A Case for Better Integration of Host and Target Compilation When Using OpenCL for FPGAs

Taylor Lloyd, Artem Chikin, Erick Ochoa, Karim Ali, and José Nelson Amaral
University of Alberta, Edmonton, Canada

Abstract

Major Field-Programmable Gate Array (FPGA) vendors, such as Intel and Xilinx, provide toolchains for compiling Open Computing Language (OpenCL) to FPGAs. However, the separate host and device compilation approach advocated by OpenCL hides compiler optimization opportunities that can dramatically improve FPGA performance. This paper demonstrates the advantages of combined host and device compilation for OpenCL on FPGAs by presenting a series of transformations that require inter-compiler communication. Further, because of extremely long FPGA synthesis times, the overhead of recompiling the host code for each compilation of FPGA kernel code is relatively inexpensive. Our transformations are integrated with the Intel FPGA SDK for OpenCL and are evaluated on a subset of the Rodinia benchmark suite using an Altera Stratix V FPGA.

1 Introduction

Open Computing Language (OpenCL) [16] has emerged as a prominent programming model for Field-Programmable Gate Array (FPGA) . Intel and Xilinx release OpenCL compiler toolchains that support hardware synthesis directly from OpenCL source [7, 8]. A major feature of OpenCL is separate host and device compilation, allowing OpenCL device vendors to specialize in device-code generation without concern for host implementations. This separation enforces strict Application Programming Interface (API) boundaries between host and device implementations and prevents otherwise trivial compiler optimizations and analyses. As a result, workarounds must be introduced to recover lost performance. For example, Intel FPGA extends the OpenCL specification with *channels* that allow kernel operations to be chained without needing to copy back to memory. If a compiler had access to the combined host and device code, chaining would be a trivial example of loop fusion. However, with kernel definitions compiled separately from invocations, programmers must implement additional APIs to realize the benefits of chaining.

This paper builds on existing work by Zouhouri et al. [17] that analyzes and improves the performance of GPU-targeted OpenCL kernels on Intel FPGA devices using the Intel FPGA SDK for OpenCL. This paper expands compiler analyses to include both host and device compilation and introduces compiler transformations that benefit from sharing analysis information between the host and device compilation. In particular, we define three compiler transformations that transform OpenCL kernels to more closely match the best-practices published by Intel FPGA:

NDRange to Loop: Convert NDRange kernels, originally intended to be repeatedly executed with a range of thread identifiers (IDs), to a single body of code that uses a loop induction variable to represent thread IDs.

Restrict Parameters: Improve device-side alias analysis by inspecting host code and by marking kernel parameters with the `restrict` attribute where applicable.

While the above two transformations take advantage of the host and device compiler information sharing; they also enable the compiler to optimize the kernel code further by applying single-work-item-specific transformations, such as:

Reduction-Dependency Elimination: Improve pipelining of reduction loops by storing partial sums in a shift register to reduce loop-carried dependencies.

These transformations integrate with the Intel FPGA SDK for OpenCL and are evaluated on the Rodinia benchmark suite [6]. Rodinia benchmark OpenCL implementations target GPU-like devices and, as such, are an appropriate baseline for FPGA-specific transformations.

2 Programming FPGAs

FPGAs consist of arrays of interconnected programmable logic blocks, which vary in complexity from simple lookup tables to complete functional units. Typically, circuit configuration is specified via a Hardware Description Language (HDL) such as Verilog or VHDL. The HDL is then compiled into a 'bitstream' - a configuration file that sets the device's logic blocks and interconnect switches into a desired state. The compilation process consists of placing circuits specified by user HDL code to the chip, while considering chip area usage and interconnect length/congestion. The placed circuits then undergo routing, i.e., adding wires to correctly connect the placed components. Arriving at an optimal circuit configuration is a known NP-hard problem. Synthesis takes from hours to days.

To attract greater numbers of software developers to the platform, the field of High-Level Synthesis (HLS) emerged in the 1990s, experimenting with using higher levels of

ISBN 978-3-8007-4443-5

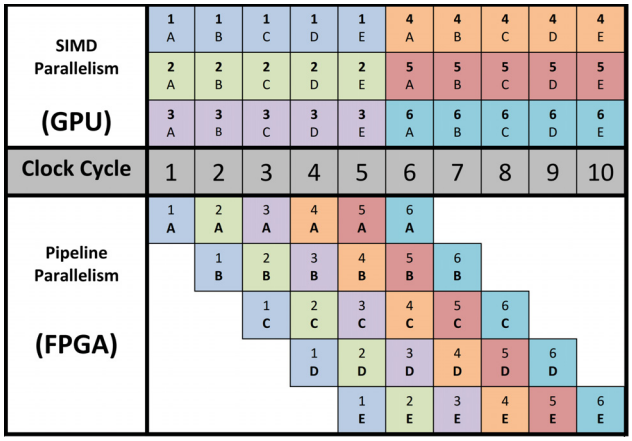

Figure 1 SIMD versus Pipeline Parallelism. From [4]. Letters represent instructions, numbers represent data elements.

