Massively Parallel Programming Models Used as Hardware Description Languages: The OpenCL Case

Muhsen Owaida, Nikolaos Bellas, Christos D. Antonopoulos, Konstantis Daloukas, and Charalambos Antoniadis
Department of Computer and Communication Engineering,
University of Thessaly, Volos, Greece
Email: {mowaida, nbellas, cda, kodalouk, haadonia}@inf.uth.gr

Abstract— **The problem of automatically generating hardware modules from high level application representations has been at the forefront of EDA research during the last few years. In this paper, we introduce a methodology to automatically synthesize hardware accelerators from OpenCL applications. OpenCL is a recent industry supported standard for writing programs that execute on multicore platforms and accelerators such as GPUs. Our methodology maps OpenCL kernels into hardware accelerators, based on architectural templates that explicitly decouple computation from memory communication whenever this is possible. The templates can be tuned to provide a wide repertoire of accelerators that meet user performance requirements and FPGA device characteristics. Furthermore, a set of high- and low-level compiler optimizations is applied to generate optimized accelerators. Our experimental evaluation shows that the generated accelerators are tuned efficiently to match the applications memory access pattern and computational complexity, and to achieve user performance requirements. An important objective of our tool is to expand the FPGA development user base to software engineers, thereby expanding the scope of FPGAs beyond the realm of hardware design.**

Keywords–OpenCL, FPGA, Electronic Design Automation, Reconfigurable Computing, Embedded Systems

I. INTRODUCTION

The level of integration of modern field-programmable gate arrays (FPGAs) has advanced to a point where a complex System on Chip (SoC) can be mapped onto a single device. FPGA manufacturers and IP vendors now offer a plethora of peripherals, processor cores (both hard and soft), and fixed IP solutions. These improvements in capacities, as well as performance and cost, have made FPGAs an attractive solution for many embedded systems. At the same time, FPGAs are increasingly used as accelerators in the context of high performance computing.

Apart from the improvements in hardware technology, tools facilitating the exploitation of FPGAs have also evolved. The current state of the art includes numerous industrial CAD tools [11], [7], [14] that provide automatic hardware generation from high level programming languages such as C/C++.

The hardware / software partitioning on a heterogeneous platform with general purpose or embedded processors and FPGAs is not an easy undertaking. Ideally, systems should be fluid; the computational problem should be expressed using a high-level programming model, letting the compiler and run-time infrastructure to determine which functionality should be assigned to gates and which will be executed on the processors. This decision would consider computational requirements, data transfers and data caching capabilities of the complex memory hierarchies of heterogeneous systems, as well as performance, power and area-limitations. At the same time, in order to make the best use of existing code-base and programmer expertise, developers of the underlying compiler and run-time infrastructures should resist the temptation of introducing new, non-standard programming models or significant extensions to existing ones. Moreover, programmers should be allowed and encouraged to express their problems in an as platform-agnostic way as possible.

In this paper we introduce a hardware-architecture accelerator template and algorithms to generate hardware accelerators from unmodified OpenCL kernels [5]. OpenCL is an industry supported programming standard for both homogeneous and heterogeneous parallel systems. It aims to be platform-independent, yet expressive enough to facilitate the efficient exploitation of parallelism at any granularity, and outline, in a general manner, data movement. An OpenCL kernel expresses parallelism at its finest granularity. The full computation is performed by mapping the kernel on a hierarchical geometry of logical threads. These features facilitate the use of OpenCL as a hardware-description language.

Apart from the computational data path, our tool-chain generates dedicated hardware, called streaming units, to be used for data movement, so that communication is decoupled from computation. The streaming units assemble the input data and present them to the computation kernels as ordered packets. Similarly, results produced by the computation kernels are combined to chunks whenever possible and sent to the main memory. A cache may be instantiated if the memory access pattern analysis reveals reuse opportunities and temporal or spatial locality. Additionally, tunnel buffers are generated to exploit read-after-write reuse across loop iterations.

Much like vector processing, stream programs hide latency, amortize instruction overhead and expose data parallelism by operating on large sets of data. This decoupling becomes possible through code slicing, a technique that automatically extracts the portions of the nested loop code responsible for address generation and data fetching/write back. Code slices are optimized and scheduled separately and are used to configure the streaming unit and the data path, respectively. This allows fully asynchronous operation and overlap across loop and iteration boundaries.

Our infrastructure supports arbitrary loop nests and shapes. Different loops at the same level of a loop nest are implemented as distinct accelerators, which communicate and synchronize through local memory buffers. Similarly, we support barrier synchronization constructs within a computational kernel. The computational data path supports both standard and complex data types and all standard arithmetic operations, including integer and IEEE-compliant single- and double-precision floating point.

We implement the developed algorithms as a backend in the LLVM based compiler infrastructure. We add preliminary passes

before the hardware generation, such as predication, code-slicing and modulo scheduling. The proposed architecture-template exposes tunable parameters that can be manipulated to achieve the performance-cost requirements. The second phase includes functional units (FUs) allocation, tasks scheduling, memory mapping and control circuitry generation.

The major contribution of this paper is a template-based hardware accelerator generation methodology which produces designs with explicitly decoupled memory access and computational units, starting from totally unmodified OpenCL code. We use code slicing techniques to generate and optimize these two units separately and exploit available bandwidth to the memory. The supported kernels may consist of arbitrary loop nests and shapes. They may contain synchronization and any kind of standard arithmetic operations. Finally, the operation of the streaming units and the computational data paths is fully asynchronous, even across the boundaries of different loops and loop nests.

