Synthesis of Platform Architectures from OpenCL Programs

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

***Abstract*—The problem of automatically generating hardware modules from a high level representation of an application has been at the research forefront in the last few years. In this paper, we use OpenCL, an industry supported standard for writing programs that execute on multicore platforms and accelerators such as GPUs. Our architectural synthesis tool, SOpenCL (Silicon-OpenCL), adapts OpenCL into a novel hardware design flow which efficiently maps coarse and fine-grained parallelism of an application onto an FPGA reconfigurable fabric. SOpenCL is based on a source-to-source code transformation step that coarsens the OpenCL fine-grained parallelism into a series of nested loops, and on a template-based hardware generation back-end that configures the accelerator based on the functionality and the application performance and area requirements. Our experimentation with a variety of OpenCL and C kernel benchmarks reveals that area, throughput and frequency optimized hardware implementations are attainable using SOpenCL.**

***Keywords*-Multithreading, OpenCL, Electronic Design Automation, Reconfigurable Computing, Embedded Systems, FPGA**

I. INTRODUCTION

The advent of computing platforms with tens or hundreds of processing elements has been characterized as an inflection point for the computing systems industry. Hardware architects opt to offer more computing elements, instead of few and more powerful ones. As manycore architectures enter the mainstream of computing, there is a pressing demand for high-level programming paradigms that can effectively map algorithms to different parallel architectures without requiring heroic programmer efforts.

Motivated by this observation, we introduce SOpenCL (Silicon OpenCL), a novel tool and methodology which generates hardware accelerators and System On Chip systems (Fig. 1) from OpenCL programs. SOpenCL facilitates the generation of application-specific hardware by the expanding body of parallel programmers, who are more common than hardware developers, thus reducing the cost, effort and time-to-market.

OpenCL [5] is an industry-supported standard for building parallel applications that are portable across heterogeneous parallel platforms. It relieves the programmer from the burden of dealing with platform specific technicalities and limitations whenever possible, thus allowing her to focus on the application itself. We briefly discuss OpenCL and an OpenCL example in section II.

There are some major challenges in the design and implementation of SOpenCL. OpenCL programs typically use kernels for expressing parallelism at its finest granularity. This is a particularly convenient feature for hardware generation, as the programmer exposes all available parallelism to the underlying tool-chain. However, mapping this parallelism in a straightforward way, namely assigning a separate hardware accelerator to each parallel path, is clearly unrealistic due to lack of resources in reconfigurable fabric. SOpenCL includes a source-to-source translator to coarsen the granularity of the kernel functions from a per-logical-thread to a per-work-group basis, thus reducing overheads (section II.B). The outcome from the translator is a pure C function which represents the work that must be executed by each work-group.

The following stage of the tool flow, architectural synthesis, converts the C code to a hardware accelerator in synthesizable HDL. We use a well-crafted architectural template that can be instantiated to match the performance requirements, and available FPGA resources for a particular application (section III.A.). A number of compiler transformations and optimizations such as predication (section III.B.1), code slicing (section III.B.2) and modulo scheduling (section III.B.3) are successively applied to the initial C code before SOpenCL back-end produces HDL code (section III.C).

A run-time system (section IV) is used to spawn a number of threads used as execution vehicles of the main program of OpenCL and as helper threads to initialize, control and monitor the functionality of the hardware accelerator.

The major contributions of this paper are the following:

- The design and implementation of a tool flow to convert OpenCL applications into a SoC design with hardware and software components.
- A template-based hardware accelerator generation methodology which produces designs with decoupled

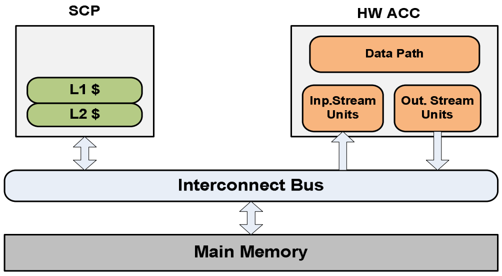


Figure 1. Target System On Chip design. SCP stands for Scalar Processor.

978-0-7695-4301-7/11 $26.00 © 2011 IEEE
DOI 10.1109/FCCM.2011.19

memory access and computational units.

- Although techniques such as thread serialization, elimination of synchronization operations and variable privatization have been proposed recently (e.g. for Cell processor in [7]) for granularity coarsening, this is the first time they are applied in the context of a hardware accelerated SoC platform.

II. SOPENCL FRONT END

A. OpenCL Programming Model

OpenCL [5] programmers typically express parallelism of the computationally intensive parts of their applications at its finest granularity, by embodying the task executed by a single logical thread in a *kernel*. Multiple such threads (called *work-items* in OpenCL) are combined to form *work-groups*, and multiple work-groups are combined to form a *grid* of computation. OpenCL provides functionality for barrier synchronization among work-items that belong to the same work-group. On the other hand, work-groups are by definition independent to each other and can always execute concurrently.