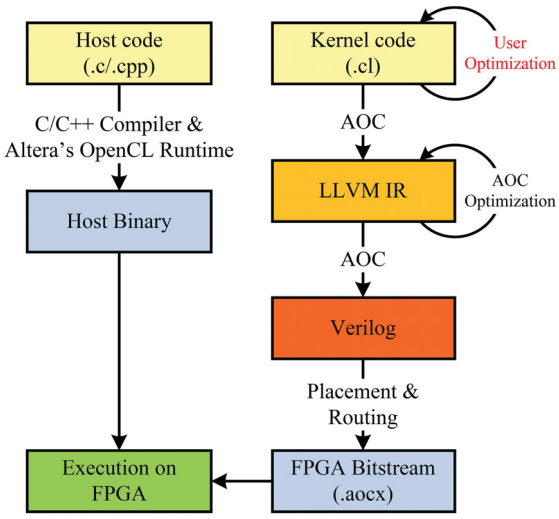

Figure 2 Altera OpenCL Compilation Flow. From [7].

abstraction to synthesize hardware from a C-derived language such as HardwareC or SystemC [12, 5]. HLS lets programmers focus on algorithmic functionality, and offloads the hardware implementation to a hardware compiler. If HLS tools can produce a circuit that is competitive with one described in HDL, this would result in a dramatic improvement in productivity and development velocity. Moreover, high-level languages often allow functional testing to be done by compiling to a CPU or GPU architecture, avoiding long hardware synthesis times during the development process. Various HLS approaches have been tried over the years with limited success. Such systems generally use proprietary ANSI C-based languages, cutting out useful language features and introducing extra constructs and pragmas, as well as a set of guidelines on a rigid programming style that must be followed for acceptable performance.

A recent push by Intel and Xilinx in the area of HLS focuses on adopting the OpenCL programming model. FPGA vendors are investing in the platform by providing end-to-end development toolchains such as Intel FPGA SDK for OpenCL, and Xilinx SDAccel.

2.1 High-Level Synthesis of OpenCL

OpenCL is an open standard for parallel programming of heterogeneous systems and a programming language specification [9]. General OpenCL program architecture consists of a host device that controls one or multiple compute devices by managing memory transfers and task distribution across devices. Compute devices are split into compute units, which, in turn, contain individual processing elements. OpenCL defines a Single Instruction, Multiple Thread (SIMT) data-parallel model where many threads execute the same instruction on many data items. In OpenCL terminology this model is called *NDRange* (for N-dimensional range). OpenCL also provides task-level parallelism that exploits concurrency through stand-alone task distribution across different compute units.

The main purpose of OpenCL is to enable portable use of various hardware accelerators. While already popular for GPU accelerators, recent adoption of the framework as an HLS input language has opened new opportunities to explore FPGA-specific compiler transformations. GPU-targeted programs rarely achieve acceptable performance when run unmodified on FPGAs [17], so new FPGA-specific compiler techniques and insights are required.

In contrast to the data-parallel model favoured by GPUs, Intel FPGA HLS tools follow a different approach when implementing the NDRange model: synthesized kernels execute instructions in a pipelined fashion as shown in **Figure 1**, similar to that of an assembly line. In an FPGA, this means that a data processing unit (e.g. a logic block) takes as input the output of a previous data processing unit. These units can perform concurrent computation because their work is independent from each other. The reconfigurable fabric on FPGAs makes SIMT parallelism a poor choice for applications. The pipeline parallelism model improves utilization by requiring fewer copies of each operator, while maintaining overall throughput [10].

2.2 Existing FPGA OpenCL Compilers

Czajkowski et al. present an LLVM-based OpenCL compiler prototype for Altera FPGAs, with a proof of concept executing on the Stratix DE4 [7]. This compiler represents all basic blocks of the program as Control-Data Flow Graphs (CDFG) with their own inputs and outputs as determined through live-variable analysis. The CDFG allows for efficient implementation of a module as a pipelined circuit, as opposed to finite state machine with a datapath - an approach much better suited to the data-parallel OpenCL model. This compiler implements the NDRange execution model by issuing individual work items into a kernel pipeline, one after another. The task-parallelism execution model, where the kernel code is written in a serial fashion, is implemented in the compiler by attempting to pipeline every loop in the code. The compiler also creates a wrapper for the generated kernel circuits to handle the standard

ISBN 978-3-8007-4443-5

interfaces to the device-memory IO and does all necessary bookkeeping to track kernel execution and to issue new work-items into the pipeline. This work subsequently became the Intel FPGA OpenCL compiler, upon which our optimizations are based. Internally, the compiler consists of an LLVM-based HLS component that compiles OpenCL kernel code into Verilog, which is then synthesized using standard Intel FPGA Quartus software. Intel FPGA also provides an implementation of the OpenCL API to allow host code to interface with devices: launch kernels, manage memory transfers, etc. **Figure 2** depicts the compilation flow as provided by Intel FPGA. A typical execution workflow consists of the host code programmatically loading a pre-compiled kernel binary file into the FPGA and initiating its execution. Intel FPGA also provides custom extensions to the OpenCL standard enabling certain architecture-specific user optimizations. While the prototype described here uses the Intel FPGA OpenCL toolchain targeting a Stratix V FPGA, the general concepts are applicable to reconfigurable architectures of other FPGA vendors.