The rest of the paper is organized as follows: Section II introduces the OpenCL programming model and describes the compiler front-end that transforms OpenCL to equivalent, appropriately coarsened C code. Section 0 discusses in detail the architectural template used as a base for the auto-generated hardware.

```
lud_perimeter( __global float *m, int m_d, int offset){
__local float dia[16][16];
__local float peri_row[16][16];
 __local float peri_col[16][16];
int loc_x = get_local_id(0);
int b_x    = get_group_id(0);

   if (loc_x < 16) {                                          B0_0
     idx      =  loc_x ;   i0  = 0; i1 = 8;
     arroff  = arroff0 = offset * m_d + offset;
     arroff1 = arroff + (b_x + 1)<<4 + idx;
   }
   else{                                                      B0_1
     // similar settings
   }
  arr_offset = arroff0;
  for (i = i0; i < i1; i++) {                                 L0_2
    dia[i][idx] = m[ array_offset + idx];
    array_offset += m_d;
  }
  array_offset = arroff1;
  for (i = 0; i < 16; i++) {
    if ( loc_x < 16)  peri_row[i][idx] = m[ array_offset+idx ];  L0_3
    else  peri_col[i][idx] = m[ array_offset+idx ];
   array_offset += m_d;
  }

barrier(CLK_LOCAL_MEM_FENCE);

  for (i = 1; i < 16; i++) {
    for (j = 0; j < i; j++) {
     if ( loc_x < 16)                                          L1_0
        peri_row[i][idx] -= dia[i][j] * peri_row[j][idx];
     else
         peri_col[idx][i] -= dia[j][i] * peri_col[idx][j] ;
    }
    if( loc_x >= 16)
          peri_col[idx][i] /= dia[i][i];
  }

barrier(CLK_LOCAL_MEM_FENCE);

  if ( loc_x < 16) {                                          B2_0
     array_offset  = (offset + 1) * m_d + offset;
     for (i = 1; i < 16; i++) {                               L2_0
        m[array_offset + (b_x + 1) * 16 + idx] =  peri_row[i][idx];
        array_offset  +=m_d;
     }
   }
  else {
      array_offset  = (offset + (b_x + 1) <<4) *  m_d + offset;   B2_1
     for (i = 0; i < 16; i++) {
        m[ array_offset + idx] = peri_col[i][idx];            L2_1
        array_offset  += m_d;
     }
   }
```

Figure 1. OpenCL kernel for LU Decomposition with marked loops (Li_j) and basic blocks out of loops (Bi_j). In this kernel, a work-item (or thread) performs LU Decomposition for a 32x32 sub-matrix. Some parts of the code have been omitted for brevity.

In Section IV we introduce the hardware generation compiler backend and several optimizations applied at this level. Section V evaluates our proposed methodology and the toolchain implementation. Finally, Sections VI and VII outline related work and conclude the paper, respectively.

II. OpenCL Front End

A. *OpenCL Programming Model*

OpenCL [5] provides a parallel programming framework for a variety of devices, ranging from conventional Chip Multiprocessors (CMPs) to combinations of heterogeneous cores such as CMPs and GPUs. It is based on a platform model that comprises a *host* processor and a number of *compute devices*. Each device consists of a number of *compute units*, which is subsequently divided into a number of *processing elements*. An OpenCL application is organized as a *host program* and a number of *kernel functions*. The host part executes on the host processor and submits commands that refer to either the execution of a kernel function, or the manipulation of memory objects. Kernel functions contain the computational part of an application and are executed on the compute devices.

The work corresponding to a single invocation of a kernel is called a *work-item*. Multiple work-items are organized in a *work-group*. OpenCL allows for geometrical partitioning of the grid of computations to an N-dimensional space of work-groups, with each work-group being subsequently partitioned to an N-dimensional space of work-items, where $1 \leq N \leq 3$. Once a command that refers to the execution of a kernel function is submitted, the host part of the application defines an abstract index space, with a maximum of six dimensions. A work-item is identified by a tuple of IDs, defining its position within the work group, as well as the position of the workgroup within the computation grid. Based on these IDs, a work-item is able to access different data (SIMD style) or follow a different path of execution.

The only synchronization that is provided by OpenCL is barrier-type synchronization among the work-items inside the same work-group. Every work-item inside a work-group must execute the barrier instruction before any work-item is allowed to continue execution beyond the barrier command. On the other hand, there is no synchronization mechanism among work-groups, which means that different work-groups can always be executed in parallel.

We use an OpenCL kernel which implements *LU Decomposition* as a running example to explain the sequence of steps to generate the hardware accelerator (Figure 1). This kernel is part of the *Rodinia* benchmark suite [3]. The *get_group_id(0)* run-time function call returns the x-coordinate of the work-group in which the work-item calling the function belongs to in the computational grid. *get_global_id(0)* returns the unique global x-coordinate of the work-item, whereas *get_local_id(0)* returns the x-coordinate of the work-item within the work-group. LU Decomposition kernel consists of three code parts, separated by barrier instructions. All work-items that execute the first part of the code, pre-fetch a segment of the input array *m* to three local buffers and have to rendevouz to the first barrier before they proceed. The second part of the code performs the main LU Decomposition operation, and, likewise, forces all work-items to synchronize to the second barrier, before proceeding to the final write-back to array *m*.

B. *Compiler transformations for OpenCL front end*

In order to enable efficient mapping of OpenCL kernel functions to the underlying platform, while at the same time taking into account hardware constraints, we apply a series of source-to-source transformations that collectively aim at coarsening the granularity of a kernel function. After this process, a kernel function represents the

amount of computations that must be executed by each work-group in the abstract index space of the application. The main step in this series of transformations is *logical thread serialization*. Work-items inside a work-group can be executed in any sequence, provided that no synchronization operation is present inside a kernel function. Based on this observation, we serialize the execution of work-items by enclosing the instructions in the body of a kernel function into a *triple nested loop*, given that the maximum number of dimensions in the abstract index space within a workgroup is three. Each loop nest enumerates the work-items in the corresponding dimension, thus serializing their execution.

The aforementioned transformation can lead to invalid execution of a kernel function if its body contains a synchronization operation. In presence of a barrier instruction, every work-item must execute that instruction before any work-item is allowed to continue its execution. However, in the modified kernel function, every work-item finishes its execution before the next work-item is able to start. In order to ensure correct execution of the coarsened kernel function, the compiler applies two additional transformations, namely *loop fission* and *variable privatization* that facilitate logical thread serialization.

