the if-pattern (then– and else-clause) and the branch is substituted by an unconditional jump to the tail. The moved code now is always executed. It has to be ensured that the code still computes the same results, i.e., the moved instructions must not have any side-effects. Now, the PHI-nodes in the tail are adapted. Previously, they selected the results from either the then– or the else-block, depending on the if-condition. After the transformation, all possible values are defined in the head, and the PHI-nodes can be eliminated. Select-instructions are inserted to decide between the results of the then– and else-block, depending on the same condition of the former if-statement. As a final step, tail and head can be

 ISBN: 978-1-61208-487-9

```
define spir_func void @Sobel(i32 addrspace(1)* noalias ↩
    nocapture readonly %a, i32 addrspace(1)* noalias ↩
    nocapture %b) #0 {
  %1 = tail call spir_func i32 @_Z13get_global_idj(i32 0)
  ...
  %7 = load i32 addrspace(1)* %6, align 4
  %8 = icmp eq i32 %1, 0
  ...
  %or.cond3 = or i1 %or.cond.not, %12
  ...
  %60 = fptosi float %59 to i32
  %61 = select i1 %or.cond3, i32 %7, i32 %60
  %62 = select i1 %8, i32 %7, i32 %61
  %63 = getelementptr i32 addrspace(1)* %b, i32 %5
  store i32 %62, i32 addrspace(1)* %63, align 4
  ret void
}
```

Figure 6. Transformation of the Sobel kernel

merged into a single block and the control flow is successfully converted into a data flow using select-instructions.

Figure 6 shows the transformed Sobel kernel. All branches are eliminated and the PHI-node is replaced by two select-instructions using the conditions of the former if-statements. This also demonstrates that the transformation correctly handles nested if-patterns, because it is applied iteratively. The following section covers the implementation of the transformation in detail.

Implementation: First we find supported patterns in the CFG: they consist of the head that is always executed and that contains the compare-instructions to compute the condition of the if-statement. The condition variable is used by conditional jumps to one of the successors. Note that the head must have exactly two successors. For if-then-else-statements, the two successors are the basic blocks for then and else. We call such a pattern a diamond (Figure 5(b)). If the statement does not contain code to be executed when the condition is not met, i.e., it has no else branch, only the then-successor remains and the other one is directly the tail block. We call those patterns a triangle (Figure 5(a)). The next step in finding appropriate patterns is to investigate the branch blocks: a diamond has two branch blocks, then (TBB) and else (FBB). Both must have the head as their single predecessor and the tail as their single successor. A triangle only contains the TBB. Compiler optimizations may result in an inverted condition and thus switched TBB and FBB. This can especially be the case for negated if-conditions. Our implementation takes care of that but it is not discussed here. The final part of a pattern is the tail that merges the control flow from the branch blocks and contains the PHI-nodes that select the proper results depending on the run-time control flow. The tail may have other predecessors than the branch blocks. Successors of the tail block are irrelevant for the transformation.

If a pattern is detected, further checks have to be performed to make sure that the transformation does not change the semantics of the blocks by executing instructions with side-effects. Especially store– and synchronization-instructions are forbidden. For example, a store inside of the then-block of an if-statement may only be executed if the if-condition is true. With merging the block into head, the store-instruction is always executed, likely leading to wrong results.

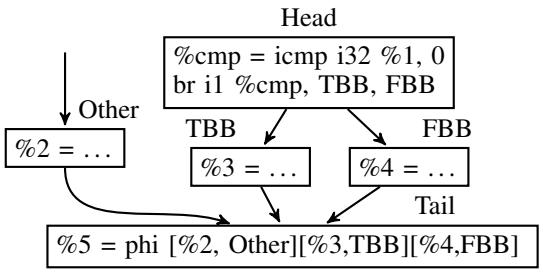
As the memory-bandwidth of FPGA boards even is a stronger performance bottleneck compared to GPUs, load-instructions can be seen to have light side-effects, as we call instructions not influencing correctness but performance. By not transforming blocks with loads, performance or resource usage may also suffer as explained above, as it leads to more and shorter basic blocks. Furthermore, appropriate caches can minimize the performance penalty of speculatively executed loads. For the Sobel kernel, we do not take light side effects into account, though it is possible by the implementation. Loads are thus moved into the head and the transformed kernel consists of a single large block instead of four smaller ones. If all tests are passed, the transformation is performed. First, the instructions of the branch blocks are moved into the head, right before the final branch. The branches inside of the blocks are omitted. At this point it becomes clear, why the branch blocks must not have any other successors than the tail. Otherwise, their final instruction would be a conditional branch to reach other blocks. That branch would also have to be transferred into the head, but as branch-instructions may have side-effects they are not allowed to be executed speculatively. Therefore, the branch instruction must not be moved into the head, and as after the transformation the single successor of the head block has to be the tail block, the branch block must also have the tail as its single successor to preserve semantics.

