ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ
Η/Υ, ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
ΚΑΙ ΔΙΚΤΥΩΝ

Χρήση Μοντέλου Παράλληλου Προγραμματισμού για Σύνθεση Αρχιτεκτονικων

Μια εργασία που εκπονήθηκε από
τον Muhsen Owaida για τις απαιτήσεις του
Διδακτορικού Διπλώματος.

Επιβλέπων Καθηγητής: Αν. Καθηγητής, Νικόλαος Μπέλλας

Βόλος, Αύγουστος 2012

Acknowledgement

The work of this dissertation has been one of the most significant academic challenges I have ever had to face. Without the support, patience and guidance of the kind people around me this dissertation would not have been completed.

First of all, my enormous debt of gratitude goes to my thesis advisor and mentor, Professor Nikolaos Bellas. Throughout the period of my PhD studies, Professor Bellas was there for me to actively support and guide me toward taking the best possible decisions and to develop to a much better researcher. I am thankful for his patience and immense knowledge and help in writing this dissertation. I am especially thankful to him for his mentorship and friendship throughout these years.

I owe sincere and earnest thankfulness to my committee member, Professor Christos D. Antonopoulos for his constant guidance and insightful comments which have been invaluable on both an academic and a personal level, for which I am extremely grateful. It has been a great privilege for me to work under his guidance.

I would like to express my sincere gratitude to my committee member, Professor Georgios Stamoulis for his assistance and guidance in getting my PhD studies started. I would like to thank also my colleagues and fellow students, especially Konstantis Daloukas and Charalambos Antoniadis for their help and cooperation in my research. I would like to thank the staff of the Department of Computer and Communication Engineering at the University of Thessaly for their support and constant encouragement.

Finally, I wish to thank my family, who have always believed in me and helped me to reach my goals. Their support forged my desire to achieve all that I could in life.

I would like to acknowledge the financial support of the Greek State Scholarship Foundation (IKY) throughout the period of my PhD studies, which without it this research would not have begun.

dedicated to my family

TABLE OF CONTENTS

i

LIST OF FIGURES

iv

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

1.1 Background

1.1.1 Modern Parallel and Heterogeneous Computing

The ever increasing demand for more efficient computing has pushed the evolution of computing systems to spectacular levels over the last few decades. Advances in computing systems are the key to the development of new domains and revolutionary technologies, such as personalized medicine, online social interaction, and immersive entertainment experiences.

While appetite for high performance and more efficient computing is increasing, today's computing systems are struggling with technology limitations. The traditional way to improve performance by increasing clock frequency has already come to an end. As a result, computing systems are shifting towards energy-efficient parallel computation models. Using many slower parallel processors instead of a single high speed core has provided higher energy efficiency.

Parallel architectures developed over the last decade, can be classified into different categories. The first category includes multiple instances of the traditional general purpose processor have been arranged within the same chip to produce multi core processors (MCPs). Another category includes the Graphic Processing Units (GPUs) with hundreds of simple processing cores. Nvidia GeForce256 was the first GPU released on 1999 [1]. Finally, streaming/Vector processors are multi-core processors, specially designed for streaming applications. Streaming processors like *RSVP, Imagine, Raw, and Merrimac* [2, 3, 4, 5] promoted high performance computing by exploiting heavy data parallelism in streaming applications and employing a distributed memory model.

1

While the many-core processing hardware technology is progressing rapidly, software development for parallel computing is falling behind. The challenge rising with parallel computing systems is to port already developed software for sequential processors on the newly introduced multi- or many-core processors. To cope with the new architectural trends, the parallel computing industry has developed a variety of parallel programming languages to allow programmers to exploit the multiple execution contexts available in the new multi-core architectures. The first class of parallel programming languages like OpenMP and Posix threads are extensions of sequential programming models, suitable for systems with few processing cores, and are widely used in the industry. New parallel programming models have been invented in the last few years to better suit systems with hundred or thousands of cores. Languages such as OpenCL, CUDA or various streaming languages fit the second category.

Yet even the shift to parallel computing is not enough. Many-core chips suffer from high power density which restricts the number of cores that can be simultaneously active, a phenomenon called *dark silicon* [6, 7]. The dark silicon phenomenon puts limits on the prospect of building many-core chips with tens or hundreds of cores without significant degradation in efficiency. This inefficiency is promoting *heterogeneous parallel computing systems*.