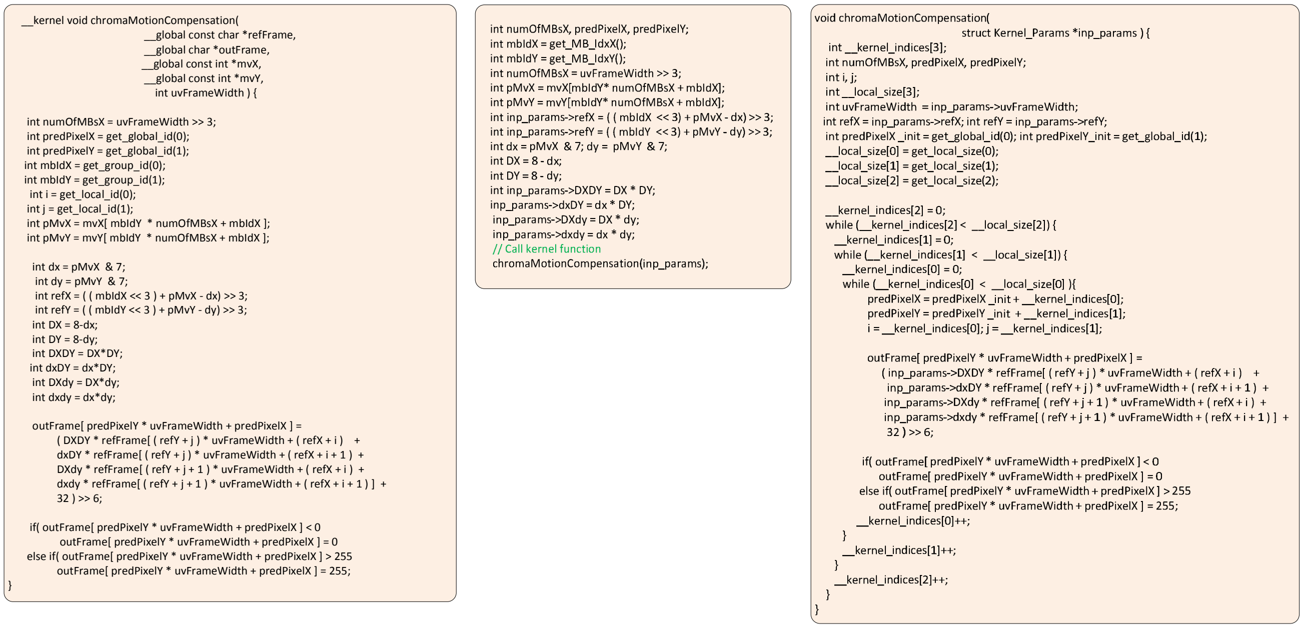
An OpenCL kernel which implements the *chroma interpolation* of the AVS video decoding algorithm [10] is used as a running example to explain the sequence of steps to generate the hardware accelerator (Fig. 2). Chroma interpolation computes intermediate values at a quarter pixel precision of the chroma plane of an image. The interpolated chroma value is given as the weighted average of the four neighboring chroma pixels A, B, C and D at integer locations as follows:

$$outFrame[y][x] = [(8 - dx) * (8 - dy) * A + dx * (8 - dy) * B + (8 - dx) * dy * C + dx * dy * D + 32] \gg 6$$

where *dx/dy* is the fractional horizontal/vertical distance of the quarter pixel from the location of integer pixel A multiplied by 8. Each work-item executes the code of Fig. 2a to calculate one of the $N_1 x N_2$ quarter pixels of a frame with N_1 columns and N_2 rows.

Geometrical partitioning is a familiar concept in data-parallel programming models. In OpenCL, the grid computation is partitioned in a 3-dimensional space of work-groups, and each work-group in a 3-dimensional space of work-items. 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 (Fig. 2a) 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.

Exposing parallelism at its finest granularity facilitates the use of OpenCL as a hardware description language, as it allows hardware generation at different levels of granularity. Another favorable feature of OpenCL is the explicit – yet not overly detailed – expression of data movement in the form of buffer transfers among the host and the compute devices. Languages with C-like semantics, as well as traditional parallel programming models such as POSIX Threads or OpenMP, express parallelism at a coarser granularity and at the same time ignore or obfuscate



```
__kernel void chromaMotionCompensation(
                                  __global const char *refFrame,
                                  __global char *outFrame,
                                 __global const int *mvX,
                                  __global const int *mvY,
                                    int uvFrameWidth ) {

  int numOfMBsX = uvFrameWidth >> 3;
  int predPixelX = get_global_id(0);
  int predPixelY = get_global_id(1);
 int mbIdX = get_group_id(0);
 int mbIdY = get_group_id(1);
   int i = get_local_id(0);
  int j = get_local_id(1);
  int pMvX = mvX[ mbIdY  * numOfMBsX + mbIdX ];
  int pMvY = mvY[ mbIdY  * numOfMBsX + mbIdX ];

   int dx = pMvX  & 7;
    int dy = pMvY  & 7;
    int refX = ( ( mbIdX << 3 ) + pMvX - dx) >> 3;
    int refY = ( ( mbIdY << 3 ) + pMvY - dy) >> 3;
   int DX = 8-dx;
   int DY = 8-dy;
   int DXDY = DX*DY;
  int dxDY = dx*DY;
   int DXdy = DX*dy;
   int dxdy = dx*dy;

   outFrame[ predPixelY * uvFrameWidth + predPixelX ] =
          ( DXDY * refFrame[ ( refY + j ) * uvFrameWidth + ( refX + i )    +
          dxDY * refFrame[ ( refY + j ) * uvFrameWidth + ( refX + i + 1 )  +
          DXdy * refFrame[ ( refY + j + 1 ) * uvFrameWidth + ( refX + i )  +
          dxdy * refFrame[ ( refY + j + 1 ) * uvFrameWidth + ( refX + i + 1 ) ]  +
          32 ) >> 6;

  if( outFrame[ predPixelY * uvFrameWidth + predPixelX ] < 0
           outFrame[ predPixelY * uvFrameWidth + predPixelX ] = 0
 else if( outFrame[ predPixelY * uvFrameWidth + predPixelX ] > 255
          outFrame[ predPixelY * uvFrameWidth + predPixelX ] = 255;
}
```

(a)

```
int numOfMBsX, predPixelX, predPixelY;
int mbIdX = get_MB_IdxX();
int mbIdY = get_MB_IdxY();
int numOfMBsX = uvFrameWidth >> 3;
int pMvX = mvX[mbIdY* numOfMBsX + mbIdX];
int pMvY = mvY[mbIdY* numOfMBsX + mbIdX];
int inp_params->refX = ( ( mbIdX  << 3) + pMvX - dx) >> 3;
int inp_params->refY = ( ( mbIdY  << 3) + pMvY - dy) >> 3;
int dx = pMvX  & 7; dy =  pMvY  & 7;
int DX = 8 - dx;
int DY = 8 - dy;
int inp_params->DXDY = DX * DY;
inp_params->dxDY = dx * DY;
 inp_params->DXdy = DX * dy;
 inp_params->dxdy = dx * dy;
 // Call kernel function
 chromaMotionCompensation(inp_params);
```