In the next step, we determine if the tail block has other predecessors than the branch blocks. For a diamond the predecessors must include TBB and FBB. Then, the input values to the PHI-node for those two blocks are extracted. Now, as these values are moved from TBB and FBB to head, they both are already available. In those cases the usage of the operands is said to be dominated by their definitions. This also is the reason why the branch blocks must have the head as their single predecessor: instructions in the branch blocks might need operands that were computed in preceding blocks. If a branch block now has other predecessors than the head, some operands may come from those other predecessors via a PHI-node. As the branch block is now merged into the head, it is not guaranteed that the definition of the operand also dominates the head block, which means that the needed operands might not be available in the head block, leading to an invalid data flow. This issue can be solved by tail duplication as mentioned in [3] by cloning the branch block and several predecessors. The head block is modified to branch to the copied branch block and like this the branch block has the head as its single predecessor. The main disadvantage is the increased hardware consumption for the copies. Therefore, we do not convert those patterns.

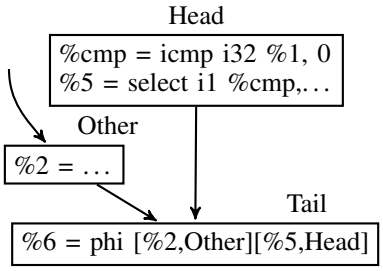
Now, a select-instruction is created: input operands are the previously extracted values from the PHI-node and the condition variable is the former if-condition. The new instruction is inserted into the head block before the final branch and uses of the PHI-node are replaced by that select-instruction. The PHI-node can then be removed from the tail block. If a triangle is processed then the input values for the new select-instruction come from the head block (if the if-condition is false) and the block for the then-part. The other steps are equal to those of processing a diamond. After the transformation, PHI-nodes in the tail are eliminated.

If the tail block has other predecessors than the branch blocks, the PHI-node cannot be eliminated. As the tail block can be reached via other control paths that do not contain

 ISBN: 978-1-61208-487-9



(a) Original



(b) Transformed

Figure 7. Control Flow Transformation of Diamond Pattern

the if-statement, the PHI-nodes have input values for other blocks as well. As a select instruction would just be able to choose between the values from the two paths of the currently processed if-statement, the PHI-node is still needed to choose between that preselected value and those values for the other predecessors not under control of the processed if-statement. This is the case for a nested if-statement. Nevertheless, in this case, the number of input values to the PHI-node can be reduced.

In a last step, the final branch of the head is replaced by an unconditional branch to the tail. It is tested if the tail can be merged into the head block: if the tail has the head as its single predecessor, head and tail can be merged, which eliminates the branch.

Figure 7(a) shows a diamond, where the PHI-nodes cannot be completely replaced. During the transformation the code from TBB and FBB is merged into the head block. Then, for each PHI-node in the tail block, a select instruction is created with the former if-condition (`%cmp`). The input values for the select instruction are extracted from the PHI-node. But as the tail block has other predecessors than TBB and FBB, the PHI-node cannot be replaced (Figure 7(b)).