Instead of a parallel computing system built only from many-core chips, a heterogeneous computing system comprises multiple different computing components (Figure 1.1) each carefully optimized to efficiently execute a particular type of task. This heterogeneous parallel computing model presents an even greater challenge for developers. Now they must not only develop parallel applications, but they are responsible for deciding what types of processors to use for which calculations [6].

Heterogeneous systems development represents the best approach on energy-efficient high performance computing. However, it is a new technology that requires extensive research and effort mostly in developing tools and compilers to help software developers to deal with the large pool of architectural variables and parameters of heterogeneous systems. Other than the architectural differences of heterogeneous system components, their programming tools and languages exhibit
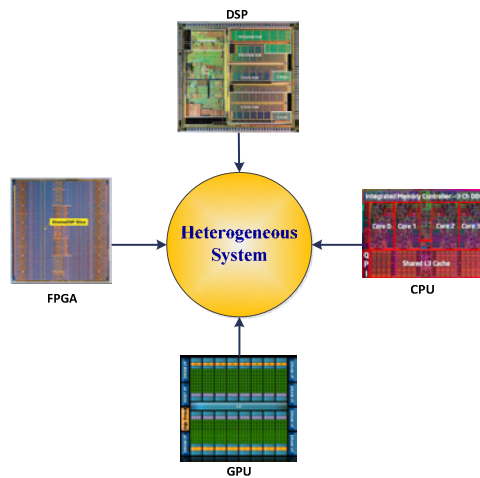
Figure 1.1: Heterogeneous System.

vast differences making it extremely difficult to develop applications that can be executed on all components. For example, porting an application on a heterogeneous system comprising MCPs, GPUs and FPGAs, requires the use of completely different programming languages; for example OpenMP for MCPs, CUDA for GPUs, and Verilog/VHDL for FPGAs

Recently, researchers in the parallel computing community have been moving towards unified programming models to support the heterogeneity of parallel computing platforms. *OpenCL* [22] is an industry-supported standard for building parallel applications that are portable across heterogeneous parallel systems. OpenCL adopts an architecture-agnostic computations model, promoting application portability across different platforms.

1.1.2 FPGA-based Computing Platforms

The recent advances in FPGA technology have placed reconfigurable platforms on the map of heterogeneous computing. FPGA accelerators offer superior performance, power and cost characteristics compared to a homogeneous CPU-based platform, at the expense of complex and expensive software infrastructure. For instance, FPGAs have been shown to offer two orders of magnitude superior performance than conventional CPUs for a variety of data-intensive applications [8].

3

Research in the last few years provided strong evidence on FPGA high performance computing capabilities. Applications in medical imaging [9], networking [10], multimedia [11], and financial applications [12], have been successfully implemented on FPGA platforms achieving orders of magnitude speedup and energy-consumption reductions over CPU- and GPU-based solutions.

Distributed logic and memory components of FPGA devices bear a significant resemblance to many-core processors. FPGA reconfigurable fabric consists of a sea of programmable logic cells and interconnects organized in rows and columns (Figure 1.2). Recently, FPGA manufacturers have included hard IP cores, like multipliers and SRAM blocks, distributed within the logic cells to improve designs efficiency. The distributed memory blocks over the FPGA architecture, provide the necessary memory bandwidth for building parallel computing architectures.

Developing FPGA-based systems is a hard undertaking and a time consuming process. The designer requires firstly analyzing the problem under consideration, partition it into multiple tasks, each then implemented carefully to fulfill the performance requirements. The design then has to be implemented using a hardware description language like Verilog or VHDL before programming the FPGA device.

Even with FPGA-based computing being up to the expectations of the high performance community, the integration of FPGAs in heterogeneous systems composed of CPUs and GPUs is far from mainstream. The main obstacle in the way of FPGAs being used in heterogeneous platforms is the need for hardware expertise
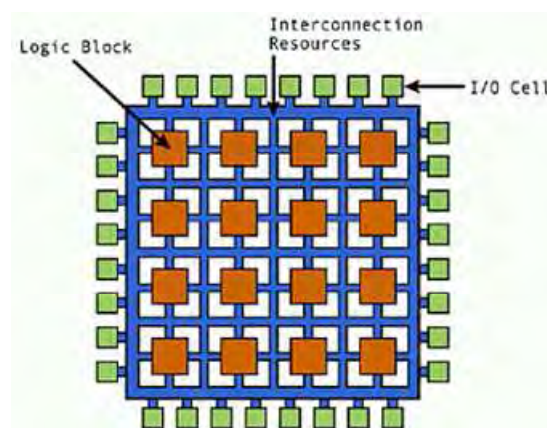
Figure 1.2: FPGA fabric basic components.