Loop fission is applied in order to enforce the execution ordering that is required by a synchronization instruction. A triple-nested loop enforces synchronization among work-items before its first and after its last iteration. Based on this observation, we partition the instructions of a kernel function into blocks such that no barrier instruction is present inside a block. Afterwards, we enclose each block into a triple-nested loop. We follow the same approach for kernel functions with multiple exit points, i.e. when break, continue or return statements are present. We treat each of the aforementioned instructions as an additional synchronization point and apply loop fission around it. Figure 2a depicts the block structure of the modified kernel function for our running example. The kernel code separated by barrier instructions is enclosed in triple nested loops (T_i), and the whole kernel code is enclosed into a doubly nested loop which spans

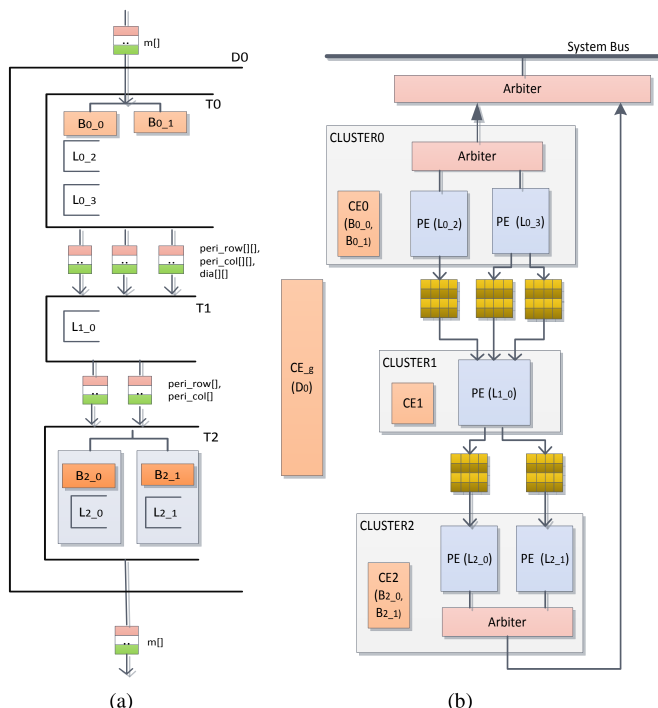


Figure 2. (a) Program structure of LU Decomposition kernel after coarsening the granularity to the equivalent of a work-group. (b) The block diagram of the automatically generated hardware accelerator for LU decomposition.

the dimensions of the 2D array.

The last step in this series of transformations is variable privatization [1]. Loop fission presents a complication for variables that are defined in one triple-nested loop construct and used in another. A work-item that defines the value of a variable in the first loop cannot use it in a subsequent loop, as its contents will be polluted by the execution of subsequent work-items, thus violating semantics. We conduct a live-variable analysis in order to identify such variables and we subsequently apply the variable privatization technique for each variable. After this transformation, each work-item is provided with a private copy of such variables. Further details on the OpenCL compiler transformations are presented in [15].

III. ARCHITECTURAL TEMPLATE

The hardware generation tool-chain transforms the C code to synthesizable HDL, based on an architectural template that can be instantiated to match the performance requirements of the application and the available FPGA resources. Figure 3 outlines the architectural template of a PE which consists of the data path and the streaming unit, and is detailed in the following sections. Figure 2b shows the block diagram of the complete hardware accelerator that may consist of multiple PE modules. Consumer-producer communication of Figure 2a translates into FPGA BRAMs, which allow pipelined operation. Inner loops Li,j are mapped into Processing Elements of Figure 3 and outer loop computation and control are mapped into Control Elements CE. Arbitration mechanisms are used to regulate access to the shared interconnect network (a bus in our case). There is a very close correspondence among the OpenCL code in Figure 1, the program structure in Figure 2a and the generated hardware in Figure 2b. The main code structures have been annotated using the same tags in all 3 figures.

A. Data Path

The data path implements the computations of the innermost loops (loops Li_j in Figure 1). It consists of a network of functional units (FUs) that produce and consume data elements using explicit input and output FIFO channels to the streaming units. Each FU is preceded by multiplexers, which – at each time-slot – direct data elements into the correct input-port. The multiplexers are driven by a periodic-count of the initiation interval (II). The control logic is distributed and spatially near the corresponding functional units, multiplexers, and buffers. The data path also includes static data-registers that hold loop invariant data generated by outer loops. Tunnels are storage elements used to bypass the streaming unit and channel data-tokens stored (*pushed*) in earlier iterations to be used by loads (*pop*s) in later iterations. Tunnels are generated wherever a load instruction has a read-after-write dependency with another store instruction with constant cross-iteration distance larger than or equal to one. Dependences with distance equal to zero are optimized away during the optimization passes. For example, tunnels would be generated to implement the cross-iteration distance of 1 in the following loop:

for (i = 0; i < N; i++)
a[i] = a[i-1] + b[i];

The reconfigurable parameters of the data path are the type and bitwidth of functional units (ALUs for arithmetic and logical instructions, multipliers, shifters, etc.), the custom operation performed within a generic functional unit (e.g. only addition or subtraction for an ALU), the number and size of registers in the queues between functional units, and the bandwidth to and from the streaming unit. Both integer and IEEE compliant single/double precision floating point operations are supported by the tool flow. Multiple versions of each floating point operation are implemented in the context of an add-on library. Each implementation is tagged with its precision, its latency, as well as the number of its pipeline stages.

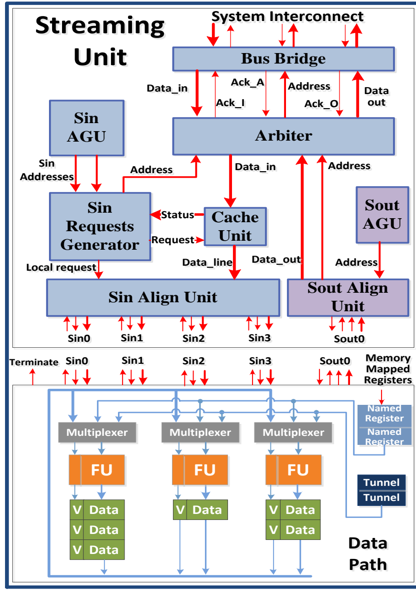


Figure 3. Architectural template of a Processing Element (PE) module.

