2016 27th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)

# Bridging the FPGA Programmability-Portability Gap via Automatic OpenCL Code Generation and Tuning

Konstantinos Krommydas
Dept. of Computer Science
Virginia Tech
kokrommy@vt.edu

Ruchira Sasanka
Intel Corporation
ruchira.sasanka@intel.com

Wu-chun Feng
Dept. of Computer Science
Virginia Tech
wfeng@vt.edu

***Abstract*—Programming FPGAs has been an arduous task that requires extensive knowledge of hardware design languages (HDLs), such as Verilog or VHDL, and low-level hardware details. With OpenCL support for FPGAs, the design, prototyping and implementation of an FPGA is increasingly moving towards a much higher level of abstraction, when compared to the intrinsically low-level nature of HDLs. On the other hand, in the context of traditional (i.e., CPU) software development, OpenCL is still considered to be low-level and complex because the programmer needs to manually expose parallelism in the code. In this work, we present our approach to enhancing FPGA programmability via GLAF, a visual programming framework, to automatically generate synthesizable OpenCL code with an array of FPGA-specific optimizations. We find that our tool facilitates the development process and produces functionally correct and well-performing code on the FPGA for our molecular modeling, gene sequence search, and filtering algorithms.**

## I. Introduction

FPGA programming using OpenCL, as introduced by Altera and Xilinx, facilitates exploiting the FPGA's reconfigurable nature and energy efficiency. While, OpenCL is considered high-level when compared to HDLs, it still remains a low-level, complex language for the typical programmer, and especially for the domain scientists (e.g., physicists, engineers). Eliminating the existing productivity/programmability issues can position FPGAs as a potential prime solution in the heterogeneous computing landscape. To that end, we present GLAF-OCL, our work in extending GLAF [1], a grid-based language and auto-tuning framework that provides a visual programming environment for parallel computing. Specifically, we add support for automatic OpenCL code generation and show how using GLAF-OCL allows rapid prototyping of OpenCL applications on reconfigurable targets.

## II. GLAF Visual Programming Framework

GLAF [1] is a visual code-generation and auto-tuning framework that aspires to facilitate parallel algorithm implementation by domain experts with minimal programming knowledge. Programming using GLAF deviates from the typical text-based programming paradigm. Instead, it employs visual, point-and-click programming (no code is actually *written*) through an intuitive GUI that incorporates code building and data visualization (Figure 1). GLAF contains three back-ends that implement the following core functionalities: *automatic code generation*, *auto-parallelization*, and *auto-tuning*. The former auto-generates code in a set of target languages, while the latter two back-ends support the former by: a) performing *parallelism analysis*, b) *automatic tuning* with respect to options from which the user can select (e.g., different data layouts, loop collapsing, loop interchange).

This work was supported in part by Intel Corporation.

## III. Methodology

### A. Automatic OpenCL Code Generation

Automatic OpenCL code generation produces three files (Figure 2): a host code (.c) file, a device code (kernels) file (.cl) and a header file (.h) that contains the OpenCL boilerplate code and auxiliary functions.

*1) Main host and device code generation:* GLAF-OCL uses the existing GLAF parallelism analysis back-end [1], in which parallelism opportunities are identified at the *GLAF step* (i.e., basic computation building block) level. For exposing parallelism via OpenCL the step's body is converted to an OpenCL kernel. GLAF-OCL code generation back-end parses a parallel loop's internal representation and converts the loop to the kernel's *NDRange* (global work size dimensions partitioning). This happens as follows (Figure 3, *line 6* of .c file): each loop index (e.g., *row, col, ind2*) represents a global dimension in the NDRange. The value of each global dimension corresponds to the number of loop's iterations across this dimension.

The parallel step's body is replaced with a kernel call (clEnqueueNDRangeKernel()) to the generated OpenCL kernel in the separate .cl file (e.g., Figure 3, *line 20* of .c file). The kernel call is preceded by a series of calls (clSetKernelArg() - Figure 3, *lines 15-18* of .c file) for setting the appropriate kernel arguments as identified by using the GLAF internal representation of grids in the current step. Last, the kernel call is followed by a clFinish() call that also functions as a synchronization point that enforces coherence between the memory contents of the host and device.