As of 2015, Xilinx SDAccel development environment provides a HLS toolchain that is fully compliant with OpenCL 1.0. SDAccel is a closed-source application and little is known about its inner workings. We can infer from the user guide that the data-parallel NDRange execution model in this compiler is emulated through generation of a 3-dimensional loop-nest that iterates over the work-group and work-item dimensions [2]. Loop pipelining is one of the essential optimizations attempted by SDAccel [8]. SDAccel would also provide a good platform to prototype the analyses and transformations.

2.3 Manually Optimized OpenCL

Writing OpenCL code that delivers good performance on FPGAs is an open problem. Intel publishes a best-practices guide [1] detailing strategies and patterns that the Intel FPGA OpenCL compiler can efficiently execute. These patterns served as inspiration for the transformations presented in this paper. Zohouri et al. [17] performed manual optimizations on six Rodinia OpenCL benchmarks compiled with the Intel FPGA OpenCL compiler. After these optimizations, FPGAs could be competitive with GPUs on performance with dramatically better power efficiency. The effectiveness of these transformations inspired our compiler transformations.

2.4 Combined Host/Device Compilation

Lee et al. [13] developed an OpenACC-to-FPGA compiler framework, that converts OpenACC programs into OpenCL using the open-source OpenARC compiler, and then uses the Intel FPGA OpenCL compiler for HLS. The approach benefits from the fact that user-level source code contains device kernel code embedded into the host control code. This source code is annotated with pragmas that specify the code blocks that are offloaded to the device. Before the device code is outlined into a separate OpenCL kernel compilation unit, their compiler is able to take advantage of certain code-transformation opportunities that would not have been possible otherwise, such as bypassing global memory for inter-kernel communication using channels. To the best of our knowledge, our work is the first to implement combined compilation on OpenCL source code exposing similar transformation opportunities.

3 Optimizing OpenCL for FPGAs

Intel FPGA maintains a best practices guide for writing OpenCL that will execute efficiently on FPGAs [1]. A subset of these optimizations motivate the remainder of the work, and are summarized here.

3.1 `restrict` Pointers to Enable Simultaneous Memory Operations

FPGAs improve performance by executing multiple operations simultaneously. However, memory operations are defined to behave as if performed in program order, and can have extremely long latencies. If two memory accesses never reference the same memory address, then the compiler can safely reorder or overlap the operations. By marking a pointer passed as kernel parameters with `restrict`, programmers guarantee that any address accessible through that pointer is inaccessible through any other pointer. The Intel FPGA OpenCL compiler can then perform other memory operations simultaneously. As kernel parameters are passed opaquely from the host to the FPGA, it is otherwise extremely difficult for the compiler to prove that memory operations are safe to overlap. Memory operations require hundreds of FPGA cycles, so overlap is required for an efficient pipeline.

3.2 Prefer Single-Work-Item kernels over NDRange kernels

In an NDRange kernel, the same computation is executed by a large number of threads to support the data-parallel model. On FPGAs, chip area constraints prevent massively parallel processing units from being constructed. Instead, NDRange kernels are pipelined on FPGAs, allowing the stages to be executed concurrently such that subsequent threads can be started each cycle. All threads are executed in a single shared pipeline and thus values that do not differ between threads can be calculated once, and referenced from within the pipeline. However common intermediate products cannot be expressed in NDRange kernels, so single-work-item kernels are preferred.

Moreover, loops in NDRange kernels would have to be fully unrolled to support efficient pipelined execution because a pipeline is constructed across thread invocations. By converting an NDRange kernel to a single work item, loop exchange can generate pipelines where not otherwise possible. This conversion can also enable new transformations such as shift register reduction (described next). The effectiveness of the pipelined execution model depends on the target algorithm. Algorithms with little synchronization or control-flow may not benefit from single work-item execution at all and will have better performance with NDRange kernels.

ISBN 978-3-8007-4443-5

```
1 __kernel
2 void double_add_1(__global double *arr, int N,
3                   __global double *result)
4 {
5   double temp_sum = 0;
6   for (int i = 0; i < N; ++i)
7     temp_sum += arr[i];
8   *result = temp_sum;
9 }
```

Figure 3 Floating-point reduction sample preventing loop pipelining. From [1].