At compile time, the system selects and integrates the appropriate implementation according to precision requirements and the target initiation interval. Section IV describes algorithmic aspects of the automatic generation of the hardware accelerator.

B. Streaming Unit

The streaming unit handles all issues regarding data transfers between the main memory and the data path. These include address calculation, data alignment, data ordering, and bus arbitration and interfacing. The streaming unit consists of one or more input and output stream modules. It is generated to match the memory access pattern of the specific application, the characteristics of the interconnect to main memory, and the bandwidth requirements of the data path.

An Address Generation Unit (AGU) aggressively generates addresses for data prefetching (and write back), and feeds them to the Address Request Module. The tool flow guides the generation of the AGUs by first identifying the code slice responsible for data I/O, and then performing modulo scheduling on that code. The output of the code slice – and, therefore, the output of the generated AGU hardware – is an address sequence for all elements of the input stream. The architecture of the AGU is very similar to that of the data path, thus the same methodology is used to generate both.

The Requests Generator module coalesces requests generated by Sin-AGU (the input data AGU) to the word width of the underlying memory interconnect (a PLB bus for Xilinx FPGAs), or to burst size if bursting is enabled. Moreover, it competes for bus accesses with the other stream units. The Requests Generator Module aims to eliminate redundant transactions on the memory interconnect. Before issuing a transaction request to the Arbiter, it checks if the addresses alias with previously requested ones, or if the requested data is available in the cache unit.

The cache unit exploits temporal and spatial locality and reduces latency of memory accesses by saving recently loaded data for future reuse. The cache unit is implemented using dual-ported Block RAMs, so that accesses from the Arbiter and the Input Streams Alignment Unit Sin-Align can be served simultaneously. A cache line is equal to the *bus-width*. The cache unit is not instantiated if the compile-time analysis dictates that the input memory access pattern has limited reuse. The input stream Alignment Unit retrieves data from the cache unit, or the *data_in* incoming data in case there is no cache, and presents them in-order to the data path. The output stream Alignment Unit aligns the output data tokens coming from the data path in a FIFO of data-lines of *bus-width* bytes. As soon as the FIFO is full or the incoming data token is out of lines, the Alignment Unit issues the write request to the Arbiter.

Finally, the Arbiter module manages the issue of Read/Write transaction requests from the input/output streams towards the memory interconnect. The priority and ratio of serving read vs. write requests is determined at compile-time, driven by the memory access pattern analysis of the application. However an interrupt mechanism is also present to intervene whenever input or output queues are full, thus stalling the data path.

C. Control Elements

Control Elements are used to control and execute code of outer loops in a multilevel loop nest. For example, blocks CEi_j in Figure 2b show the Control Elements (CEi) for the LU Decomposition benchmark. Control Elements have a simpler, less optimized architecture, since outer loop code does not execute as frequently as inner loop code.

A critical architectural optimization is interleaving the execution of multiple outer loop iterations. Figure 4a shows the conventional execution model, according to which the next iteration of an outer loop can be initiated only after the last iteration of the inner loop. The conventional model creates execution bubbles at the prologue and epilogue of each outer loop iteration (T0 and T2, respectively), during which computing resources remain idle, thus causing unnecessary execution delays.

To ameliorate this inefficiency, we decouple initiations of innermost loops from outer loops and treat them as separated entities (Figure 4b). We should note that, in most cases, outer loops correspond to the abstract index space of a work group (triple nested loops Ti of Figure 2a). Since all work-items of a work-group in OpenCL are allowed to be executed concurrently, there is no data dependence among iterations of such outer loops. All local variables are implemented as FIFOs, thus the correct data are presented to each loop iteration. Updates to local variables from future iterations do not pollute the data presented to previous, concurrently executing iterations. Therefore, the execution of multiple loops and multiple iterations within each outer loop can be overlapped without violating data dependences.

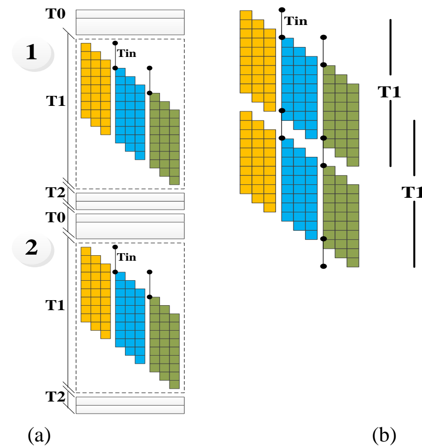


Figure 4. Nested loop execution model (a) when there is no overlap between successive outer loop iterations (synchronous model) and (b) when successive outer loops overlap (asynchronous model). We show inner loops that correspond to Input Stream Units, Computational Kernels, and Output Stream Units.

D. Pipeline Memory

Pipelined communication between outer loops using arrays in the generated C code (Figure 2a) is translated into pipeline memory in the hardware accelerator (Figure 2b). Pipeline memory is implemented using FPGA BRAMs, which are typically large enough to accommodate data channels in which the producer loop generates data using a different memory access pattern than the consumer loop. Processing Elements that exchange data with pipeline memory do not need a cache memory (e.g. PE for L_{1_0} in Figure 2b).

Pipeline memory also enables exploitation of pipeline-level parallelism available in most OpenCL kernels. In fact, the LU Decomposition OpenCL kernel uses the first and third pipeline stage to pre-fetch and write-back data from/to the main memory, respectively. This is an optimization step typically used by OpenCL and CUDA software developers to place data in a local memory before operating on them.

IV. Compiler Optimizations and Hardware Generation

After the front-end OpenCL to C transformation (Section II), the hardware generation flow generates the synthesizable HDL of the accelerator. We extend the functionality of the LLVM compiler infrastructure [6] to implement *predication*, *code slicing* and *modulo scheduling*. Then, the compiler backend generates the final hardware modules of the application-specific architecture, targeting the templates of Figure 2b and Figure 3.