(b)

```
void chromaMotionCompensation(
                                        struct Kernel_Params *inp_params ) {
   int __kernel_indices[3];
  int numOfMBsX, predPixelX, predPixelY;
  int i, j;
  int __local_size[3];
  int uvFrameWidth  = inp_params->uvFrameWidth;
 int refX = inp_params->refX; int refY = inp_params->refY;
  int predPixelX _init = get_global_id(0); int predPixelY_init = get_global_id(1);
  __local_size[0] = get_local_size(0);
  __local_size[1] = get_local_size(1);
  __local_size[2] = get_local_size(2);

  __kernel_indices[2] = 0;
  while (__kernel_indices[2] <  __local_size[2]) {
    __kernel_indices[1] = 0;
    while (__kernel_indices[1]  <  __local_size[1]) {
      __kernel_indices[0] = 0;
      while (__kernel_indices[0]  <  __local_size[0] ){
             predPixelX = predPixelX _init + __kernel_indices[0];
             predPixelY = predPixelY _init  + __kernel_indices[1];
             i = __kernel_indices[0]; j = __kernel_indices[1];

             outFrame[ predPixelY * uvFrameWidth + predPixelX ] =
                 ( inp_params->DXDY * refFrame[ ( refY + j ) * uvFrameWidth + ( refX + i )    +
                  inp_params->dxDY * refFrame[ ( refY + j ) * uvFrameWidth + ( refX + i + 1 )  +
                 inp_params->DXdy * refFrame[ ( refY + j + 1 ) * uvFrameWidth + ( refX + i )  +
                 inp_params->dxdy * refFrame[ ( refY + j + 1 ) * uvFrameWidth + ( refX + i + 1 ) ]  +
                  32 ) >> 6;

            if( outFrame[ predPixelY * uvFrameWidth + predPixelX ] < 0
                 outFrame[ predPixelY * uvFrameWidth + predPixelX ] = 0
           else if( outFrame[ predPixelY * uvFrameWidth + predPixelX ] > 255
                 outFrame[ predPixelY * uvFrameWidth + predPixelX ] = 255;
          __kernel_indices[0]++;
      }
      __kernel_indices[1]++;
    }
    __kernel_indices[2]++;
  }
}
```

(c)

Figure 2. (a) The OpenCL code for AVS chroma interpolation. (b), (c) The equivalent main thread (executed by the host scalar processor) and kernel C code (used to generate the hardware accelerator) after coarsening the granularity to the equivalent of a work-group. Loop invariant computations are hoisted outside the kernel.

communication. This inhibits the exploitation of all available parallelism to architectures where this would be possible, or to create efficient hardware, thus placing the burden of re-discovering parallelism and communication patterns to an optimizing compiler and/or the user – typically with limited success. Although we only describe granularity management (coarsening) at the work-group level, it is possible to generate hardware in different levels of granularity using OpenCL [8].

B. *Front End Transformations*

To enable efficient execution of OpenCL kernel functions, we apply a series of source-code transformations to coarsen the granularity of the kernel function from a per-logical-thread to a per-work-group basis, thus reducing overheads and respecting hardware constraints. After the transformations, the modified kernel function represents the work that must be executed by each work-group in the index space of the application.

The transformation process consists of two main steps:

1) Logical Thread Serialization: In the absence of synchronization operations, work-items (i.e. logical threads) inside a work-group can be executed in any sequence. We enclose the instructions in the body of a kernel function within a triple-nested loop – given that the maximum number of OpenCL allowable work-group dimensions is three – thus executing the logical threads in sequence. Fig. 2c shows the C code after thread serialization.

The selection of a work-group as the preferred degree of granularity for logical threads serialization may seem arbitrary. However, in the next section it will become evident that other options may introduce complications in the presence of synchronization operations or multiple exit points within the kernel.

2) *Elimination of Barriers and Variable Privatization:* The next transformation addresses the problems introduced by synchronization operations or multiple exit points within a kernel. OpenCL provides barriers to allow synchronization of work-items inside a work-group. In the presence of a barrier, all work-items in the work-group must execute the barrier instruction before any of them is allowed to continue execution beyond the barrier. Similarly, a barrier command inside a loop implicitly enforces all work-items to execute the barrier before the next loop iteration.

To ensure correct execution of the coarsened kernel functions, we partition the statements into blocks, so that each block contains no synchronization operations (Fig. 3a) and apply loop fission around each synchronization statement. Thus, two loop constructs are required to ensure correct execution of the coarsened kernel function. A similar problem occurs when the kernel code includes a barrier instruction inside conditional statements or loops. OpenCL requires that the barrier statement be encountered by all work-items executing the kernel [5]. Fig. 3b and Fig. 3c show the body of an OpenCL kernel that contains conditional statements and loops, and the code transformations to pure C with equivalent semantics.

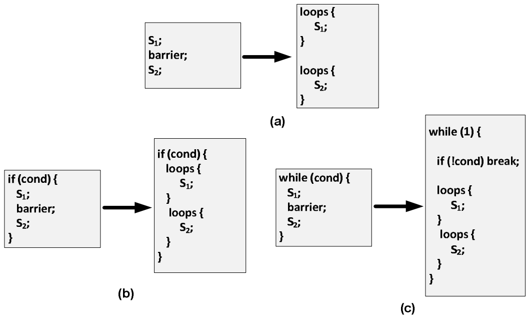


Figure 3. Elimination of barriers.

We follow a similar approach for kernel functions with multiple exit points, i.e. when statements that change the control flow are present, such as continue, break, or return. We treat such statements similarly to synchronization points and perform loop fission around them.