3.3 Pipelining Reduction Operations with Shifting Arrays

The performance of the single work-item execution model depends on the ability to pipeline loops in the kernel code. Thus, removing loop-carried dependencies is especially important because such dependencies induce longer loop initiation intervals. Reduction operations, such as the `double_add_1` method shown in **Figure 3**, cannot be pipelined well because the intermediate value `temp_sum` must be computed for each iteration before the next iteration can begin. Floating-point operations are relatively slow, causing the FPGA to stall for the majority of the computation. Addressing a similar problem in the context of software loop pipelining in 1992, Rau et al. [14] first introduced a technique called *modulo scheduling* that employs a rotating register file as a means to achieve a more compact loop schedule and thus reduces the loop initiation time. This technique was later implemented in hardware in the Intel IA-64 architecture [15].

As suggested in the best-practices guide [1] and implemented by hand by Zouhouri et al. [17], a variation of the rotating register technique can be employed by the programmer manually to minimize pipeline delays caused by the intermediate value in a reduction operation. **Figure 4** shows the same reduction, but with the introduction of a local array to emulate a shift register. Instead of reducing elements of `arr` into a single variable, they are accumulated into a shift register. The shift register's depth is equal to the latency, in cycles, of the floating-point operations that form the dependency. Reduction input is read from the first element of the shift register, and written into the last. Effectively, this reduces the initiation interval of the loop to 1 cycle. After the loop completes, an extra reduction on the shift register contents produces the final reduction value. Because the final shift register summation loop has a smaller trip count, the improved initiation interval of the original loop yields an overall performance improvement. The Intel FPGA OpenCL compiler looks for the pipelining idiom from **Figure 4** in OpenCL code, and efficiently implements it using a shift register in hardware.

4 Compiling OpenCL for FPGAs

The transformations suggested in the Intel FPGA best-practices guide are meant to be performed by programmers. However, a sufficiently capable compiler should be able to automatically transform OpenCL device code to de-

```
1 __kernel
2 void double_add_2(__global double *arr, int N,
3                   __global double *result)
4 {
5   double shift_reg[II_CYCLES+1];
6   //Initialize all elements of shift_reg to 0
7   for(int i = 0; i < N; ++i)
8   {
9     shift_reg[II_CYCLES] = shift_reg[0]+arr[i];
10    #pragma unroll
11    for(int j = 0; j < II_CYCLES; ++j)
12      shift_reg[j] = shift_reg[j+1];
13  }
14  double temp_sum = 0;
15
16  #pragma unroll
17  for(int i = 0; i < II_CYCLES; ++i)
18    temp_sum += shift_reg[i];
19  *result = temp_sum;
20 }
```

Figure 4 Floating-point reduction using a shift register to enable loop pipelining. From [1].

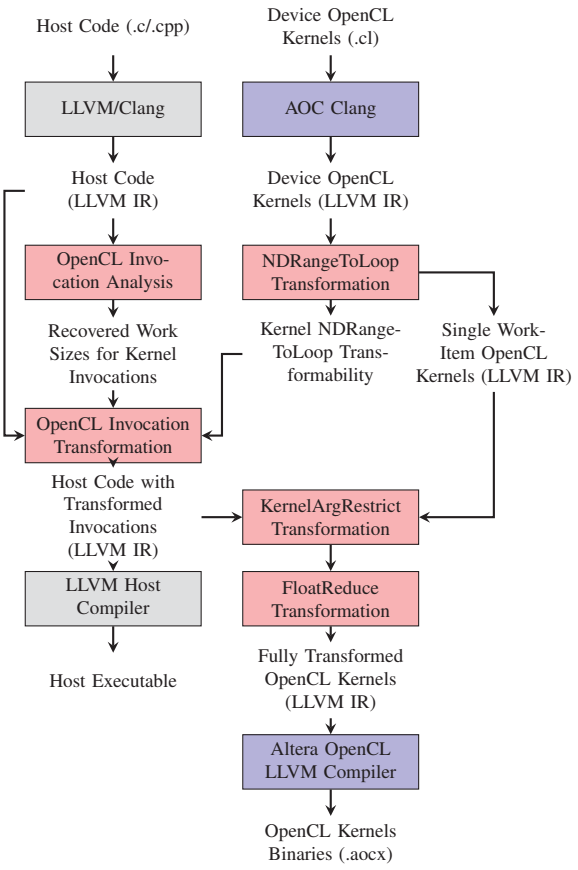

Figure 5 Custom OpenCL Compilation Flow

ISBN 978-3-8007-4443-5

liver more efficient execution on FPGAs. Thus, we integrated some of the transformations into the Intel FPGA SDK for OpenCL compiler. This compiler is a closed-source application based on the LLVM compiler infrastructure. Thus, arbitrary compiler passes targeting LLVM 3.0 can modify the Intermediate Representation (IR) .