4

to program the FPGA. The community of software programmers and especially programmers of parallel systems will resist a platform with its own programming language when the industry is moving towards unified higher-level programming models for multi-core and heterogeneous platforms. Using FPGAs in heterogeneous platforms ideally requires enabling FPGA programming using high level parallel programming languages like CUDA, and OpenCL.

1.2   Research Objective and Contribution

The problem of automatically generating system architectures from high level programming languages has been at the forefront of academic and industrial research in the last few decades. Generating system designs from high level programming languages such as C/C++ or Matlab has been investigated to increase design productivity and enable rapid design space exploration [13, 14, 15, 16, 17]. However, High Level Synthesis tools have not been so well adopted by the software engineer community because the design flow of the current commercial tools is more suited for the hardware rather than the software engineer. The designer is required to tune the application source code specifically for hardware design, and may have to intervene to specify low level details which may discourage most software engineers from using the technology. A successful high level synthesis tool targeting software engineers and parallel programmers will have to hide the architectural details from the programmer.

Using parallel programming models like OpenCL, to generate FPGA-based systems, open up system hardware design for the large community of software engineers to exploit the capabilities of high-end FPGA devices without the need for hardware expertise.

OpenCL programs express parallelism at its finest granularity. This is a particularly convenient feature for hardware generation, as the programmer explicitly exposes all available parallelism of the application. Exposing parallelism at its finest granularity allows hardware generation at different levels of granularity. Another favorable feature of OpenCL is the explicit expression of data movement in the form of buffer transfers between 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 communication, thus placing the burden of re-discovering parallelism and communication patterns to an optimizing compiler and/or the user – usually with limited success.

Our research develops algorithms, and architectures to generate automatically hardware accelerators from OpenCL kernels. Our synthesis tool, Silicon OpenCL (SOpenCL), generates a hardware accelerator from a single OpenCL kernel using two phases: OpenCL to C source to source transformation and C to RTL generation. Our research concerns the second phase. A C function generated by the source to source transformation consists of one or more nested loops that encapsulate the computationally intensive parts of the OpenCL kernel.

The contributions [18, 19, 20] of our research can be summarized as follows:

1. *Code Transformations:* The tool flow performs novel transformations specific for architectural synthesis. *Bitwidth analysis* transforms variable bitwidth from the standard size (char, int, etc.) into arbitrary sizes to minimize the amount of hardware resources. *Predication* replaces control dependencies with data dependencies, thus increasing the size of basic blocks and the potential of instruction schedulers to find an optimal instruction schedule. *Code slicing* decouples data movement from data computations, and overlaps their execution. A major transformation introduced in the tool flow is *Code Clustering*. SOpenCL analyzes patterns of instructions and produces application specific *macroinstructions*, where a *macroinstruction* consists of multiple basic arithmetic and logic operations. Macroinstructions provide a compact form of computation that can be implemented more efficiently than basic arithmetic and logic operations.

2. *Architectural Template:* SOpenCL utilizes an architectural template designed and configured to meet user performance requirements and fit the target device. The architecture of a hardware accelerator of an OpenCL kernel has a hierarchal structure which resembles the loop hierarchy in the generated C function. Each nested loop is allocated a single cluster of hardware which allows pipelining the nested loops execution. The architectural template

decouples and overlaps the execution of data computation and data movement by allocating separated modules for data computations (Datapath) and data movement to and from memories (Streaming Interface Unit).

3. *Concurrent Execution Model:* To exploit the separate hardware components in the architectural template, an asynchronous execution model is adopted. The operation of the streaming units and the computational datapaths is fully asynchronous, even across the boundaries of different loops and loop nests. Asynchronous execution model allows pipelined and parallel execution of multiple nested loops, and increases hardware utilization.