After applying loop fission around synchronization or control flow statements, the compiler needs to cope with variables whose lifeline crosses loop fission points. Once serialization is applied, logical threads that belong to a work-group share the memory corresponding to local variables. This introduces a complication for variables whose life extends beyond a synchronization point or a control flow statement. Values assigned by logical-threads at the first loop construct introduced by loop fission cannot be used during the execution of the second loop construct, as their content has been overwritten by the execution of subsequent logical threads, thus violating semantics.

Our compilation infrastructure conducts a live variable analysis to identify the variables that are live beyond the boundaries of the loops introduced by loop fission. Following, we apply variable privatization for these variables, namely we allocate them to a separate memory area for each logical thread. Each logical thread is therefore provided with a private copy of such variables.

A more detailed discussion of source code transformations can be found in [3].

III. ARCHITECTURAL SYNTHESIS

After the front-end transformations, the hardware generation flow generates the synthesizable HDL of the accelerator as shown in Fig. 4. The SOpenCL tool flow, which extends the LLVM [6] compiler infrastructure, currently supports C code with only one nested loop as shown in Fig. 2c. An important advantage of the SOpenCL

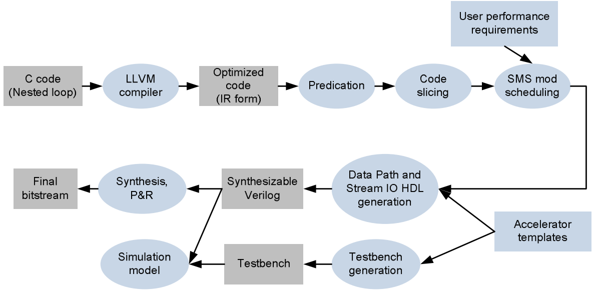


Figure 4. Hardware generation tool flow.

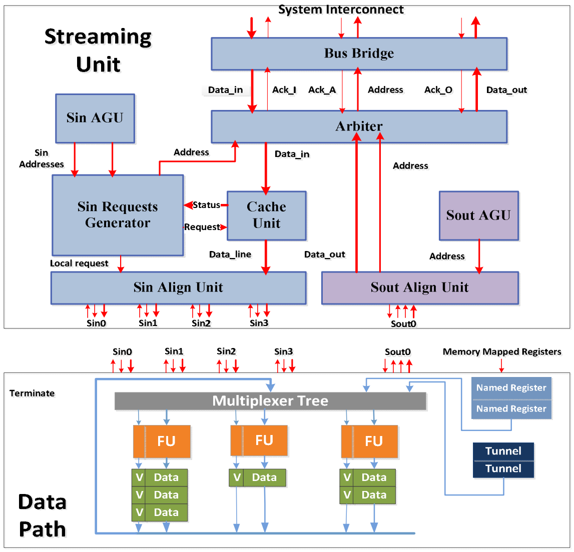


Figure 5. The hardware accelerator template includes the Data Path and the Stream Units.

methodology is that it leverages the structure of a well-crafted architectural template instead of producing arbitrary HDL code.

A. *Architectural template*

SOpenCL uses an architectural template that can be instantiated to match the specific target application, user performance requirements, and the available FPGA resources (Fig. 5). The architecture decouples and overlaps data accesses and computations, thus minimizing the effects of memory access latency.

The datapath of Fig.5 consists of a network of functional units (FUs) that produce and consume data elements using explicit FIFO channels to the stream units. A reconfigurable link is formed by a tree of multiplexers and buffers to direct proper data elements from the output of a producing functional unit to the input of the next consuming functional units. The control logic is distributed and spatially near the corresponding functional units, multiplexers, and buffers. Unlike a centralized VLIW codeword which tends to increase the signal critical path, distributed control logic avoids long interconnects in critical paths and is more suitable for FPGA implementation.

The reconfigurable parameters of the datapath 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 interconnect between functional units, and the bandwidth to and from the streaming unit.

The streaming unit handles all issues regarding data transfers between the memory and the datapath. 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 (Processor Local Bus - PLB for a Xilinx FPGA [1]), and the bandwidth requirements of the datapath.

An Address Generation Unit (AGU) aggressively generates addresses for data prefetching and write back, and feeds them to the Address Request Module. SOpenCL 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 Requests Generator module coalesces requests generated by “Sin-AGU” (the input data AGU) to the word width of the underlying memory interconnect, or to burst size if bursting is enabled, and competes for memory accesses with the other stream units. Before issuing a transaction request to the Arbiter it checks if the addresses aliases 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.

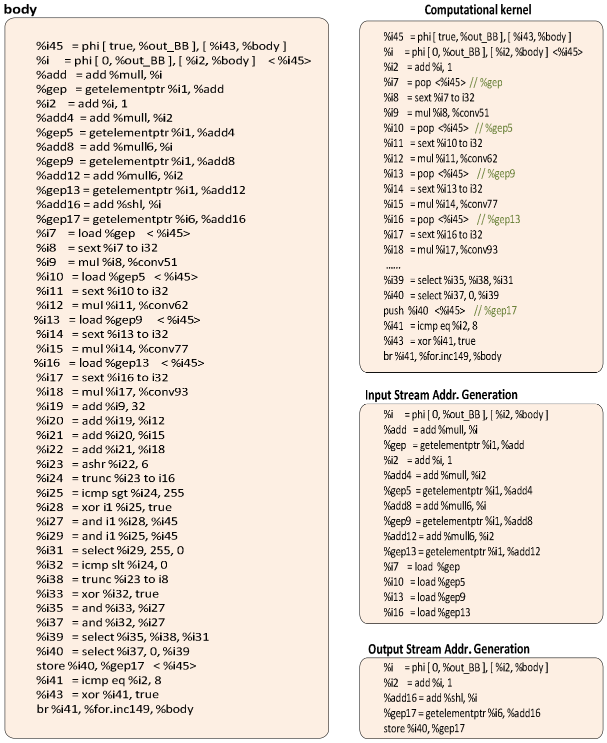
The input stream Alignment Unit retrieves the data from the Cache Unit (or the *data_in* incoming data channel if there is no cache), eliminates any gaps between successive elements due to a stride greater than one and presents the data tokens in-order to the data path. The output stream Alignment Unit aligns the incoming data tokens from the data path in a line of *Bus_Width* bytes. As soon as the line is full or an incoming data token falls out of the address range of the specific line, the Alignment Unit issues the write request to the arbiter.