Our transformations are performed early in the compilation process because they attempt to automate best practices when writing source code. OpenCL is designed to allow for the separate compilation of host and device code. However, combined compilation allows for optimizations not previously possible. Moreover, the Intel FPGA compiler already requires some degree of such coordination by the user. For instance, the compiler may generate single-work-item code for a kernel that the host invoked in NDRange mode. Our compiler passes make use of coordination between the host and device code compilation processes, passing information between the two to enable certain transformations. A custom compiler driver facilitates combined compilation, accepting as input the host and device source-code files. The driver coordinates between the Intel FPGA compiler modified with our transformations and the host compiler, based on LLVM 4, with modifications to analyze and transform host-to-device communication. To integrate our three transformations with the Intel FPGA OpenCL Compiler, the host code analyses are choreographed with the device code transformations (**Figure 5**).

4.1 NDRange to Single Work-Item Loop (NDRangeToLoop)

Under the OpenCL NDRange model, a kernel function is invoked for a number of threads (work-items) that can cooperate and synchronize within a work-group. This model maps extremely well to GPUs, which have many Single Instruction, Multiple Data (SIMD) processors, but makes it difficult for kernel functions to express common work products. On FPGAs, where parallel kernels are implemented using pipelines, factoring out common work is key to improving performance. OpenCL allows thread and work-group sizes to be specified in three dimensions, denoted here as Z, Y, and X. The conversion of a NDRange kernel into a single work-item, transforms the kernel body into a series of loops over the respective n dimensions. Each loop executes the original kernel body for each thread. Thread ID references are remapped to the appropriate loop induction variable. Unfortunately, however, the number of dimensions, size, and count of work-groups are specified in host code and are inaccessible to device compilation. An example host invocation is shown in **Figure 6**.

Thus it is necessary to recover the number of dimensions, the starting indices of threads in each dimension, the number of threads in each dimension, and the number of threads per work-group in each dimension. To do so we created a host transformation that injects dummy functions that take as argument each value of interest. After this transformation, standard LLVM passes for interprocedural constant propagation are applied. Calls to the dummy function are inspected, and constant arguments are transmitted to the device compiler. This technique can be easily extended to share arbitrary constants and ranges for device kernel parameters, allowing additional device code specialization. For generality, the kernel function signature is first modified to take as argument the `work_dim`, `global_work_offset`, `global_work_sizes` and `local_work_size` values. Then, the results of the host NDRange invocation analysis are read, and used in place of the kernel arguments when available.

```
1  size_t work_dim = 2;
2  size_t gbl_offset[2] = {0, 0};
3  size_t gbl_size[2] = {64, 8};
4  size_t lcl_size[2] = {32, 1};
5  clEnqueueNDRangeKernel(
6         cmd_queue, kernel, work_dim,
7         gbl_offset, gbl_size, lcl_size,
8         wait_list_size, wait_list, event);
```

Figure 6 Host NDRange kernel invocation

As long as a kernel contains no synchronization points, which can be verified by checking the kernel for calls to the OpenCL `barrier()` function, the NDRange execution can be emulated through a single loop nest. To emulate work-groups, the kernel body is wrapped in loops corresponding to any dimensions for which `get_group_id()` is accessed. Calls to `get_group_id()` are then replaced with accesses to the appropriate loop induction variables. Next, loops are inserted to emulate work-items within each work-group, replacing accesses to `get_local_id()` and `get_global_id()` with the appropriate expressions on the loop induction variables. Finally, accesses to the invocation dimensions are replaced. As an example, a Hello World kernel is shown in **Figure 7**. After applying the transformations above, the kernel appears as depicted in **Figure 8**. Several optimizations can be performed at this point. Single-iteration loops can be elided entirely, and dead code elimination can remove the computation of unnecessary values. By applying these optimizations, the single-work-item example can be simplified to **Figure 9**.

In NDRange kernels, work-items must halt execution at barrier points until all other work-items in the work-group reach the same point. This behavior can be preserved after NDRangeToLoop transformation by splitting work-item loop nests at each barrier point. Our prototype NDRange-ToLoop transformation can only convert NDRange kernels with barriers in top-level control flow, outside of any conditional statements or loops. When such barriers are encountered, the kernel is partitioned into unsynchronized sections, and each section is wrapped separately into work-item loop nests.

If values are calculated before and used after a barrier, they must be preserved across work-item loops. These values are identified by inspecting instruction operands, and then collecting operands calculated in a different partition than the user. Local arrays equal in size to the work-group are allocated, and each operand is saved into the array as it is calculated. Then, uses in other partitions are redirected to the allocated array.