The transformation is repeated until no more patterns are found. By that, nested if-statements can also be converted. Figure 8 shows the CFG corresponding to Figure 3 containing two triangles: the first triangle consists of BB9 as head, BB62 as tail and BB14 as branch. BB14 is merged into BB9 in the way described above: a select-instruction is inserted into BB62 to either select `%7` or `%60` depending on the value of `%or.cond3`. As BB62 has other predecessors than BB9 and BB14, the PHI-node is not removed but the input operands for BB9 and BB14 are replaced by the result of the created select instruction (the result is stored in `%61`). Now the next triangle gets visible consisting of the entry block, BB62 as tail and the merged block BB9/BB14. This new pattern can

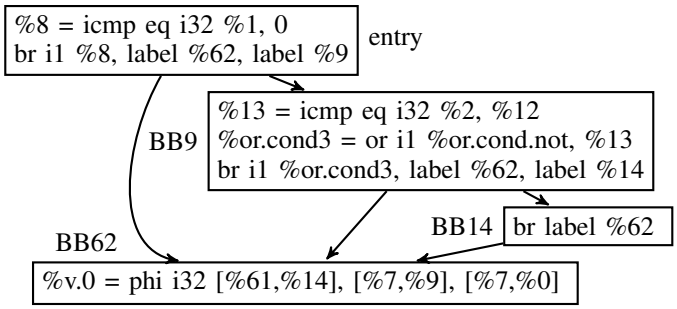


Figure 8. CFG of Sobel

now be transformed, too: again a select instruction is inserted using the new input values of the PHI-node: `%7` and `%61`. As condition variable `%8` (defined in the entry block) is used. But as now the PHI-node does not have any other predecessors than BB9/BB14 and entry, it can be removed and all instructions using its result will use the result of this second select. Finally, Figure 6 shows the fully transformed code using the created select-instructions.

Metrics: The implementation is able to consider certain additional soft constraints. Those are used to determine whether the transformation is useful to reduce hardware resources. Currently, the block size can be used to prevent the transformation to take place. This is useful if the critical path of the new block is very long. As one block can execute a single work item, a long critical path may degrade performance. More elaborate metrics take FPGA resources like LUTs, registers and BlockRAM into account.

A first indicator for those numbers is the amount of instructions in a block. All instructions are weighted equally, ignoring their operation or number and size of their operands. This of course is only a very rough estimation, but it can be computed very fast and it does not need any knowledge of the target device. For the Sobel kernel, we can replace four basic blocks with 72 instructions in total and different length to a single block with only 70 instructions. On the other hand, if the current kernel computes a border value, the same work has to be done instead of only 13 instructions.

To take the differing complexity of operations into account, instructions can be weighted by an instruction-specific factor. A bitwise shift operation is far cheaper than a division, represented by a smaller coefficient. Furthermore, number and size of operands give an additional weight.

More accurate metrics need knowledge of the targeted FPGA. Depending on the available resources, instructions can be mapped to different hardware, e.g., DSP blocks or IP cores. Prototypes of the instructions are synthesized by the backend of OCLAcc and the results of the vendor tools give the hardware consumption. This gives the most precise estimation of the resources needed. In fact, it even overestimates them as global hardware optimizations performed by the synthesis tool are not available when each component is separately synthesized. However, the estimation may take a very long time.

V. Minimize Bitwidth

As already mentioned, OpenCL does not support custom data types, which is one of the key advantages of an FPGA

 ISBN: 978-1-61208-487-9

design. This cannot be circumvented by adding keywords to the OpenCL frontend, because this would invalidate the kernel for other OpenCL platforms. The optimization presented works for integer values, which of course may also be a fixed point representation. This is often used in hardware design to save resources for integer but not for floating point values, because of the requirements of IEEE 754.

Similar to Altera's SDK for OpenCL, our transformation requires user input to statically reduce bitwidths in the form of bitmasks applied to variables. Typically, constrains are set for inputs and output, and all operations and values in between are derived from them. For the example in Figure 3, each load from image `a` may be replaced with `0xFFFFFF & a[...]` if only 3 Byte per pixel are to be processed. Constant values, which have no dedicated bitwidth in LLVM-IR, are stored with the minimum of bits. Though these optimizations may seem trivial, many cases have to be respected as explained in the following.