A. Compiler Optimizations

As a first pass, we apply *predication* to the body of each innermost loop. Predication converts control dependences to data dependences in the loop, transforming its body to a single basic block. This is a prerequisite in order to apply modulo scheduling in subsequent steps. The predicated code encapsulates both data transfer operations and computations of the inner loop body. A *code slicing* step partitions the code to three distinct kernels:

Input Streaming Kernel: This kernel consists of all the *load* instructions and any instruction participating to the calculation of load addresses. The kernel drives the hardware generation of the Input Stream AGU (Sin-AGU module).

Output Streaming Kernel: Similar to the Input Streaming Kernel, however for *store* instructions. It drives the hardware generation of the Output Stream AGU (Sout-AGU module).

Computational Kernel: This is the core of the accelerator, and comprises all instructions that receive input data from the Input Stream Units and produce output data to the Output Stream Units. Since data are streamed in and out of the data path in-order, a *pop/push* instruction consumes/produces the next element without the need to specify a memory address. The computational kernel drives the hardware generation of the data path module.

The aim of code-slicing is to disassociate computation from data management instructions and facilitate their overlap. Table I depicts the pseudocode of code slicing for Input Streaming kernel and Computational kernel. All loads and store instructions of the Computational kernel and all their predecessors, i.e. instructions used to compute memory addresses are allocated to the Input and Output Streaming Units, respectively. In the Computational unit, these instructions are substituted by *pop* and *push* instructions used to stream data from the Input Streaming Unit to the Data Path and from the Data Path to the Output Streaming Unit, respectively.

This asynchronous data flow model allows data to be fetched ahead of computation, provided there are no inter-kernel data

Table I. Code slicing algorithm. Output streaming kernel generation is similar to the input streaming kernel, with stores being the instructions of interest.

Input: Kernel code in LLVM assembly code
Output: Three distinct modified kernels in LLVM assembly code

```
// Input Streaming Kernel generation
get_sin_kernel(inner_loop, InstructionList *sin_list){
  sin_list = NULL;
  foreach (instruction It in inner_loop)
      if (It is a load instruction)
              add(It, sin_list);

  It = select any instruction from sin_list;
  while (It!= NULL) {
      foreach (predecessor(It) != NULL)
             add(predecessor(It), sin_list);
      It = select any (predecessor(It)!= NULL);
  }

  It = select any instruction from sin_list;
  while (It!= NULL) {
      pred = predicate(It);
      if (pred != NULL){
          foreach (predecessor(pred) != NULL)
             if (sin_list(predecessor(pred)) == NULL){
                pred = NULL; break; }
          if(pred != NULL)
            add(pred, sin_list);
  }}}
//Computational Kernel generation
get_comp_kernel(inner_loop,  InstructionList *sin_list ,
                             InstructionList *sout_list
                             InstructionList *comp_list){
  comp_list = NULL;
  foreach (instruction It in inner_loop)
      if ((It not in sin_list) && (It not in sout_list))
          add(It, comp_list);
          if (predicate(It)!=NULL)
              add( predicate(It), comp_list);
}
```

dependencies besides pipeline dependencies. Most streaming and data-parallel applications, which are the target of our methodology, follow this pipelined model. However, some applications include data-dependent memory accesses, as shown in the following code.

for (i = 0; i <N; i++)
c[i] = a[ptr[i]+1] +ptr[i]+1;

This results in a dependency of the Input Stream AGU from the Computational kernel. To deal with such problems, we duplicate the parts of the code used to compute data-dependent addresses as necessary, when we perform code slicing. Referring to the example code above, both the Input Stream AGU and the Computational kernel perform the addition (*ptr[i]+1*) on the incoming data stream *ptr[i]*.

Finally, we use *Swing Modulo Scheduling (SMS)* [9] to generate a schedule for each of the three kernels. The scheduler identifies an iterative pattern of instructions and their assignment to functional units (FUs), so that each iteration can be initiated before the previous ones terminate. SMS creates software pipelines under the criterion of minimizing the Initiation Interval (II). The latter is the main factor affecting computational throughput. At the same time, SMS reduces the lifetime of intermediate variables, thus alleviating register pressure. Long variable lives result to larger ALU queues and may lead to unnecessarily large data paths.

The inputs to the SMS scheduler are the instructions corresponding to each kernel, as well as an XML-based hardware model description of the target FPGA, denoting FPGA device characteristics. The main parameters affecting the instantiation of each FU are the supported instructions, the total bitwidth, its latency and the number of pipeline stages.

B. Hardware Generation

Hardware generation is guided by the modulo-scheduled kernels of the innermost loops and by inter-loop dependence analysis. A synthesizable HDL module, similar to the module of Figure 2b is generated. The remainder of this section discusses hardware generation a) for a Processing Element (PE) and b) for a larger accelerator that may consist of multiple PEs.

PE Hardware Generation. Given the modulo-scheduled innermost loop kernels, the compiler backend generates modular Verilog for the steady state body of the Computational kernel and the Address Generation Units (AGU) kernels according to the template of Figure 3. AGUs are also modulo-scheduled kernels, translated to hardware in a similar way to data path (computational kernel) generation, however they are typically simpler and have no tunnels. The modules of the streaming unit communicate with each other through FIFO channels. Each module absorbs tokens from its input FIFOs and writes tokens to its output FIFOs, with a rate depending on the amount of parallel logical paths provided to process its input tokens, and the memory access pattern. When cache creation is bypassed in streaming applications, the cache input channels are directly connected to the "Sin-Align" module.

Note that no Verilog code is produced for the prologue and epilogue of the modulo-schedule [9]. The generated hardware utilizes a *valid bits* mechanism to facilitate the correct execution of the prologue and epilogue. Each data token is tagged with a *valid bit*. An operation produces valid output data only if both input data are valid. A *pop* operation produces data with valid bits when data are available, and a *push* operation accepts data only when they are valid. Since the only source of valid data are pop operations, the rest of the data path produces valid data at the correct loop iteration of the modulo-schedule, thus implicitly implementing the prologue and epilogue of the schedule.