ISBN 978-3-8007-4443-5

```
__kernel void hello_world(int tid) {




  unsigned thread_id = get_global_id(0);

  if (thread_id == tid) {
    printf("foo #%u: Hello!\n", foo);
  }


}
```

Figure 7 HelloWorld Kernel. From [3].

```
__kernel void hello_world(int tid,
    int offset_x, int offset_y, int offset_z,
    int global_x, int global_y, int global_z,
    int local_x, int local_y, int local_z) {
  int group_sz_x = (global_x-1) / local_x+1;
  for (int c = 0; c < group_sz_x; c++) {
    for (int f = 0; f < local_x; f++) {
      unsigned thread_id = (c *
                local_x + offset_x) + f;
      if (thread_id == foo) {
         printf("tid #%u: Hello!\n", tid);
      }
    }
  }
}
```

Figure 8 After NDRange [3].

```
__kernel void hello_world(int tid,
    int offset_x, int offset_y, int offset_z,
    int global_x, int global_y, int global_z,
    int local_x, int local_y, int local_z) {

  for (int c = 0; c < 2 ; c++) {
    for (int f = 0; f < 32; f++) {
      unsigned thread_id = (c * 32) + f;

      if (thread_id == foo) {
         printf("tid #%u: Hello!\n", tid);
      }
    }
  }
}
```

Figure 9 After Constant Propagation [3].

After transforming the NDRange kernel to a single work-item kernel, it is necessary to transform the kernel invocation on the host. To invoke transformed device kernels appropriately, each `clEnqueueNDRangeKernel()` invocation is replaced with the following routine:

1. Ensure that the kernel about to be executed is one which was transformed. If not, invoke with the original call to `clEnqueueNDRangeKernel()`
2. Pass the original dimensions and work-group sizes and counts as arguments through `clSetKernelArg()`.
3. Invoke the kernel with `clEnqueueTaskKernel()` for single work-item execution.

The prototype implementation of the NDRange transformation described and evaluated in this paper has some limitations. Its NDRangeToLoop cannot handle barriers inside control-flow. Also, the work-group size must be known at compile-time to enable the allocation of the array to allow uses across partitions.

4.2 Reduction-Dependence Elimination

The floating-point reduction-dependence elimination transformation implements an idiom suggested by the Intel FPGA Best Practices Guide [1] as a technique to remove loop-carried dependencies by inferring shift registers for loops that carry out floating-point reductions, as demonstrated by going from **Figure 3** to **Figure 4**.

First, an analysis detects all reduction idioms that are safe to transform. All loops that do not contain other loops are scanned for reduction expressions. A reduction expression a store to a value where an operand of the stored value is obtained from a load from the same address. The pattern-matcher handles two cases: when the reduction value is accessed through a pointer with no offset, and when the reduction value is a memory location specified via a base address and an indexing expression. The latter case requires the use of exactly the same indexing expression for both the store and the corresponding load. Once a reduction expression is found, the analysis must verify if it is safe and beneficial to apply the transformation. To do so, the analysis performs the following checks:

- The type of the reduction value must be either 32-bit or 64-bit floating-point.
- The reduction value must not be used elsewhere in the loop body other than in the reduction operation.
- If reduction is done on an array element, the indexing expression and the array base pointer must be loop invariant.
- The binary operations that constitute the reduction must be associative and commutative.
- If the loop trip count is known at compilation time and is less than the shift register size, the reduction should not be transformed.

The final loop trip-count check warrants further explanation: calculation of the final reduction value requires the computation of the sum of the values in the shift register, which is a loop with exactly the same type of loop-carry dependency that was eliminated in the transformed reduction loop. This summation loop has a trip count equal to the number of elements of the shift register. Thus, the transformation is only beneficial when the number of original reduction iterations exceeds the size of the shift register, which is a compiler-specified constant suitable to the target FPGA. The shift register must have enough elements to cover the latency of floating-point operations that would prevent pipelining. In the prototype, targeting the Intel FPGA Stratix V, this constant is eight, which is the floating-point operation latency for the device.

Code generation consists of the following steps: a shift register array is created for a given reduction operation. All its values are initialized to zero in the loop pre-header. The original reduction statement is then rewritten to one that instead performs a store into the shift register's tail element. Immediately after the reduction expression, the values of the shift register are shifted down. In the loop epilog, the final reduction value is computed by performing a sum over all shift register values and is stored into the original intended reduction value.

4.3 Restrict Pointer Kernel Parameters

When creating a buffer with the OpenCL API `clCreateBuffer()` function a programmer can use the `CL_MEM_USE_HOST_PTR` to indicate that the buffer should use memory referenced by the host [11]. Thus, the prototype assumes that in files that do not contain the `CL_MEM_USE_HOST_PTR` flag there are no overlapping buffers. Based on this assumption the prototype marks all global pointers as `restrict` for kernels in these files. In a refinement to this approach, the compiler would first mark all buffers that are not allocated in the host memory and

ISBN 978-3-8007-4443-5

then exclude only the kernels that use more than one such buffer from having their parameters marked `restrict`.