Implementation: We have to differentiate between values and constants. A value, starting with a % in LLVM-IR, has a fixed type (`i32` for 32 bit integer, `float`), but integer values are signless. However, a constant (`true`, `-1`, `1.5`) does not have a fixed bitwidth but it depends on the type of the operation using it, e.g., `%29 = add nsw i32 %5, -1` assigns the sum of value `%5` and the constant `-1` to `%29` without declaring the constant's bitwidth.

For values and constant operands, the bitwidth of the type and the minimum bitwidth are stored, as well as flags to indicate whether no extension (next), sign-extension (sext), zero-extension (zext) or one-extension (oext) has to be used. To gather the minimum bitwidth, the transformation is split into two phases: a forward propagation (FP) for each value, starting at the first instruction of the first basic block to get the minimum based on input width and the operation and a backward propagation (BP) in reverse to determine the actual bitwidth used, depending on the output.

In the forward step, we predict the required bitwidth based on its operands. If there is at least one zext operand, the output is as wide as the smallest of them. This is safe since all leading zeros of the operand eliminate possible ones of others, independent of their length. If all inputs are sext the required bitwidth is as wide as the output's width because sign-bits cannot be determined statically. For example, if all sign-bits of the operands are one, the result also is one-extended. On the other hand, we cannot just use the sext-flag as one input may be positive with a zero sign-bit, resulting in an output value being zero from that bit on.

For BP, we start with the bitwidth from FP. If any of the instruction's operands is a constant, its width delimits the instruction's, and all other operands' bitwidth. This is true even if FP resulted in a wider operation. If we have no requirements from FP, we use the width of the smallest operand to be propagated to it's predecessors.

In general, the bitwidth of constants can only be determined based on an operation using it. The constant -1 can be encoded using a single bit in two's complement, which works with signed operations like integer multiplication. For bitwise operations and operations being the same operation for signed and unsigned values (e.g., add), its width has to fit the other operands' to prevent wrong results.

By these steps, we first learn how wide a value may become because of the operations performed to compute it, and then reduce its size to a minimum based on the width of its users.

Results: The exact number of bits saved depends on the application, precisely the exact width of input and output values. The results demonstrate the effects of our optimization for the Sobel operator, but will be different for other applications. For the kernel in Figure 3, we save 414 bits of 2528 or around 16% for operations and 496 bits for constants if we limit input and output to 24 bit by using bitmasks, and the complexity of operations and thus the resource usage on an FPGA is significantly reduced.

VI. Conclusion

In this paper, we presented an algorithm to simplify the control flow of a given OpenCL kernel by transforming several blocks of if– or switch-statements into a single basic block. The new block's instructions coming from different branches are executed speculatively and the correct value is chosen depending on the if-condition by a select-instruction. We also presented a static method to reduce the bitwidth of instructions and constants, based on user-provided bitmasks. Both enable OCLAcc to produce more efficient and less complex hardware designs, inevitable for fast and efficient FPGA designs of signal processing applications.

References

[1] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," San Jose, CA, USA, Mar 2004, pp. 75–88.

[2] F. Richter-Gottfried and D. Fey, "OCLAcc: An open-source generator for configurable logic block based accelerators," in Embedded World Conference Proceedings, Feb 2014.

[3] W. M. W. Hwu et al., "The superblock: An effective technique for vliw and superscalar compilation," The Journal of Supercomputing, vol. 7, no. 1, pp. 229–248.

[4] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," SIGMICRO Newsl., vol. 23, no. 1-2, Dec. 1992, pp. 45–54.

[5] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ser. POPL '83. New York, NY, USA: ACM, 1983, pp. 177–189.

[6] J. Z. Fang, "Compiler algorithms on if-conversion, speculative predicates assignment and predicated code optimizations," in Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing, ser. LCPC '96. London, UK, UK: Springer-Verlag, 1997, pp. 135–153.

[7] S. Hauck and A. DeHon, Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, pp. 158–159.

[8] M. Willems, V. Bursgens, T. Grotker, and H. Meyr, "Fridge: an interactive code generation environment for hw/sw codesign," in Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on, vol. 1, Apr 1997, pp. 287–290 vol.1.

[9] D. Lee et al., "Accuracy-guaranteed bit-width optimization," IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 25, no. 10, 2006, pp. 1990–2000. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2006.873887

 ISBN: 978-1-61208-487-9