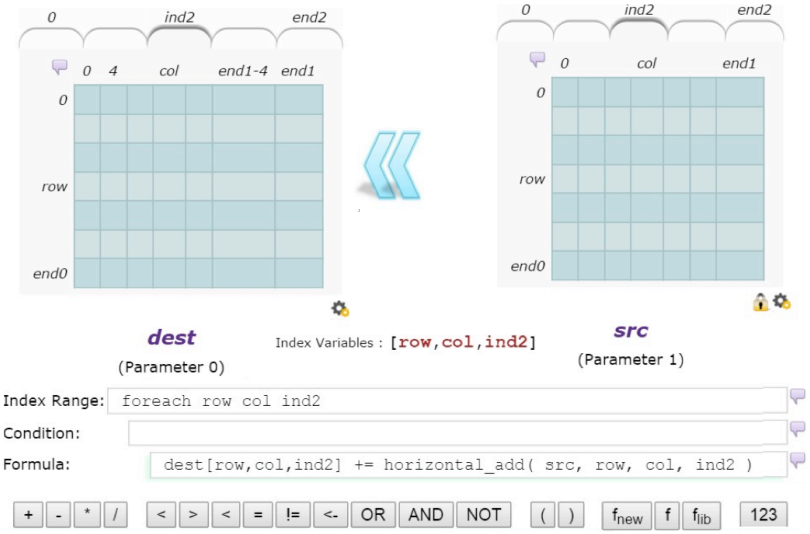



Fig. 1: GLAF Graphical User Interface.

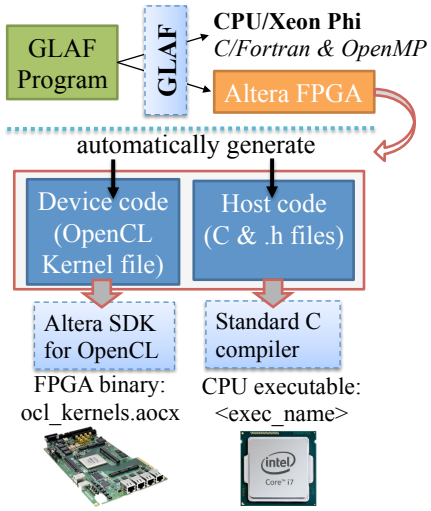



Fig. 2: GLAF-OCL Development Workflow.

As far as device code is concerned (.cl file), *kernel functions* and *device functions* are generated along with their parameters (in the form of *__global* pointers for dynamically allocated memory or as scalar variables). Data parallelism in kernel functions (as per the OpenCL SPMD paradigm) is achieved by accessing different memory locations based on the combination of work-group/work-item IDs (Figure 3, *lines 5-7* of .cl file). Obtaining the index for each dimension takes into account the *start* and *step* values for each index variable of the original loop that is being encapsulated in the OpenCL kernel (star-annotated lines in Figure 3).

*2) Host/device memory considerations:* In OpenCL, when the host and device (e.g., CPU and FPGA) have separate memory address spaces, memory needs to be allocated in both. GLAF-OCL parses a step's grid objects' internal representation (name, size, data type, etc.) and automatically generates appropriate code for declaring and allocating space for cl_mem device-side buffers and passes them as needed to the kernel code.

As execution alternates between host and device, data is transferred between the two. GLAF-OCL generates the code (in the form of wrapper functions - e.g., *h2d_tran()*, Figure 3, *lines 12-13* of .c file) that is responsible for host-to-device and device-to-host data transfers, ensuring data coherence across host and device memory spaces, and elimination of redundant data transfers. The above scheme is implemented by tracking allocation and read/write accesses to the non-scalar grids in host and device code and by performing appropriate checks at run-time before each GLAF step.

*3) Data linearization:* Disjoint memory spaces in an OpenCL execution scenario (e.g., CPU host/FPGA device) impose inherent limitations to passing structs that include pointer elements to a kernel. This kind of structs needs to undergo linearization/marshaling, i.e., a struct declaration needs to be expanded as multiple declarations of arrays/pointers of the respective type on the host-side and corresponding declarations need to take place for the device-side (cl_mem buffers). Initialization of the latter may be needed. All the above procedures, as well as parameter passing and struct element accesses are automatically handled by GLAF-OCL.

*4) Boilerplate OpenCL code:* Every OpenCL program requires an initialization and finalization procedure that, among others, selects the required OpenCL platform and OpenCL device, and initializes/finalizes various OpenCL objects (e.g., program, command queue(s)). GLAF-OCL obviates the need for programmers to familiarize themselves with the aforementioned procedure by auto-generating the appropriate boilerplate code. Last, GLAF-OCL provides wrapper functions for memory allocation that enforce the alignment requirements of Altera OpenCL, as well as data transfers (III-A2).