5 Prototype Performance Study

Several unmodified OpenCL kernels from the Rodinia benchmark suite, form the baseline for a study of the prototype performance. To generate transformed kernels all the transformations described in this paper are enabled unless otherwise specified. In both the baseline and transformed benchmarks, the host code that loads and launches kernels had to be hand-modified to load kernels from FPGA-synthesized binaries, rather than compile them from source at runtime. We evaluate only benchmarks that NDRangeToLoop can transform, limiting ourselves to `gaussian`, `hotspot3D`, `kmeans`, `nn`, and `SRAD`. The remaining benchmarks either fail to compile under the Intel FPGA OpenCL compiler, or are unaltered by our transformations. Out of the benchmarks tested, pointer `restriction` was applied to `hotspot3D`, `kmeans`, and `SRAD`. Reduction-dependence elimination applies to the `gaussian` and `kmeans` benchmarks.

Performance was evaluated on a Terasic DE5-Net board that contains an Altera Stratix V GX FPGA with 4GB of 1600 MHz DDR3 memory. The board is connected to a machine with an Intel Core i7-4770 CPU with 32GB of DDR3 memory, running CentOS 6.7 (Linux 2.6.32). We use the Intel FPGA SDK for OpenCL version 16.1.0.196. Our transformations on the device code are implemented against the SDK-compatible LLVM 3.0, while the host code transformations and analyses are implemented with the LLVM 4 compiler.

Each benchmark was run ten times for both the baseline and the transformed versions with the mean overall execution time reported. Minimal variance was observed between runs of a given program, never exceeding 0.5% of the mean. Thus the variance is not reported.

5.1 Benchmarks

Table 1 shows transformed kernel execution time and ratio over untransformed kernels. The five benchmarks handled by this prototype implementation can be divided into three groups according to the performance in relation to the baseline. For `gaussian` and `hotspot3d`, the transformed code is significantly slower than the baseline. `nn` and `srad` have roughly baseline performance; and `kmeans` is 2.6× faster than the baseline.

5.1.1 `gaussian`

This kernel contains a loop with a memory dependence on load operations. Before our transformations, the performance impact of the memory dependence is mitigated because multiple work-items can be simultaneously executed. After NDRangeToLoop however, the pipelining opportunities are obscured, because our analysis is unable to determine that the various kernel parameters are independent. Though the introduced loops can be pipelined, the load and store operations must be executed in order to preserve semantics, and this effect cannot be mitigated by operator duplication as before our transformation. Performance could be improved either by not applying NDRangeToLoop kernel parameters are not marked restrict, or by exposing heuristics from the underlying compiler.

5.1.2 `hotspot3D`

This kernel contains a loop-carried dependence. Some loops are therefore not pipelined, which degrades the overall performance. Additional heuristics that measure overall kernel capacity for pipelining across all loops would be useful to decide when NDRangeToLoop transformation would be beneficial.

5.1.3 `nn` and `srad`:

Algorithms with little control-flow may not benefit from single work-item execution. In these benchmarks, the kernels transformed contained at most two conditional branches. As a result, both the baseline and transformed benchmarks can issue a new thread each cycle.

5.1.4 `kmeans`

Both the NDRangeToLoop and Restrict transformations are applied to this kernel. One of the kernels in `kmeans` contains a nested loop. Performance improves dramatically because this nested loop can be fully pipelined only with the NDRangeToLoop transformation.

Reduction-dependence elimination for `kmeans` was disabled because it degraded performance. The reduction loop nests in a loop that is already fully-pipelined, and the resulting improved reduction loop initiation interval means that a new loop iteration is dispatched every cycle for this and the outer loop. As a result, the loop induction variable increment and the comparison between the IV and the loop upper bound, together, form 87% of the kernel critical path (according to the Intel compiler optimization report). In turn, the FPGA is forced to reduce the operating frequency to accommodate the number of integer operations that must be performed simultaneously on the induction variables of both loops in the loop nest. There is no way for this prototype implementation to perform this kind of analysis without access to the Intel compiler's internals that perform loop pipelining. However, with access to the pipelining code, an analysis could be added to the transformation that would detect these conditions and deem the transformation unprofitable.

5.2 Reduction-Dependence Elimination Efficacy

To measure the impact of the floating-point reduction dependence elimination as a stand-alone transformation, we evaluated several applications where the transformation finds opportunities to apply this transformation. For applications where the transformation analysis finds no opportunities, the code is left untouched so there is no impact. The following data points are not general because few single work-item example kernels are available for evaluation;

ISBN 978-3-8007-4443-5

Benchmark	Execution Time Baseline (s)	Execution Time Transformed (s)	Ratio	Restrict	NDRangeToLoop	FloatReduce
gaussian	0.28	1.85	6.69	✘	✔	✔
hotspot3D	9.65	25.12	2.60	✔	✔	✘
kmeans	37.52	13.43	0.36	✔	✔	✘*
nn	0.05	0.05	0.98	✘	✔	✘
srad	105.50	111.70	1.06	✔	✔	✘

Table 1 Benchmark Execution Time and Applicable Transformations

rather, they serve as motivating examples of the kinds of gains that are possible. The data is not presented in an aggregated fashion because the transformed code varies in source: some are hand-written kernels taken from [17], some are Rodinia benchmark kernels transformed using the prototype toolchain.