Full Accelerator Hardware Generation. Multiple PEs can be instantiated to execute in parallel (to exploit task-level parallelism), or interconnected as producer-consumer (to exploit pipeline parallelism). This capability also allows our toolchain to support hardware generation for arbitrary loop nests and shapes, forming arbitrary data- and control-flow DAGs. The compiler backend detects data and control dependencies between the basic blocks allocated to each PE. It performs memory access analysis to determine the dependency edges between individual load/store instructions in each basic block and guides the interconnection of PEs based on this analysis.

For example, loops L_{0_2} and L_{0_3} of Figure 1 have no data dependence and do not modify the same memory locations, because L_{0_2} generates local buffer *dia*, whereas L_{0_3} generates local buffers *peri_row* and *peri_col*. Therefore, PE(L_{0_2}) and PE(L_{0_3}) can execute in parallel as shown in Figure 2b. On the other hand, loop L_{1_0} has to wait for the termination of both loops L_{0_2} and L_{0_3} since it has a triple data dependency on their outputs. The synchronization between PE clusters is maintained through a set of finish- and ready-flag signals (asynchronous mode only). This facilitates independent execution rates for different PEs, and alleviates the need for lock step execution between pipeline stages.

V. Experimental Evaluation

A. Methodology

We tested the proposed methodology and architectural template on the six OpenCL and C benchmarks outlined in Table II. The OpenCL benchmarks come from the NVIDIA OpenCL SDK (2D-DCT, MatMul), from the Rodinia [3] benchmark suite (LUD-P) or were developed internally (CMC, Jacobi). All benchmarks were automatically converted to structured C before hardware generation as explained in Section II.B. Finally, LMC was extracted from the AVS video decoder C reference code [13]. All but the two video benchmarks contain FP operations. We used the LLVM compiler infrastructure to implement the hardware generation passes. The tool generates synthesizable Verilog, as well as a testbench for functional simulation.

Table II. Applications used for experimental evaluation. The table summarizes the working set of each app., whether it uses caches or local buffers (LB) and whether it performs integer (Int) or floating point (FP) arithmetic.

App.	Description	Work. Set	Caching	FP/Int
LMC	Luma Motion Compensation (Video)	16×16	Cache	Int
CMC	Chroma Motion Compensation	16×16	Cache	Int
Jacobi	Jacobian	1024×1024	LB	FP
2D-DCT	2-dimensional DCT	640×480	LB	FP
MatMul.	Matrix Multiplication	128×128	LB	FP
LUD-P	LU Decomposition-Perimeter	128×128	LB	FP

To evaluate the efficiency of the methodology and the potential of the proposed architectural template, we used three different hardware configurations (C_A, C_B and C_C) to guide the module scheduling of the Computational and I/O streaming kernels. These configurations represent three levels of resource availability; C_A is an extreme configuration, which allocates just a single FU of each required type (e.g. one adder, one multiplier, etc.) and one word I/O bandwidth. On the contrary, C_C allocates as many FUs as required to achieve the minimum possible II for each loop. Barring any cyclic dependences, this corresponds to II=1. The C_B configuration is selected differently for each application to achieve the average II between the two extremes. For applications with little computation in each loop (LUD-P and MatMul) the C_B configuration proved similar to C_C. We executed the three configurations for the two execution modes, synchronous and asynchronous, as described in Section III.C. Moreover, we created both synchronous and asynchronous configurations of each benchmark. Asynchronous configurations allow the overlap of successive loop iterations, whereas synchronous ones do not. It should be noted, however, that the data path and streaming units are always, even in the synchronous configurations, separate entities. Streaming units prefetch data, implementing the load / store code slices and executing them at a certain distance from the corresponding iterations of the computational slice. For the evaluation of our design we used Xilinx Virtex-6 LX760 FPGA and Xilinx ISE 12.4 toolset for synthesis, placement and routing.

B. Results

Table III summarizes the area results after the synthesis performed for the six benchmarks of Table II. The general trend is that area requirements increase from configuration A to configuration C when the loop body encompasses enough computations to exploit the additional resources. Asynchronous mode configurations tend to consume more slices than synchronous ones. The additional hardware implements the synchronization FIFOs of the PE modules and synchronization flags for Local Buffers. The results show that this hardware overhead is nearly the same in all configurations (C_A, C_B, and C_C). Dual-Port Block RAMs are used for both local buffers and caches. LMC and CMC are the only benchmarks that utilize their Block RAMs as cache, while the rest of the benchmarks use their Block RAMs to implement local buffers for local arrays. In LUD-P, each of the local arrays *dia, peri-row*, and *peri-col* (Figure 2a) is allocated a Block RAM of 36Kbit. In all applications, the Block RAMs are configured as 512 lines long, each being 64-bits wide. The caches and Local Buffers work in simple-dual-port mode (one port allocated for write-only and the second port allocated for read-only) to allow pipelining the writes and reads.

Table III. Area results for the six benchmarks as synthesized, placed and routed on Xilinx Virtex-6 LX760 device. The Virtex-6 LX760 device includes 118560 slices, 720 RAMB36 Block-RAMs, and 864 DSP48 modules.