Finally, the arbiter regulates the access of the stream units to the PLB system bus. It uses a round-robin algorithm, and its complexity depends on the number of input and output streams of the application.

B. *Low level Transformations*

1) LLVM Compiler Optimizations and Predication: Prior to modulo scheduling and hardware generation, SOpenCL utilizes the LLVM compiler framework [6] to generate the LLVM intermediate representation (IR) and perform standard compiler optimizations as code motion of loop-invariant instructions, redundant instructions elimination, constant propagation, etc.

Predication is a combined software / hardware mechanism that supports conditional execution of instructions based on boolean guards, typically implemented as 1-bit predicate. It allows control dependencies to be converted to data dependencies and facilitates efficient



body

```
%i45   = phi [ true, %out_BB ], [ %i43, %body ]
%i     = phi [ 0, %out_BB ], [ %i2, %body ]    < %i45>
%add   = add %mull, %i
%gep   = getelementptr %i1, %add
%i2    = add %i, 1
%add4  = add %mull, %i2
%gep5  = getelementptr %i1, %add4
%add8  = add %mull6, %i
%gep9  = getelementptr %i1, %add8
%add12 = add %mull6, %i2
%gep13 = getelementptr %i1, %add12
%add16 = add %shl, %i
%gep17 = getelementptr %i6, %add16
%i7    = load %gep    < %i45>
%i8    = sext %i7 to i32
%i9    = mul %i8, %conv51
%i10   = load %gep5   < %i45>
%i11   = sext %i10 to i32
%i12   = mul %i11, %conv62
%i13   = load %gep9    < %i45>
%i14   = sext %i13 to i32
%i15   = mul %i14, %conv77
%i16   = load %gep13    < %i45>
%i17   = sext %i16 to i32
%i18   = mul %i17, %conv93
%i19   = add %i9, 32
%i20   = add %i19, %i12
%i21   = add %i20, %i15
%i22   = add %i21, %i18
%i23   = ashr %i22, 6
%i24   = trunc %i23 to i16
%i25   = icmp sgt %i24, 255
%i28   = xor i1 %i25, true
%i27   = and i1 %i28, %i45
%i29   = and i1 %i25, %i45
%i31   = select %i29, 255, 0
%i32   = icmp slt %i24, 0
%i38   = trunc %i23 to i8
%i33   = xor %i32, true
%i35   = and %i33, %i27
%i37   = and %i32, %i27
%i39   = select %i35, %i38, %i31
%i40   = select %i37, 0, %i39
store %i40, %gep17   < %i45>
%i41   = icmp eq %i2, 8
%i43   = xor %i41, true
br %i41, %for.inc149, %body
```

Computational kernel

```
%i45   = phi [ true, %out_BB ], [ %i43, %body ]
%i     = phi [ 0, %out_BB ], [ %i2, %body ]  <%i45>
%i2    = add %i, 1
%i7    = pop  <%i45> // %gep
%i8    = sext %i7 to i32
%i9    = mul %i8, %conv51
%i10   = pop  <%i45>   // %gep5
%i11   = sext %i10 to i32
%i12   = mul %i11, %conv62
%i13   = pop  <%i45>   // %gep9
%i14   = sext %i13 to i32
%i15   = mul %i14, %conv77
%i16   = pop  <%i45>   // %gep13
%i17   = sext %i16 to i32
%i18   = mul %i17, %conv93
 ......
%i39   = select %i35, %i38, %i31
%i40   = select %i37, 0, %i39
push  %i40   <%i45>   // %gep17
%i41   = icmp eq %i2, 8
%i43   = xor %i41, true
br %i41, %for.inc149, %body
```

Input Stream Addr. Generation

```
%i     = phi [ 0, %out_BB ], [ %i2, %body ]
%add   = add %mull, %i
%gep   = getelementptr %i1, %add
%i2    = add %i, 1
%add4  = add %mull, %i2
%gep5  = getelementptr %i1, %add4
%add8  = add %mull6, %i
%gep9  = getelementptr %i1, %add8
%add12 = add %mull6, %i2
%gep13 = getelementptr %i1, %add12
%i7    = load  %gep
%i10   = load %gep5
%i13   = load %gep9
%i16   = load %gep13
```

Output Stream Addr. Generation

```
%i     = phi [ 0, %out_BB ], [ %i2, %body ]
%i2    = add %i, 1
%add16 = add %shl, %i
%gep17 = getelementptr %i6, %add16
store %i40, %gep17
```

Figure 6. (a) Single basic block LLVM code after if-conversion. Predicate variables are shown in brackets at the end of the instruction. (b) Slicing of the predicated code in three kernels: the Computational and the In/Out-Streaming Address Generation Kernels. Note that the load/store instructions in (a) have been replaced by pop/push instructions in the computational kernel in (b).

instruction scheduling and hardware generation. All branches (except the backward branch of the inner loop) are removed, and their functionality is replaced with *select* instructions and predicate variables.

Predicates such as *%i25, %i28* etc. in Fig. 6b are computed using 1-bit predicate define functions. Our scheme supports a partially predicated instruction set. *Store* instructions are predicated because they could inadvertently modify the architectural state of the system if they belong to the false path. Likewise, *load* instructions are predicated to avoid fetching data that will not be needed by the data path.

2) Code Slicing: The single basic block, predicated LLVM code of Fig. 6b encapsulates both the data I/O and computation of the coarsened kernel. A code slicing step partitions the predicated code to three slices (Fig. 6c):

- Input Stream Kernel, which consists of all *load* instructions and any instruction that participates in the calculation of load addresses. The kernel drives the hardware generation of the Input Stream AGU.
- Output Stream Kernel, similar to the previous one for *store* instructions. It drives the hardware generation of the Output Stream AGU.
- 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 read/write element without the need to specify a memory address. The Computational Kernel drives the hardware generation of the data path.