### B. Generation of Optimized Code for Altera FPGAs

GLAF-OCL performs certain FPGA-specific code optimizations and renders code amenable for further optimizations by the Altera Offline Compiler (AOC). Here, we describe such optimizations, while in Section IV we provide examples and discuss the effect of the most important ones.

**Single Work-Item (SWI) Kernels:** In Section III-A, we describe how GLAF-OCL generates device code in the form of *NDRange kernel* (i.e., multiple work-groups and work-items). In FPGAs, constructing a kernel as a *SWI kernel* (equivalent to an OpenCL *task*) may be a more appropriate paradigm. This method utilizes *loop pipelining* and can enable further optimizations that are not applicable (or beneficial) with NDRange. With SWI kernel code generation the generated host code defines a single-dimension, single-item *globalWorkSize[]* array. On the device code the kernel contains the loop itself (i.e., the parallel loop that was previously converted to NDRange) and the *get_global_id()* calls are omitted. AOC identifies the kernel as a SWI kernel and attempts to infer a loop pipeline.

**Initiation Interval (II) Reduction:** SWI kernel compilation with AOC yields an optimization report that informs whether pipelined execution was inferred, what the *initiation*

**GLAF step code (as filled automatically using the GUI)**
*(parallelism identified at the granularity of a GLAF step)*

```
foreach row col=dest(4:end1-4:1) ind2
```

b

```
    dest[row,col,ind2] += horizontal_add( src, row, col, ind2);
```

a

a Generation of OpenCL kernel code for GLAF steps that are identified to be parallelizable.

```
__kernel void ft_st_3dfd_1_dev(__global double * restrict ft_dest,
                               int ft_grid_dimen,
                               __global double * restrict ft_src,
                             * __global int* restrict index_dat) {
1  int ft_row;
2  int ft_col;
3  int ft_ind2;
4  double res_horizontal_add_device;

5  ft_row= get_global_id(0);
6* ft_col= get_global_id(1)+index_dat[2];
7  ft_ind2= get_global_id(2);

8  ft_dest[ft_ind2*ft_grid_dimen*ft_grid_dimen + ft_row*
           ft_grid_dimen + ft_col] +=
9  ft_horizontal_add_device(ft_src,ft_grid_dimen, ft_row, ft_col,
                            ft_ind2);
}
```

.cl file

b Generation of appropriate OpenCL host code for parallelizable step (transformation of loop into an OpenCL kernel call to the kernel generated for the step).

```
int ft_st_3dfd(cl_mem *ft_dest_dev, double *ft_dest, int ft_grid_dimen,
               cl_mem *ft_src_dev, double *ft_src) {
   (...)
1  ft_end0 = ft_grid_dimen-1;
2  ft_end1 = ft_grid_dimen-1;
3  ft_end2 = ft_grid_dimen-1;
4* index_data[2] = 4;
5  //Converting parallel loop to NDRange dimensions.
6  size_t globalWorkSize_s0[3] = {ft_end0+1, ft_end1-4-4+1, ft_end2+1};
7  //Creating kernel (generated in .cl file from loop's body).
8  kernel_cl = clCreateKernel(program, "ft_st_3dfd_1_dev", &error);
9  assert(error == CL_SUCCESS);
10 //Data transfers.
11* clEnqueueWriteBuffer(commands, index_data_dev, CL_TRUE, 0,
                  sizeof(int)*9, (void *)index_data, 0,NULL, NULL);
12 h2d_tran(ft_dest, sizeof(float)*ft_grid_dimen*ft_grid_dimen*
          ft_grid_dimen);
13 h2d_tran(ft_src, sizeof(float)*ft_grid_dimen*ft_grid_dimen*
          ft_grid_dimen);
14 //Setting kernel arguments (based on parallel step's grids).
15 clSetKernelArg(kernel_cl, 0, sizeof(cl_mem), ft_dest_dev);
16 clSetKernelArg(kernel_cl, 1, sizeof(int), (void* )&ft_grid_dimen);
17 clSetKernelArg(kernel_cl, 2, sizeof(cl_mem), ft_src_dev);
18* clSetKernelArg(kernel_cl, 3, sizeof(cl_mem), &index_data_dev);
19 //Enqueuing OpenCL kernel for execution.
20 clEnqueueNDRangeKernel(commands, kernel_cl, 3, NULL,
                         globalWorkSize_s0, NULL, 0, NULL, NULL);
21 clFinish(commands);
   (...)
}
```

.c file

Fig. 3: GLAF-OCL Auto-Generated Code.