	Jacobi					
	Asynch			Synch		
Config.	C_A	C_B	C_C	C_A	C_B	C_C
Slices	2073	2250	2486	2077	2139	2382
RAMB36	1	1	1	1	1	1
DSP48	2	4	6	2	4	6

	LUD					
	Asynch			Synch		
Config.	C_A	C_B	C_C	C_A	C_B	C_C
Slices	4643	4405	4405	3688	4118	4118
RAMB36	3	3	3	3	3	3
DSP48	14	16	16	14	16	16

	DCT					
	Asynch			Synch		
Config.	C_A	C_B	C_C	C_A	C_B	C_C
Slices	4928	4823	7393	3894	3334	6379
RAMB36	1	1	1	1	1	1
DSP48	13	14	41	13	14	41

	CMC					
	Asynch			Synch		
Config.	C_A	C_B	C_C	C_A	C_B	C_C
Slices	1396	1511	1783	1229	1282	1637
RAMB36	2	2	2	2	2	2
DSP48	5	6	8	5	6	8

	LMC					
	Asynch			Synch		
Config.	C_A	C_B	C_C	C_A	C_B	C_C
Slices	4440	5244	5314	4231	5123	5322
RAMB36	2	2	2	2	2	2
DSP48	4	8	14	4	8	14

	Mat.Mul.					
	Asynch			Synch		
Config.	C_A	C_B	C_C	C_A	C_B	C_C
Slices	2318	2297	2297	2190	2198	2198
RAMB36	3	3	3	3	3	3
DSP48	8	8	8	8	8	8

Figure 5 depicts the execution time (in ms) and clock rate for each benchmark under different configurations. As expected, the asynchronous mode implementations in all benchmarks achieve higher computational rate and reduced execution time compared to synchronous mode ones. Synchronous operation (without data prefetching) frequently throttles the throughput of PE modules. Asynchronous operation tends to become performance critical when II is small. This is typically the case in the C_C configuration. Faster data paths and AGUs make better use of the control element (CE) module executing the outer loops and preparing data used by the PE modules in subsequent operations.

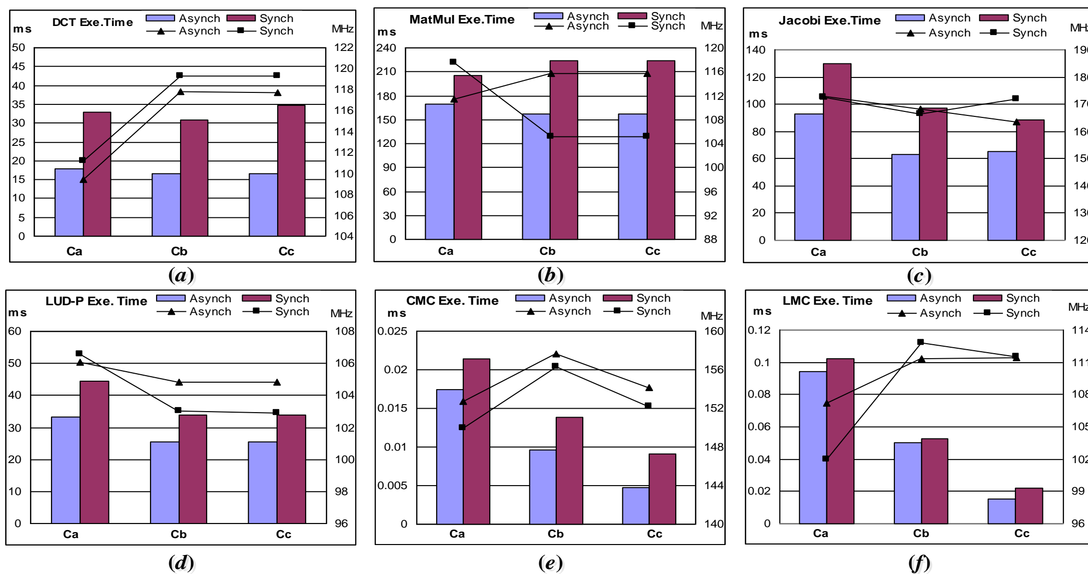
Our infrastructure exploits asynchrony, alleviating at the same time the programmer burden of managing it. Compiler-time analysis proved enough to identify dependences determine the necessary synchronization flags and signals and produce an architecture that executes independent parts of the algorithm in parallel and pipelines dependent blocks to overlap the execution of subsequent iterations. The performance of asynchronous operation may be limited by the existence of data dependences between loops at different level of the loop nest, i.e. when computations in the outer-loops (executed by CE modules) are dependent on results produced from the innermost loops (executed by PE modules). This is the case in LUD-P, where an outer loop computation waits data to be written to a local buffer, performs multiplication and division operations and only then initiates the next iteration. Even in this case, the experimental results indicate that asynchronous execution outperforms synchronous one.

VI. Related Work

There is a large body of literature that deals with conversion of an application written in a high level language to hardware. The PICO-NPA system translates C functions written as perfectly nested loops into a systolic array of accelerators [11]. An accelerator data path consists of a network of FUs, static-registers that hold constant values, and a set of inputs/outputs FIFOs. The list of FUs is allocated initially according to cost-functions considering the instructions types within the loop. A modulo scheduler schedules the loop instructions on the allocated FUs. Trident [12] targets hardware which consists of a list of separate blocks, each with its own state-machine and data path. A global controller is built to transfer control between the blocks.

The LegUp synthesis tool generates a hybrid architecture comprising a MIPS processor and hardware accelerators to speed up performance critical C code [2]. The hardware accelerator generation utilizes conventional HLS techniques for resources allocation, scheduling, and binding. A similar tool from Altera, C2H, selects C functions to be mapped into hardware [7]. OpenRCL platform utilizes OpenCL to schedule fine-grain parallel threads to a large number of MIPS-like cores [8]. OpenRCL does not generate customized hardware accelerators, although each MIPS core can be configured to match application characteristics.

Jääskeläinen et al. introduce a compilation infrastructure based on LLVM to generate transport-triggered architectures from OpenCL codes in an approach seemingly similar to our work [4]. The



(*a*) (*b*) (*c*) (*d*) (*e*) (*f*)

Figure 5. Execution time and Clock Frequency achieved for the six benchmarks.

processors generated with their design flow are statically scheduled VLIW-style architectures with up to hundreds of programmer visible general-purpose registers. Parallelism at the granularity of work-items is exploited in order to overlap memory access latency with computations. They also introduce and use OpenCL extensions in order to code performance-critical parts of the kernels. Our approach is inherently different. We do not favor OpenCL extensions, but perform extensive compile-time analysis instead, and granularity coarsening in order to avoid putting additional burden to the programmers. Our architectural template disassociates memory transfers from computation, thus effectively overlapping them, without necessitating support (and the associated overheads) for in-flight operations from multiple work-items, in arbitrary points of their execution, competing for the same FUs.