The current state of the tool produces a single accelerator per OpenCL kernel. The supported kernels may consist of arbitrary loop nests and shapes. They may contain synchronization and any kind of standard arithmetic operations. The tool flow also provides an IP library for floating point operators and math functions optimized to enhance the performance of the accelerator. OpenCL kernels that include dynamic memory allocation or function call are not supported.

1.3 Thesis Structure

The structure of the thesis is as follows:

Chapter 2 covers the background material necessary to understand the proposed algorithms and design techniques. More precisely, Chapter 2 presents the framework and infrastructure used by our tool flow..

Chapter 3 introduces the proposed architectural template for architectural synthesis. It describes the skeleton of the template, its basic structure and how an OpenCL kernel is mapped on the template components. The chapter addresses the architectural techniques used in handling synchronization and exploiting data reuse to reduce memory access overhead. The execution model of OpenCL kernel on the generated hardware accelerator is also discussed.

Chapter 4 describes the low level transformations/optimizations and hardware generation methods applied on the OpenCL kernel source code to provide architectural optimizations. Transformations include bitwidth optimization, predication, code slicing and instruction clustering. Code slicing separates portions of

7

code responsible for addresses generation from computations to decouple and overlap their execution. Instruction clustering generates application specific instructions to build custom functional units. Later in the chapter we introduce methods used in taking architectural synthesis decisions. More precisely, scheduling instructions on allocated resources, data caching configurations, and synchronization/interconnect data channels generation. Two scheduling algorithms are described: *modulo scheduling* and *as soon as possible* scheduling.

Chapter 5 presents the experimental evaluation of the proposed techniques and architectural template. Finally, Chapter 6 completes this dissertation with the presentation of the conclusions and reference to future work.

CHAPTER 2

SILICON-OPENCL TOOL FLOW

2.1   Tool Flow and Infrastructure

Silicon-OpenCL (SOpenCL) is an architectural synthesis CAD tool targeting heterogeneous parallel computing platforms (Figure 2.1). The objective is to allow a software programmer to develop an OpenCL application once, and deploy it on any platform, without the need for modifications. The tool consists of a two levels compilation process: High Level Compilation (HLC) and Low Level Compilation (LLC).

The high level compiler processes an OpenCL application and partitions its kernels as appropriate across the available computing platforms (CPU, GPU, and FPGA). The low level compiler processes OpenCL kernels selected to run on FPGA platforms. The task of the LLC is to compile an OpenCL kernel, and generate an equivalent hardware design that fits the target FPGA device and fulfills performance requirements. SOpenCL tool infrastructure also provides runtime environments for each of the target platforms to facilitate their integration and the execution of
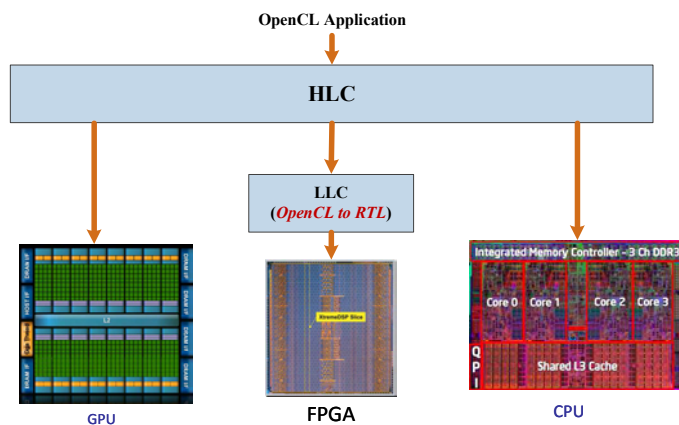
Figure 2.1: Silicon-OpenCL Tool Flow.

9

OpenCL kernels.

Figure 2.2 shows the low level compiler flow. The LLC converts unmodified OpenCL kernels into a system on chip (SoC) with hardware and software components. The tool flow generates a hardware accelerator for each OpenCL kernel in two phases: OpenCL-to-C transformation, and C-to-RTL. The tool flow also generates the runtime environment and drivers, in addition to the testbench generated for simulation and verification purposes. The OpenCL-to-C frontend developed by Daloukas [21] generates a C function from an OpenCL kernel by coarsening the computation granularity as will be detailed in section 2.3. The C-to-RTL backend developed in this thesis generates a hardware accelerator RTL description for each OpenCL kernel.