*interval* between successive loop iterations is and, if possible, the reason. In GLAF-OCL SWI execution for parallelizable loops leads to successful pipeline inference. In certain cases, however, II may be high. While, the range of such cases is too broad to conclusively address, we show how GLAF-OCL can automate *loop relaxation* [2]. These optimizations can drastically reduce or eliminate II for *reduction* operations within a SWI kernel. An example is shown in Figure 4. GLAF parallelism analysis back-end identifies reductions and stores the reduction variable and operation in its internal representation. After compiling code, parsing the optimization report reveals the II value and whether it is reduction-induced.

```
1  float sum = 0;
2  for (i = 0; i < N; i++) {
3      sum += A[i];
4  }
5  result[idx] = sum;
```

a) Original Code

```
1  float sum_copies[M];
2  for (i = 0; i < M; i++) {
3      sum_copies[i] = 0;
4  }
5  for (i = 0; i < N; i++) {
6      float cur = sum_copies[M-1] + A[i];
7      #pragma unroll M-1
8      for (j = M-1; j > 0; j--) {
9          sum_copies[j] = sum_copies[j-1];
10     }
11     sum_copies[0] = cur;
12 }
13 #pragma unroll M
14 for (i = 0; i < M; i++) {
15     sum += sum_copies[i];
16 }
17 result[idx] = sum;
```

b) Code after II Reduction

Fig. 4: Initiation Interval (II) Optimization.

In this case, GLAF-OCL can automatically generate *M* copies of the reduction variable (line 1, Figure 4b), and code (lines 2-4) for initialization of this variable according to the reduction operation (e.g., zero for addition). The main computational loop now is transformed to a temporary variable in which we store the reduction operation on the last reduction variable copy (line 6), a loop that shifts all copies by one position (lines 7-10), and code for storing the temporary variable to the first copy (line 11). Finally, code for reduction on the reduction variable copies is generated outside the reduction loop (lines 13-16) and the result is assigned to the original reduction variable (line 17). This method relaxes the dependencies and reduces II. The key change according to the problem at hand lies on substituting A[i] with the corresponding computation of the reduction at hand. The number of copies M is the important factor in reducing II. Different values can be attempted manually (by changing a simple *#define*) or automatically through a script in a feedback loop with AOC compilation and optimization report parsing.

**Shift Register Inference:** *Sliding window* computation [3] is a common pattern (e.g., filters) that can benefit from a SWI kernel design. This pattern includes a loop that accesses a fixed number of contiguous locations in an array shifted by one position per iteration. Such sliding-window memory access patterns can benefit from using *shift registers*. For AOC to infer a shift register implementation, code has to be written in a certain, counter-intuitive from a software development standpoint, way. The resulting implementation is very similar to the method used for enhancing II in SWI kernels: declaration and initialization of the shift register to a zero value, a fully unrolled shifting loop that includes shifting contents across neighboring elements except to the first (or last) that gets its value from the original input array. Last step entails replacing the original input array accesses with shift register accesses. Since sliding-window algorithms have a fixed number of iterations (usually small) the above optimization is coupled with full loop unrolling. What is challenging, and currently limiting its practical implementa-

tion within GLAF-OCL, is automatically identifying sliding-window patterns in applications. We see two examples in more detail in Section IV.

**Kernel Vectorization (SIMD) /Multiple Compute Units (CU)**: *Kernel vectorization* enables work-items (in NDRange kernels) to execute in a SIMD-like fashion. A desirable potential side effect of kernel vectorization is static memory coalescing automatically performed by AOC. *Compute Unit replication* refers to generating multiple copies of a CU for a kernel, at the cost of increased global memory traffic. Generally, between the two, kernel vectorization is more efficient resource-usage-wise, but the trade-offs may not always be straightforward. GLAF-OCL can generate multiple code implementations to be compiled and evaluated by annotating a kernel with the corresponding attributes (*__attribute__*): *num_simd_work_items(N)* for kernel vectorization with vectors of length *N*, and *num_compute_units(N)* for *N* compute units.