The single work-item version of the `srad` benchmark, taken from [17] (the baseline single work-item implementation), which has a frequently executed reduction kernel, sees a 2.6× decrease in overall kernel execution time with the transformation applied.

A hand-written single work-item version of the `lud` benchmark, from [17], with the transformation applied only sees a 7% improvement in kernel execution time. Despite catching several opportunities in a hot region of the kernel code, the impact is small because the reduction loop has a varying trip count that depends on the iteration of the containing loop. As a result, some reductions are smaller than the size of the shift-register and some are larger. The effect is that these two cases cancel each other out, yielding a marginal overall improvement. A classical solution to this pattern would be to version the loop into a portion with a small trip-count that maintains the original reduction pattern, and a portion with a sufficiently large trip-count that is transformed. However, versioning is costly for FPGAs because it results in higher resource utilization. In our experiments, loop versioning yielded no benefit.

The `gaussian` benchmark, after our NDRangeToLoop transformation, executes 3.3× faster with the reduction transformation applied. One of the two kernels in this benchmark consists of a single reduction operation on an array element. After NDRangeToLoop transformation this reduction becomes a single hot loop.

The only case where eliminating reduction dependence leads to performance degradation is in the `kmeans` benchmark, as described in section 5.1.4. That scenario appears to be anomalous. However a more robust implementation of the transformations should include a more thorough profitability analysis. Such analysis would be best implemented with full access to the source code for the entire software toolchain.

6 Conclusion

Combining compilation of device kernel code and host code these compilation paths allows for inter-compiler communication which, in turn, enables new, previously impossible, compiler transformation opportunities. We have implemented three transformations for OpenCL execution on FPGAs using the combined compilation toolchain and studied their performance. The variable performance across benchmarks indicates that more analysis is required to determine when transformations should be applied. A sophisticated analysis could prevent transformations from occurring when it would be unprofitable; more specifically, having access to the loop pipelining code of the Intel FPGA compiler would allow for the application of the NDRangeToLoop transformation only when the kernel can be pipelined successfully. Access to such code and analyses would also help with the issue encountered when applying the reduction-dependency elimination transformation to the `kmeans` benchmark.

This work is a step forward in the automatic optimization of OpenCL applications for FPGA execution. While these transformations were only tested on a small number of benchmarks, and experienced varying levels of success, the performance improvements seen show that these techniques, when applied judisciously, can dramatically improve program performance without programmer involvement. Such automatic transformations will play a key role in continued FPGA adoption, as specialized device knowledge can be further reduced and performance of generic OpenCL programs made truly portable.

Acknowledgements

This work was made possible by grants from the Intel FPGA university program.

7 Literature

[1] Intel FPGA OpenCL Best Practices Guide. URL `http://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf`.

[2] SDAccel Environment User Guide. URL `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_3/ug1023-sdaccel-user-guide.pdf`.

[3] Altera. Hello world design example. URL `https://www.altera.com/support/support-resources/design-examples/design-software/opencl/hello-world.html`.

[4] Altera. OpenCL on FPGAs for GPU programmers, 2014. URL `https://www.altera.com/`

content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-201406-acceleware-opencl-on-fpgas-for-gpu-programmers.pdf.

[5] G. Arnout. Systemc standard. In *Design Automation Conference (DAC)*, pages 573–577, June 2000.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[7] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to high-performance hardware on FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534, 2012.

[8] J. Fifield, R. Keryell, H. Ratigner, H. Styles, and J. Wu. Optimizing OpenCL applications on Xilinx FPGA. In *International Workshop on OpenCL*, pages 5:1–5:2, New York, NY, USA, 2016. ACM.

[9] K. O. W. Group. "the opencl specification: Version 1.0", october 2011., 2011. URL https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf.

[10] S. Hauck and A. DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation*, volume 1. Morgan Kaufmann, 2010.

[11] Khronos. clcreatebuffer. URL https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clCreateBuffer.html.

[12] D. Ku and G. D. Micheli. High-level synthesis and optimization strategies in hercules and hebe. In *European Event in ASIC Design EURO ASIC*, pages 124–129, 1990.

[13] S. Lee, J. Kim, and J. S. Vetter. OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 544–554, 2016.

[14] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Programming Language Design and Implementation (PLDI)*, pages 283–299, 1992.

[15] H. Sharangpani and H. Arora. Itanium processor microarchitecture. *IEEE Micro*, (5):24–43, Sep 2000.

[16] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering (CSE)*, 12(3):66–73, 2010.

[17] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 35:1–35:12, 2016.

ISBN 978-3-8007-4443-5