Figure 2.3 shows the C to RTL back end tool flow which-along with the front end is based on the LLVM compiler infrastructure. LLVM compiler translates the input C function into an assembly-like intermediate representation, called *LLVM-IR*. The LLVM compiler provides conventional optimizations and transformations such as dead code elimination, redundant code elimination, constants propagation, algebraic transformations, loop transformations, loop unroll, and loop invariant code motion. Given the LLVM-IR, the backend performs two sets of tasks, low level transformations and optimizations, and hardware allocation and generation.
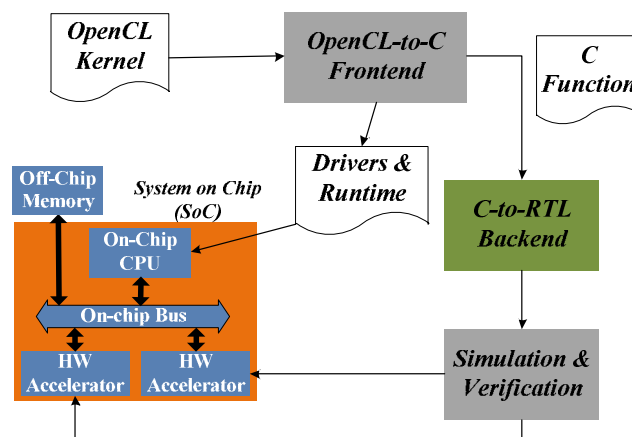
Figure 2.2: SOpenCL Low Level Compiler (SOpenCL-LLC). (*C-to-RTL backend is the result of this thesis research*).
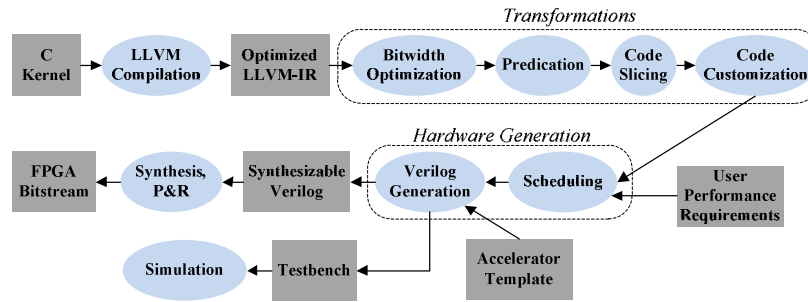
Figure 2.3: C-to-RTL backend.

2.2 OpenCL Programming Model

2.2.1 Overview

OpenCL [22] is a programming framework for heterogeneous computing platforms. OpenCL was initially developed by Apple Inc. as a portable programming framework for the vast number of multi-core CPUs and GPUs. Apple submitted an initial proposal in collaboration with technical teams at AMD, IBM, Intel, and Nvidia, to the Khronos group. Within six months Khronos group released the first OpenCL specification for the public. OpenCL programming language is based on ISO C99 with some limitations and extensions. The language is extended to provide explicit representation of parallelism, synchronization and memory regions.

OpenCL programming framework was designed with software portability in mind. The vision is to write a single application that can run on a variety of potentially heterogeneous platforms, from embedded systems to workstations and supercomputers. The OpenCL platform model comprises a host processor and a number of compute devices (Figure 2.4). Each device consists of a number of compute units, which are subsequently divided into a number of processing elements. An OpenCL application consists of a host program and a number of kernel functions. The host part executes on the host processor and submits commands that can refer either to execution of a kernel function or to manipulation of memory objects. A kernel function contains the computational part of an application and is executed on the compute devices.

11
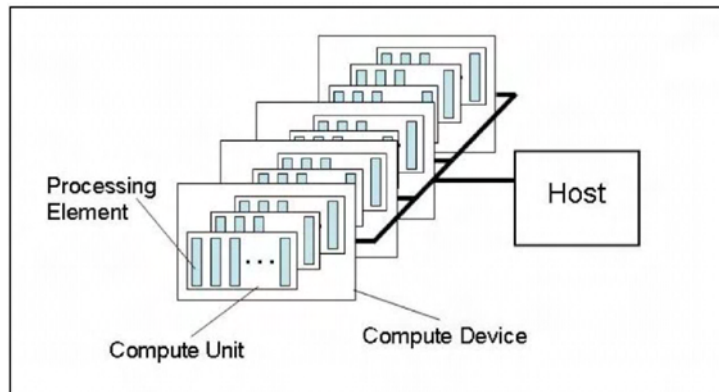
Figure 2.4: OpenCL Platform Model.