This asynchronous data flow model allows data to be fetched ahead of computation provided there are no inter-kernel data 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 < local_size[0]; 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 this problem, we duplicate the code used to compute data-dependent addresses when we perform *code slicing.* Referring to the example code above, the Input Stream AGU computes the address *&ptr[i],* performs the addition (*ptr[i]+1*) on the incoming data stream *ptr[i],* computes the address *&a[ptr[i]+1]* and fetches stream *a*. The Computational kernel reads streams *ptr[i]* and *a[ptr[i]+1]* from the Input Stream Unit*,* and performs the addition *ptr[i]+1*.

An unfortunate side-effect of computation-driven memory accesses is that they may hinder data prefetching. The Input Stream Units may have to wait for data to come from memory before they issue new read commands, hence reducing effective bandwidth and increasing latency.

3) *Swing Modulo Scheduling (SMS):* Modulo scheduling is an instruction scheduling technique that exploits instruction level parallelism in loops by overlapping successive loop iterations and executing them in parallel. We use SMS [9] to generate a schedule for each of the three kernels described in section III.B.2. SMS uses heuristics to minimize the Initiation Interval (II), i.e. the constant interval between the start of successive loop iterations, which is the main factor affecting computational throughput. SMS also reduces the time each intermediate variable is live. Long variable lives translate into larger ALU queues, and lead to unnecessarily large data paths.

C. Hardware generation

Fig. 7 shows the block diagram of the Input Stream AGU which implements the scheduled code of Fig.6c (for brevity, we do not show the block diagram of the computational kernel). The output port feeds addresses to the Address Request Module as shown in the template of Fig. 5. Constant values %mull, %mull6 and %i1, are set in the outer loops. The hardware generator creates a small register file with three registers, and the control unit of the accelerator initializes them with the appropriate values at run-time.

Valid bits: Modulo scheduler generates an instruction schedule for the steady state body of the schedule, however not for the prologue and epilogue. The generated hardware utilizes a *valid bits* mechanism to facilitate the correct

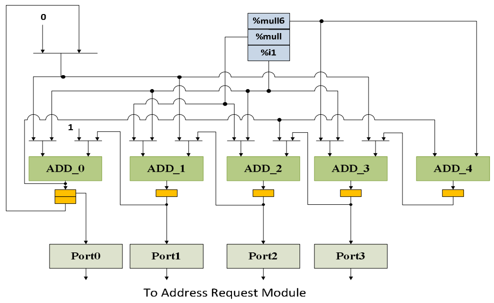


Figure 7. Block diagram of the Input Stream AGU with II=1.

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 on a FIFO 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 datapath produces valid data at the correct loop iteration of the modulo schedule, thus implicitly implementing the prologue and epilogue of the schedule.

Valid bits are also used to implement predication. When a predicate input to a functional unit is false, the valid bit at the output of the functional unit is also set to false, regardless of the value of the valid bits of inputs A and B. By tagging each data value with a valid bit, we can selectively ignore values that are produced by a false control path.

Cross-iteration dependencies

In addition to intra-iteration data dependencies, the data flow graph may also express cross iteration dependencies, such as in the following code:

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

Most compilers avoid costly memory spills and optimize the code by allocating a register to temporarily store the live variable *a[i]* before its next use (as *a[i-1]*). We use a similar approach by introducing *tunnels*, a small set of registers organized as a FIFO queue. In case of loop unrolling by the LLVM compiler, tunnels between unrolled iterations become CDFG (Control/Data Flow Graph) arcs and tunneling occurs only between iterations of distance larger than the degree of unroll.

SOpenCL only instantiates tunnels when the dependence distance between the *definition* and the *use* of a variable in the iteration space is a small constant known at compile-time. Otherwise, array variables are spilled to memory and it is the responsibility of the programmer to avoid overlapping source and destination arrays. Hardware does not check that there are no Read-After-Write dependence violations.

If SOpenCL detects irregular memory dependencies, it bypasses the slicing step and generates a unified architecture to reduce the area cost at the same latency cost.

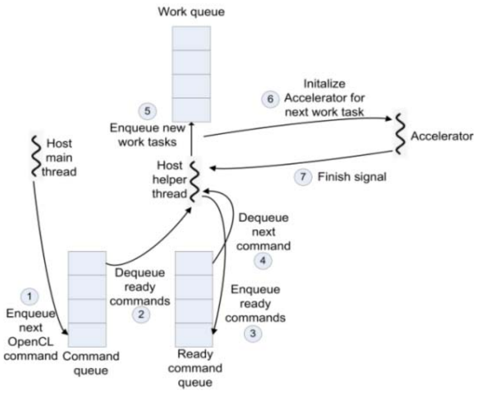


Figure 8. The run-time system architecture for a platform comprising one scalar processor and one hardware accelerator. The numbering denotes a typical sequence of operations.

IV. SOPENCL RUN-TIME

The OpenCL main program is executed as a main thread in the host processor of the platform (PowerPC 440 in Virtex-5 FPGAs) as shown in Fig. 8. The main thread is responsible for managing and monitoring the execution of the computational kernels on the FPGA platform and running the portions of the application code not assigned to a hardware accelerator.

OpenCL expresses kernel invocations and data buffer transfers as commands, which are queued to a *command queue*. A separate helper thread that runs concurrently with the main thread continuously executes a scheduling loop. It transfers commands to the ready queue when the commands are ready to be executed. More specifically, when in-order execution is used, a command is transferred to the ready queue if it is at the top of the command queue, and provided that all previously issued commands have finished their execution. When the command queue is configured for out-of-order execution, a dependence driven self-scheduling scheme is used. Each command is marked as dependent to one or more previous commands, and is enqueued to the ready queue as soon as all of its dependencies are satisfied.