**Restrict Clause:** Visual programming via GLAF entirely hides the concept of pointers from users and *aliasing* issues are de facto not applicable. As such, all pointers in auto-generated code are further annotated with the *restrict* keyword in the function header (see kernel header in Figure 3). This eliminates unnecessary assumed memory dependencies and leads to more efficient designs, both in terms of area and performance.

**Constant cache memory:** Declaring kernel pointers for data that are read-only throughout kernel execution as *__constant* enables loading into an on-chip cache optimized for hit performance. In GLAF-OCL we can keep track of read-only grids in a kernel (by conservatively analyzing the read/write locations in the code via the GLAF internal representation). If data can fit in cache (detectable for static grid sizes by inspecting the data type and dimension sizes elements of the grid object) the generated code for the corresponding kernel pointers is annotated as *__constant*.

**Memory Alignment:** Aligned memory allocation of the host-side buffers enables direct memory access (DMA) transfers that can be considerably faster than data transfers between the host CPU and FPGA from/to unaligned memory. GLAF-OCL auto-generated code ensures that all memory allocations follow the board-specific alignment requirements. Specifically, instead of the default *malloc()* call in GLAF, GLAF-OCL generates *alignedMalloc()* calls in the host code and the implementation of this function in the header file (.h) that is effectively a wrapper of *posix_memalign()*.

## IV. Results

### A. Experimental Setup

**Hardware/Software:** For the FPGA implementations we use a *Bittware S5-PCIe-HQ* board (S5PHQ-D8), based around a high-performance Altera Stratix V GS FPGA, with 16 GB of DDR3 SDRAM. The OpenCL kernel codes were compiled using Altera OpenCL SDK v14.2. For the CPU implementations we use a *Intel E5-2697* (Ivybridge) with 12 cores (24 threads), clocked at 2.7GHz, AVX support and 30MB of L3 cache. The CPU (parallel) implementations (in C with OpenMP directives), as well as the host-side code of the OpenCL implementations, were compiled using gcc v.4.8.2 and run on a Debian host (kernel 3.2.46) with 64GB RAM.

TABLE I: Kernel Implementations.

| Implem. | Type | CUs | SIMD | Const. mem. | Kernel Freq. |
|---|---|---|---|---|---|
| **NB0** | NDR | 1 | 1 | N | 268.95 |
| **NB1** | SWI | 1 | 1 | N | 280.58 |
| **NB2** | NDR | 1 | 8 | N | 244.91 |
| **NB3** | NDR | 1 | 16 | N | 223.01 |
| **NB4** | NDR | 2 | 16 | N | 190.73 |
| **NB5** | NDR | 3 | 16 | N | 193.19 |
| **NB6**† | NDR | 1 | 1 | N | 215.33 |
| **SS0** | NDR | 4 | 8 | N | 183.95 |
| **SS1** | NDR | 2 | 16 | N | 184.16 |
| **SS2** | NDR | 1 | 1 | Y | 144.3 |
| **SS3** | NDR | 6 | 16 | Y | 153.04 |
| **SS4**⋆ | NDR | 1 | 8 | N | 186.7 |
| **SS5*** | SWI | 1 | 1 | Y | 118.35 |
| **FF0** | NDR | 4 | 16 | Y | 183.89 |
| **FF1** | SWI | 1 | 1 | Y | 262.61 |
| **FF2** | SWI | 10 | 1 | Y | 195.65 |
| **FF3**‡ | SWI | 1 | 1 | Y | 191.97 |
| **FF4**‡ | SWI | 10 | 1 | Y | 170.12 |
| **FF5*** | SWI | 1 | 1 | Y | 188.46 |

† Resource-driven optimized ‡ Initiation interval reduction
* Shift register inference ⋆ Inner loop unrolling
**NDR:** NDRange **SWI:** Single work-item

**Applications:** We present our experiences and results for a set of example applications that span three different domains (physics, bioinformatics, signal processing) and that exhibit different types of parallelism: **(a) Electrostatic surface potential calculation (NB):** An n-body type of algorithm in which the electrostatic potential on a set of points near the surface of a biomolecule is calculated as a result of their interaction with a set of atoms within the biomolecule, **(b) Gene Sequence search (SS):** An algorithm that scores a given search DNA sequence against all parts of a reference DNA sequence, given a similarity scoring matrix, and returns the start index of the most similar sequence in the reference DNA, **(c) Time-domain FIR filter (FF):** A sliding-window algorithm that implements a set of FIR filters, where each filter's output is the convolution of its coefficients (complex number) and the input vector.