Our work is closer to FCUDA, a CAD tool that converts CUDA kernels to synthesizable hardware [10]. A CUDA kernel implicitly describes multiple CUDA threads that are organized in groups called thread-blocks. The inner loop body comprises of function calls to load data, perform computations and write data back. The generated C code is annotated with AUTOPILOT directives indicating parallel code regions as directives for the compiler.

The AutoPilot Compiler [14] generates RTL descriptions for each function in a C program. Each function is translated into an FPGA core. AutoPilot provides code directives to indicate parallel-code regions, and further unroll inner-loops to run concurrently when no-across iterations dependencies are detected. AutoPilot allocates all arrays onto local BRAMs.

In LAURA [16] tool a Kahn Process Network (KPN) specification of an application is converted into a network of concurrent accelerators that communicate through FIFO buffers. LAURA flow uses a library of predefined IP Cores to generate a synthesizable VHLD description of the final architecture. An accelerator is an one-to-one mapping of Kahn Virtual Process and consists of an Execute Unit, Read Unit, Write Unit and Controller. Chip Generator approach [17] proposes a tiled architecture comprised of a network of Quads, where each Quad is comprised of four dual-processor tiles with configurable memory blocks and programmable controller.

Our work targets massively parallel programs written in OpenCL and is based on the premise that the target platform is a pre-defined template that can be configured according to the needs of the application and the user requirements. Moreover, in our work, there is a decoupling of data communication and computation explicitly enforced by the code slicing technique used in the tool flow.

VII. CONCLUSION

In this paper we described a methodology to generate hardware accelerators for complex, unmodified OpenCL kernels and C functions. Our template based design methodology and automatic hardware-generation infrastructure allows the hardware implementation of arbitrary, imperfect loop nests and data- and control-flow DAGs. The architectural template allows the disassociation of computational operations and data-transfers, effectively facilitating the overlap of computation and communication. Moreover, it allows concurrent execution of multiple loop iterations and exploits task- and pipeline parallelism. All the aforementioned capabilities are based on compiler analysis of memory access patterns, control- and data-dependencies and require no programmer intervention. Equally importantly, the hardware-generator can be tuned to match the available FPGA resources and respect target performance requirement.

The experimental evaluation proved the potential of our infrastructure to generate efficient hardware. Moreover, it quantified the tradeoffs of different hardware configurations, as well as of optimizations like the asynchronous execution of loop iterations.

REFERENCES

[1] Randy Allen and Ken Kennedy. "Optimizing Compilers for Modern Architectures: A Dependence-Based Approach". Morgan Kaufmann, 2002

[2] Andrew Canis *et al*. "LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems". In Proc. of the IEEE International Symposium on Field Programmable Gate Arrays (FPGA), pp. 33-36, February 2011, Monterey, CA, U.S.A.

[3] Shuai Che, *et al*. "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads". In Proc. of the IEEE International Symposium on Workload Characterization (IISWC), pp. 44-54, October 2009, Austin, TX, U.S.A..

[4] Ekka Jääskeläinen, Carlos S. de La Lama, Pablo Huerta, Jarmo Takala. "OpenCL-based Design Methodology for Application-Specific Processors". In Proc. of SAMOS X: Embedded Computer Systems: Architectures, MOdeling, and Simulation, pp. 223-230, July 2010, Samos, Greece.

[5] Khronos OpenCL Working Group. Editor: A. Munshi, "The OpenCL Specification", Version: 1.1 Document Revision: June 11, 2010.

[6] Lattner Chris and Adve Vikram. "LLVM: A Compilation Framework for Lifelong Program Analysis Transformation". In Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04), pp. 75-86, March 2004, Palo Alto, CA, U.S.A.

[7] David Lau, Orion Pritchard, Philippe Molson. "Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions". In Proc. of the 2006 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), April 2006, Napa Valley, CA, U.S.A.

[8] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. "OpenRCL: Low-Power High Performance Computing with Reconfigurable Devices". In Proc. of the 2010 International Conference on Field Programmable Logic (FPL), pp. 458-463, September, 2010, Milano, Italy.

[9] Llosa Josep, Gonzalez Antonio, Ayguade Eduard, Valero Mateo. "Swing Modulo Scheduling: A Lifetime-Sensitive Approach". In Proc. of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96), pp. 80-90, 1996, Washington, DC, U.S.A.

[10] Alexandros Papakonstantinou *et al*. "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs", In Proc. of the 7th Symposium on Application Specific Processors, pp.35-42, July, 2009, Boston, MA, U.S.A..

[11] Robert Schreiber *et al*. "PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators". Journal of VLSI Signal Processing, Vol.31, pp.127-142, June 2002.

[12] Justin L. Tripp, Maya B. Gokhale, Kristopher D. Peterson. "Trident: From High-Level Language to Hardware Circuitry". IEEE Computer Vol.40(3), pp.28-37, March 2007.

[13] Lu Yu, Feng Yi, Ding Jie, and Cixun Zhang. "Overview of AVS-video: tools, performance and complexity". In Proc. Visual Communications and Image Processing (VCIP), pp.679-690, July 2005, Beijing, China.

[14] Z. Zhang et al. "AutoPilot: A Platform-Based ESL Synthesis System". In "High-Level Synthesis: From Algorithm to Digital Circuit", Springer Netherlands, 2008, www.autoesl.com.

[15] Konstantis Daloukas, Christos D. Antonopoulos, Nikolaos Bellas. GLOpenCL: "OpenCL Support on Hardware- and Software-Managed Cache Multicores". In Proc. of 6th International Conference on High Performance Embedded Architectures & Compilers (HiPEAC), pp. 24-26, January, 2011, Heraklion, Greece.

[16] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere. "LAURA: Leiden Architecture Research and Exploration Tool". On Proc. of the 13th Int. conference on Field Programmable Logic and Applications (FPL' 03), pp. 911-920, September 2003, Lisbon, Portugal.

[17] Alex Solomatnikov *et al*. "Chip Multi-Processor Generator", in Proc. of the 44th Annual Design Automation Conference (DAC '07), 2007, New York, NY, U.S.A