Work-tasks are created by the helper thread when a command related to an accelerator execution is processed from the ready queue. A work-task corresponds to the initialization and invocation of the accelerator which implements the modified kernel function. Referring back to Fig. 2, the helper thread initializes the accelerator with input arguments *inp_params* needed for the execution of the coarsened kernel. Input arguments are stored in memory-mapped registers in the accelerator. After initialization, the helper thread initiates execution of the hardware accelerator.

The hardware accelerator is responsible for reading input data and writing the resulting output data back to memory, without any involvement of the helper thread. When the accelerator finishes executing a work task, it notifies the run-time system and the helper thread initiates a new accelerator invocation if there are available work tasks in the work queue.

Table I. Applications used for SOpenCL evaluation. *Pred* indicates if the benchmark contains a complex control flow and the translator has to apply predication.

App.	Description	Working set	Pred.
sAdd	Addition of two vectors	Two 16,768 vectors	No
MatMul	Matrix Multiplication	64×64 Matrices	No
Conv	Convolution	7×7 filter, 16×16 array	No
1D-DCT	1-dimensional DCT	1-D vector: 1024 Entries	No
CMC	Chroma Motion Compensation(Video)	16×16 Block	Yes
LMC	Luma Motion Compensation(Video)	16×16 Block, Quarter Pixel	Yes

Table II. Area result of the three configurations. The first three applications have been synthesized with caching disabled, whereas the last three with caching enabled.

App.	C_A			C_B			C_C		
	II	I/O	Slices	II	I/O	Slices	II	I/O	Slices
sAdd	2	2/2	835	1	2/2	862	1	2/2	862
MatMul	2	2/2	675	2	4/2	732	2	4/2	732
Conv	2	2/2	912	2	4/2	967	2	4/2	967
1D-DCT	32	2/2	1549	16	4/4	1603	2	8/8	1920
CMC	7	2/1	923	4	4/1	1015	1	4/1	1124
LMC	38	4/1	2990	8	8/1	3922	2	16/1	5348

V. EXPERIMENTAL EVALUATION

A. Methodology

We tested the SOpenCL methodology on six OpenCL and C applications (Table I). *Luma quarter-pixel interpolation (LMC)*, the most complex benchmark of the group, consists of a large conditional statement comprising 16 separate cases. Each kernel invocation processes an 8x8 pixel block and produces a new 8x8 interpolated block, always following the same conditional path.

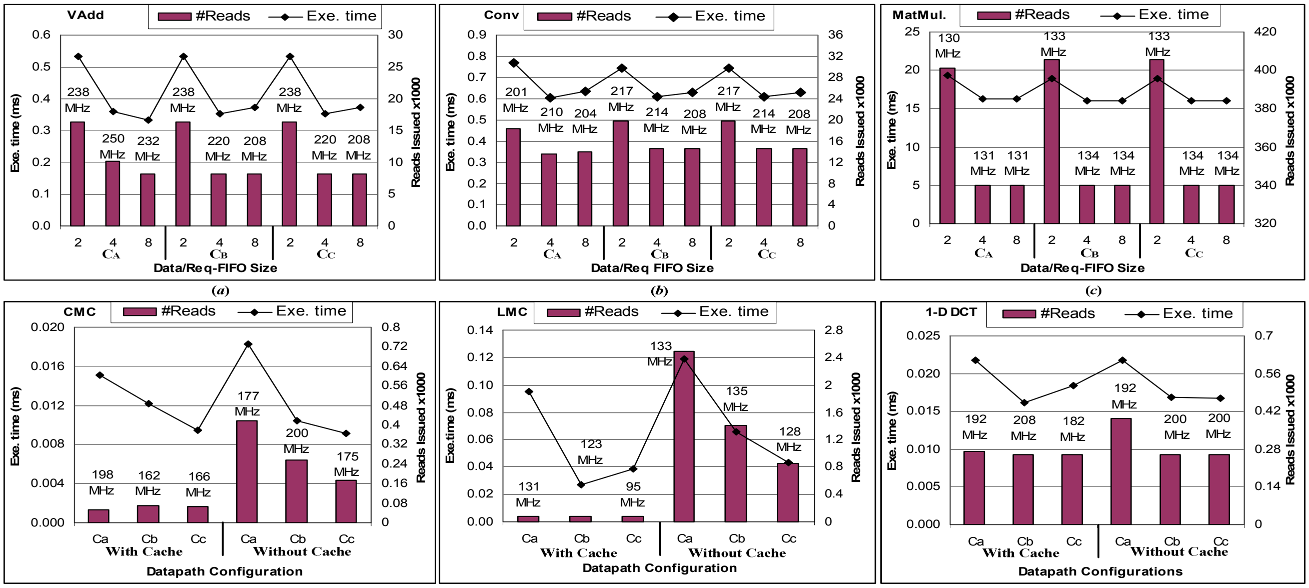
We used three different machine configurations *{C_A, C_B, C_C}* to guide modulo scheduling. They correspond to the allocated resources (FUs, I/O bandwidth) to implement the kernel architecture. Configuration C_A only provides the minimum resources necessary for accelerator execution. For example, it provides one 32-bit adder for the *sAdd* benchmark. C_C is a theoretically maximal configuration, describing systems with unlimited computational resources. Finally, C_B is a typical configuration between the two extremes, and is different for each benchmark. For example, for the LMC benchmark it allows for up to 320 ALU bits, 320 Shifter bits and 320 Multiplier bits. In a realistic design flow, the selection of a machine configuration should be driven by user requirements and available FPGA resources.

For the evaluation of our designs we used a Xilinx Virtex-5 FX70 FPGA which includes an embedded PowerPC 440 core. We used Xilinx ISE 11.4 toolset for synthesis, placement and routing.