### B. Results and Analysis

Figures 5a-5c show the **execution time** of OpenCL kernel implementations normalized to the corresponding execution time of the OpenMP-parallel CPU implementation. Both the CPU and FPGA implementations are generated by GLAF (in C) and GLAF-OCL (in OpenCL) to ensure a certain level of fairness of comparisons. We also show the FPGA **resource utilization** to obtain insights on the effect of various optimizations on it and trade-offs between resource utilization and performance. The characteristics of alternative implementations for each example application are outlined in Table I.

*1) Electrostatic surface potential calculation (NB):* NB is a highly parallel application and provides insight about the optimizations of kernel vectorization (SIMD), compute unit (CU) replication, NDRange (*NDR*) versus single work-item *SWI*, and the effectiveness or resource-driven optimizations by the Altera Offline Compiler (AOC).

As seen in Figure 5a, increasing SIMD lanes from 1 to 8 (NB0, NB2) and doubling SIMD from 8 to 16 (NB2, NB3) yields a 7.32- and 1.81-fold speed-up, respectively. With each doubling in SIMD vector length resource utilization increases by about 1.35x. Memory access patterns of NB make it amenable for kernel vectorization, as observed in the profiling data: memory accesses are coalesced and result to cache hits in over 99% of the time minimizing memory-related pipeline stalls below 4%. Increased SIMD length only leads to increase of the datapath of a CU (all SIMD lanes share control logic). CU replication case differs, as seen in NB3, NB4, NB5 (SIMD length kept constant): performance increases by a 1.68x when increasing the number of CUs from one to two, while tripling the number of CUs yields a 2.27x increase in performance. Increasing the number of CUs comes at the expense of increased global memory bandwidth across CUs and each doubling of CUs leads to an approximate doubling of resource utilization, too.

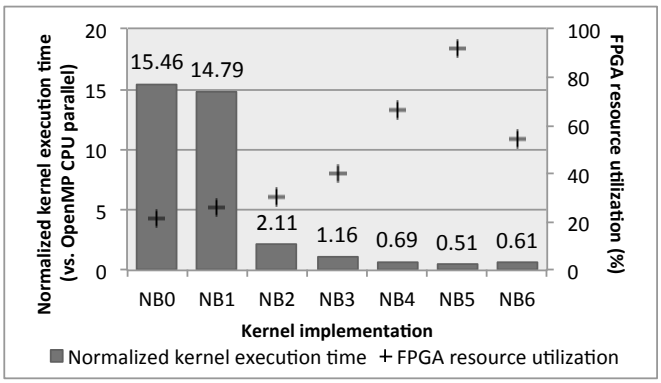
Comparing NDRange and SWI kernel, we observe (NB0, NB1) that performance and resource utilization are almost similar. This is expected, since both NDR and SWI kernels have no kernel vectorization or CU replication and are expressed via pipeline parallelism in FPGA hardware. In fact, the execution time ratio ($t_{NB0}/t_{NB1}$) equals the inverse kernel frequencies ratio ($f_{NB1}/f_{NB0}$). As we see in other example applications (SS, FF), SWI can be beneficial over NDR after applying further optimizations that are not applicable in the NDR paradigm.

Last, NB5 and NB6 provide insight on the effectiveness of resource-driven optimization by AOC. Specifically, in NB6 we use this feature: AOC compiles a kernel with attributes (SIMD length, number of CUs, loop unrolling) based on estimated throughput derived using heuristics. In NB6, AOC identifies loop unrolling (by a factor of 32) to be the most beneficial optimization. Our (brute-force) choice (SIMD 16, CU 3), which provides a 1.2x speed-up over resource-driven optimization, indicates certain limitations of the latter.

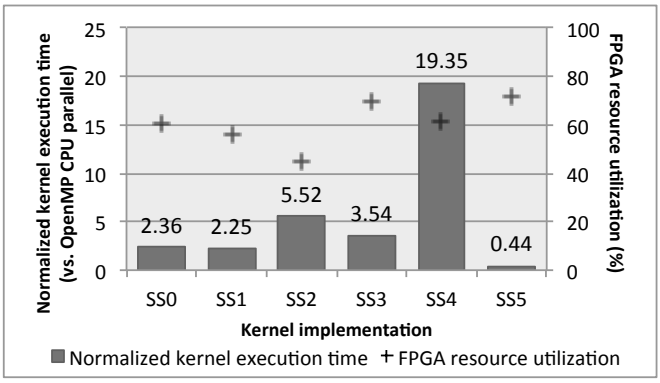
*2) Gene sequence search (SS):* SS serves as an example of the trade-offs in combinations of SIMD length, number of CUs and loop unrolling, use of constant memory, as well as shift register inference in SWI implementations.