A key feature in OpenCL is that the compiler is built into the runtime system, which provides flexibility and portability, and allows OpenCL applications to select and use different compute devices in the system at runtime.

2.2.2 Computation Model

The work corresponding to a single invocation of an OpenCL kernel is called a work-item. Multiple work-items can be 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 $1 \leq N \leq 3$ (Figure 2.5). Once a command that refers to execution of a kernel function is submitted, the host part of the application defines an abstract index space, with a maximum of 3 dimensions of work groups and 3 dimensions of work items in each work group. A work-item is identified
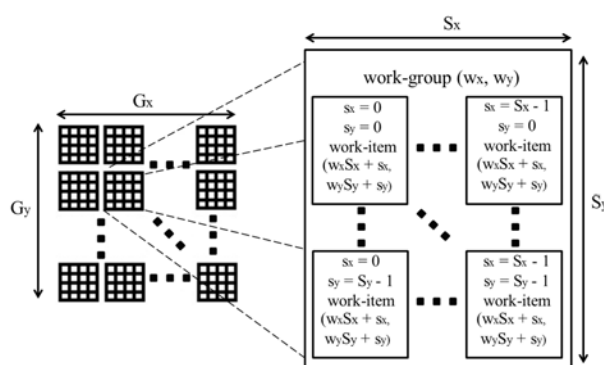
Figure 2.5: 2-dimensional computations grid geometry (N = 2).

12

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.

Figure 2.6 shows an example of Chroma interpolation OpenCL kernel. Chroma interpolation computes sub-pixels from chrominance components in a video frame. Each work item (one kernel invocation) computes one sub-pixel by applying a 4-tap filter on 4 chrominance pixels. The filter output is then clipped to the value range [0,255]. The kernel utilizes 2-dimentional computations grid like the one shown in Figure 2.5. The *get_global_id(0)* and *get_global_id(1)* runtime functions return the unique global *x*- and *y*-coordinates of the work-item, respectively.

OpenCL also provides runtime functions to return local work-item coordinates within a work group (Figure 2.7). For example, *get_local_id(0)* and *get_local_id(1)* return the *x*- and *y*-coordinates (*Sx* and *Sy* in Figure 2.5) of the work-item within the work-group.

The programmer explicitly defines the dimensions of a single work group when she invokes the kernel function. The number of work groups is determined implicitly in the runtime depending on the size of the computation problem. For example, the chroma interpolation kernel of Figure 2.6 has 2-dimensional work group of size 4×4, i.e. 16 work-items, where each work-item processes a single pixel. The number of work groups depends on the grid size, i.e. the video frame size. For 640×480 VGA frame, the grid includes 80×60 work-groups.

```
__kernel void ChromaInter(__global char * refF,
                          __global char * predF,
                          __local  char * Coeffs, int W)
{
  int id0 = get_global_id(0);
  int id1 = get_global_id(1);

  int tmp = (refF[id1*W + id0]         * Coeffs[0] +
             refF[id1*W + id0 + 1]      * Coeffs[1] +
             refF[(id1 + 1)*W + id0]    * Coeffs[2] +
             refF[(id1 + 1)*W + id0 + 1] * Coeffs[3]) >> 6;

  predF[id1*W + id0] = Clipping( tmp );
}
```

Figure 2.6: Chroma Interpolation OpenCL kernel

13

```
__kernel void MatrixMul(__global float* a, __global float* b,
                        __global float* c)
{
 int row = get_global_id(1);
 int col = get_global_id(0);

 __local float aTile[HEIGHT][WIDTH], bTile[HEIGHT][WIDTH];

 int y = get_local_id(1);
 int x = get_local_id(0);
 aTile[y][x] = a[row*WIDTH + col];
 bTile[y][x] = b[row*WIDTH + col];

 barrier(CLK_LOCAL_MEM_FENCE);

 int sum = 0, j;
 for(j = 0; j < WIDTH; j++)
     sum += aTile[x][j] * bTile[j][y];

 c[row*WIDTH + col] = sum;
}
```

Figure 2.7: Matrix Multiplication OpenCL kernel example.