B. Results

Table II shows detailed area results for the benchmarks. Note that C_C does not always result into a schedule with II=1 because some benchmarks have circular paths (recurrences) which place extra constraints to the schedule [9]. The I/O column shows the I/O bandwidth to (from) the computational kernel in bytes/cycle. The amount of logic slices tends to increase for machine descriptions with a large number of functional units. The cache unit is the only module that requires BRAMs. Either none or two BRAMs are instantiated.

Fig. 9 shows execution time and clock frequency after placement and routing. The size of the *Request FIFO* and *Data FIFO* in the *Sin Requests Module* affects performance



(*a*) (*b*) (*c*)

(*d*) (*e*) *f*)

Figure 9. Performance results for the six benchmarks of Table I under the three configurations *{C_A, C_B, C_C}* and three different sizes of Data/Request FIFOs (2, 4, 8}. Implementations without a cache have always a Data/Request FIFOs of size 8. We report the execution time (in ms), the clock frequency and the number of read requests issued in each case.

in architectures without a cache unit particularly for applications with spatial locality (Fig. 9.a, b, c). FIFOs with just two entries make it impossible to exploit all data-tokens within a line, resulting to more memory requests (Fig. 9 Reads axis). FIFOs with at least four entries provide full reuse, at least for the first iterations until the pipeline is full.

The cache unit is useful in holding data across outer-loop iterations, especially in applications with temporal locality. Implementing *LMC* (Fig. 9.e) without a cache leads to a significant increase in read requests, which in turn increases the execution time. On the other hand, although there is an increase of read requests for *CMC* (Fig. 9.d) without a cache, the additional reads seem to overlap with computation.

As expected, machine configuration affects speedup only when the kernel has a significant amount of computations as in CMC, LMC, and 1D-DCT (Fig. 9, d, e, f respectively). C_B configuration has a smaller II and hence results into lower execution time (Fig. 9.d, e, f). However, larger machine configurations as C_C in LMC and 1D-DCT (Fig. 9.e, f) produce routing complexities and large circuit size, which degrades clock rate and hence execution time.

Finally, we observe that clock frequency does not always scale with the number of available resources and computational bandwidth. The location of the critical path depends on the application and the iteration interval. For complex applications with large II (e.g. LMC Fig. 9.e), routing and multiplexing to the inputs of functional units dominates the critical path. In most other cases, the critical path can be in the address request module or the address generation unit, etc. and is therefore, less than straightforward to predict the clock frequency. The multiplexer tree dominates clock rate in small configurations and large computational kernels as found in C_A for LMC.

VI. RELATED WORK

Tools such as PICO [4] and MATCH [2] are some of numerous efforts to generate hardware from high level languages like C, Java, and Matlab.

Our work is closer to FCUDA, a CAD tool that converts CUDA kernels to synthesizable hardware [11]. SOpenCL translates a complete OpenCL program, not CUDA, and the target platform is a template-based accelerator with decoupled memory accesses and computation. SOpenCL uses unmodified OpenCL code as input and performs extensive compile and run-time analysis to attain all information required for the generation and utilization of efficient, application-specific hardware.

OpenRCL platform utilizes OpenCL to schedule fine-grain parallel threads to a large number of MIPS-like cores [8]. Although the cores can be configured according to extensible ISAs, OpenRCL does not generate customized hardware accelerators like SOpenCL.

VII. CONCLUSION

We presented the design, implementation and evaluation of SOpenCL, a tool flow to produce the hardware and software architecture of accelerator-based SoCs. SOpenCL corroborates the idea of a unified programming model, compilation and run-time infrastructure usable for both programming manycore systems and automatically generating hardware for FPGAs. In fact, techniques used for coarsening parallelism granularity and run-time system management can be used on diverse parallel systems with hardware- or software-managed cache memories.

We are currently investigating automating the configuration selection process based on the target-device and user performance requirements. We are also planning to extend the underlying architectural model to include kernels with multiple accelerators interconnected through customized memory hierarchies.

ACKNOWLEDGMENT

This work is partially supported by the EC Marie Curie International Reintegration Grant (IRG) 223819.

REFERENCES

[1] CoreConnect Architecture – Processor Local Bus. www.xilinx.com

[2] P. Banerjee et al. "A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems*," Proceedings of the IEEE Symposium on Field Custom Computing Machines (FCCM)*, April 17-19, 2000, pp. 39-48, Napa Valley, CA.

[3] K. Daloukas, C. D. Antonopoulos and N. Bellas, "GLOpenCL: OpenCL Support on Hardware- and Software-Managed Cache Multicores," *Proceedings of the 6th Symposium on High Performance and Embedded Architectures and Compilers (HiPEAC), January 21-24, 2011, pp 15-24. Irakleio, Greece.*

[4] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. Cronquist, and M. Sivaraman. "PICO: Automatically Designing Custom Computers,"*IEEE Computer Magazine*, vol. 35, no. 9, September 2002, pp. 39-47.

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

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

[7] J. Lee et al. "An OpenCL Framework for Heterogeneous Multicores with Local Memory, *"Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 11-15, 2010, pp. 193-204,Vienna, Austria.

[8] M. Lin, I. Lebedev, and J. Wawrzynek. «OpenRCL: Low-Power High Performance Computing with Reconfigurable Devices, *"Proceedings of the 2010 International Conference on Field Programmable Logic (FPL)*, September, 2010, pp. 458-463, Milano, Italy.

[9] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. "Swing Modulo Scheduling: A Lifetime-Sensitive Approach, *"Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT)*, October, 1996, pp. 80-86, Boston, MA.

[10] L. Yu, Y.Feng, D.Jie, and C. Zhang."Overview of AVS-video: tools, performance and complexity, "*Proceedings –SPIE The International Society for Optical Engineering*, vol. 5960, 2005, pp. 679-690.

[11] A. Papakonstantinou, G. Karthik, J. A. Stratton, D. Chen, C. Jason, and W.-M. W. Hwu. "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs*, "Proceedings of the 7th Symposium on Application Specific Processors*, July, 2009, pp.35-42, Boston, MA.