With respect to SIMD, CU and loop unrolling, we compare versions SS0, SS1, SS4 (Figure 5b) that yield an overall 32-way parallelism (e.g., SIMD 16 with 2 CUs, or SIMD 8 with 4 CUs). Using wider SIMD (SS1) requires less hardware resources than SS0 and SS4 and provides speed-up over SS0, as expected for similar reasons with NB (e.g., more efficient hardware, coalescing). In SS4, enforcing 4-way loop unrolling together with SIMD only illustrates that careless combination of SIMD and loop unrolling without taking memory access paterns into account can be detrimental for performance (high cache misses and lengthy pipeline stalls).

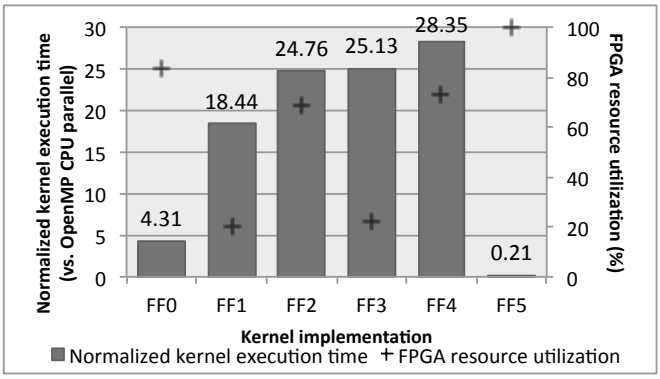
For small search sequences, when applicable (as in our case), use of constant cache memory may present a considerable advantage for an FPGA design, that is better resource utilization that may allow wider SIMD or more CUs to fit




(a) Electrostatic Surface Potential Calculation (NB).




(b) Gene Sequence Search (SS).




(c) TDFIR (FF).

Fig. 5: Results: Execution Time and FPGA Resource Utilization (lower is better)

in an FPGA (e.g., SS0 and SS1 versus SS3, Table I). In SS, due to the parallelization scheme and memory access pattern (i.e., each thread accesses contiguous parts of the reference sequence array shifted by one position) we may have an unfavorable partitioning of the problem to CUs. While the details of scheduling are transparent to the programmer, bandwidth efficiency (i.e., percentage of data acquired from global memory system that the kernel actually uses) is indicative of such an unfavorable partitioning (about 84% in SS3, 95% in SS0 and 99% in SS1). Notice that the bandwidth efficiency increases as the number of CUs decreases.

SS5 illustrates the SWI optimization that pertains to the sliding-window access pattern of the reference sequence array.

This access pattern is ideal for the shift register inference optimization, where OpenCL code follows the guidelines (static size, full unrolling) that allow AOC to generate a shift register structure that is considerably faster than global memory accesses and placed into block RAM. It is worth noting that without the constant memory optimization the FPGA hardware resources would not suffice for full loop unrolling (and hence successful shift register inference). Also, a search sequence longer than 128 bases would lead to insufficient hardware resources, i.e., the search sequence needs to be small and statically determined to allow the shift register optimization.

*3) Time-domain FIR filter (FF):* Some optimizations (Section III-B) found in NB and SS are relevant for FF, too. For example, all shown FF implementations utilize constant cache memory for the filter coefficients. Notably, FF serves as an example where observed results can be counter-intuitive.

Here, the fastest SWI implementation, with shift register inference and loop unrolling (FF5) is 20.5 times faster than the fastest NDR one we compiled. FF reiterates the importance of the above optimizations in sliding-window algorithms. Notice that FF5 barely fits the FPGA for our 128-tap filter example. Without use of constant memory and its lighter resource usage, FF5 would not be possible. Also, without the shift register inference optimization full loop unrolling cannot be applied at all, due to resource restrictions. The above observations highlight the importance of applying FPGA-specific optimizations *in concert*, rather in isolation. GLAF-OCL is useful in this respect, in that it automates code-generation with multiple combinations of optimizations that the user can evaluate.

Conversely, optimizations that may be beneficial for a specific computation pattern can be detrimental for another. While in NB we observe the positive effect of wider SIMD and more CUs, in FF this is not the case (FF1 is 1.34x faster than FF2, with 1/10 of the CUs). Accordingly, higher resource utilization does not necessarily imply better performance. Last, the II reduction optimization in FF3 (reduces II from 8 cycles to 1) yields worse performance than FF1. Despite the reduction in the number of cycles between iterations in this particular case the resulting clock frequency for the design is 1.36 times slower than FF1 (as is the speed-up of FF1 over FF3).

## V. Related Work

High-level synthesis (HLS) approaches, as an alternative to complex RTL-level HDLs, can be broadly divided to two main categories, *text-based* and *model-based/GUI-based*:

**Text-based:** This approach adopts text-based programming in languages that allow behavioral algorithm description. Languages specifically devised for HLS, like *Bluespec* [4], typically require a steep learning curve and cannot take advantage of existing code. Most HLS languages are based on C/C++ [5], [6] (e.g., C-to-Verilog, Impulse C, Catapult C, Mitrion C, Synphony C, Vivado HLS), but the code often needs to be annotated with language-specific constructs. C/C++ itself was designed as sequential language, thereby the majority of such HLS languages lacks *intrinsic* support for describing parallelism. HLS based on languages like CUDA (FCUDA [7]) or OpenCL (Altera OpenCL [8], SOpenCL [9]) addresses this problem. In all above tools HDL code is generated and synthesized at the last step to produce the FPGA binary.

**Model-based/GUI-based:** Tools in this category are based on graphical interfaces. *NI LabVIEW FPGA Module* [10] extends the capabilities of LabVIEW and targets NI FPGAs. *Matlab HDL Coder* [11] allows using Simulink models and Matlab functions to generate synthesizable VHDL and Verilog code for Xilinx and Altera FPGAs. Other graphical model-based design tools include *SystemVue* [12] and *VisualSim* [13].

GLAF-OCL attempts to combine the advantages of both the above approaches. As a framework based on OpenCL (in its code-generation back-end), its base performance and resulting design features are largely contingent on the underlying OpenCL compiler framework. On the other hand, being a GUI-based tool (in its development GUI front-end) it provides an even higher level of abstraction, also obviating any need of hardware details knowledge.

## VI. Conclusions

We presented a new way of programming FPGAs by providing OpenCL-oriented extensions for the GLAF intuitive, visual programming framework. Automating OpenCL code generation and FPGA-specific optimizations via tools like GLAF-OCL has great potential to further democratize FPGA programming by helping to bridge the performance-programmability gap and ensuring functional and performance portability in heterogeneous computing.

## References


[1] K. Krommydas, R. Sasanka, and W.-C. Feng, "GLAF: A Visual Programming and Auto-tuning Framework for Parallel Computing," in *International Conference on Parallel Processing (ICPP)*, Sept. 2015.

[2] *Altera SDK for OpenCL: Best Practices Guide*, Altera, 2015. [Online]. Available: http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf

[3] C. C. Aggarwal, *Data Streams: Models and Algorithms*. Springer Science & Business Media, 2007, vol. 31.

[4] R. Nikhil, "Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications," in *ACM and IEEE 2nd International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, June 2004.

[5] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA Programming for the Masses," *Communincations ACM*, vol. 56, no. 4, pp. 56–63, Apr. 2013.

[6] W. Meeus, K. Van Beeck, T. Goedem, J. Meel, and D. Stroobandt, "An Overview of Today's High-Level Synthesis Tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, 2012.

[7] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs," in *IEEE 7th Symposium on Application Specific Processors (SASP)*, July 2009.

[8] Altera, "Implementing FPGA Design with the OpenCL Standard," https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01173-opencl.pdf.

[9] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos, "Synthesis of Platform Architectures from OpenCL Programs," in *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2011.

[10] N. Instruments, "LabVIEW FPGA Module," http:http://www.ni.com/labview/fpga/.

[11] MathWorks, "Matlab HDL Coder," http://www.mathworks.com/products/hdl-coder/.

[12] Keysight, "SystemVue," http://www.keysight.com/en/pc-1297131/systemvue-electronic-system-level-esl-design-software.

[13] Mirabilis, "VisualSim," http://mirabilisdesign.com/new/visualsim/.