2.2.3 Synchronization

OpenCL uses what is called a *relaxed memory consistency model* which means that different work-items may see a different view of global memory as the computation progresses. Synchronization is required to ensure data consistency within the work items of a work group, while reads and writes to all memory spaces are consistently ordered within work-items.

OpenCL programming model provides two types of synchronization functions among work-items inside a work-group, memory-fence and barrier function. A barrier function requires all work-items inside a work-group to rendevouz at the *barrier* call. In other words, every work-item in the same work group must execute the barrier function before any work-item is allowed to continue execution beyond the barrier command. A memory-fence only requires that loads and stores preceding the *mem_fence* all be committed to memory. On the other hand, there is no synchronization mechanism among work-groups, which means that work-groups can be executed in parallel.

Figure 2.7, depicts an OpenCL kernel for naive matrix multiplication. Each work-item first prefetches an entry from each matrix and stores it in local memory. After the *barrier* function, each work item computes an entry in the output matrix. The *barrier (CLK_LOCAL_MEM_FENCE)* function stalls the execution of every work-

14

item in the work group before allowing any work-item to execute the last statements in the kernel. The barrier synchronization here is necessary to enforce memory dependencies between work-items in the same work group; loaded matrices entries by each work item are used by the rest of work items to perform their computations.

2.2.4   OpenCL Memory Structure

OpenCL defines a memory hierarchy of four types: *global memory, constant memory, local memory,* and *private memory* (Figure 2.8). OpenCL standard only specifies the access level of different type of memory. Programmers can use memory region address qualifiers; **__global, __constant, __local**, and **__private** to specify the type of memory hosting data as in Figure 2.6 and Figure 2.7.

Global memory has the largest size on a compute device. Global memory is visible to all work-items in the computations grid. While the largest and visible to all work-items, global memory is considered the slowest memory. *Constant memory* is a read-only section of the global memory visible to all work-items. Constant memory can be associated with specialized hardware optimizations to broadcast data. *Local memory* is much faster than global memory, and is typically located on-chip. A local memory is a shared section of memory within the work-items of the same work-group. Synchronization of memory accesses in the local memory is the responsibility of the programmer. A *private memory* is used within a work-item, and implemented generally using registers in a GPU or CPU core. A private memory is fast and can be
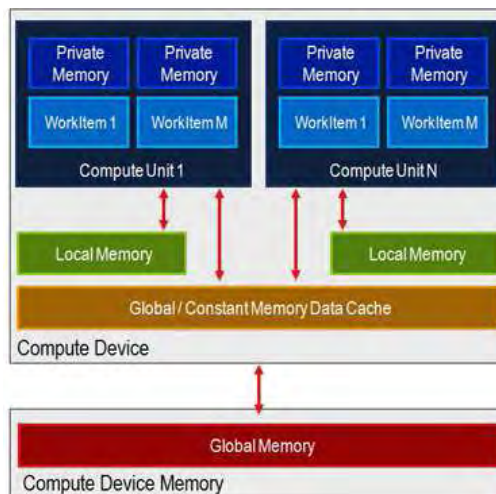
Figure 2.8: OpenCL memory hierarchy.

15

used without the need for synchronization primitives. In situations where the compute device has inadequate number of registers, variables stored in private memory are spilled to global memory space causing significant performance drop.

2.3   OpenCL to C transformation

As explained in the previous section, OpenCL exposes parallelism at a fine level of granularity by allowing the programmer to embody the task executed by a single logical thread in an OpenCL *kernel*. For example, the OpenCL code for chroma interpolation (shown in Figure 2.6) describes the computation of a single loop iteration which comprises an OpenCL work-item in this case. Depending on performance requirements, and resource availability, any number of hardware accelerators can be generated spanning from a simple interpolator, executing a single thread per invocation, to an accelerator that produces the complete interpolated frame every time it is invoked. Between these two extremes, a hardware generation tool can generate any number of accelerators, each, potentially, being assigned a different amount of workload per invocation.

In order to enable efficient mapping of OpenCL kernel functions to the underlying platform while at the same time taking into account any hardware constraint, SOpenCL tool applies a series of source-to-source transformations in the high level compiler frontend (Figure 2.2) that collectively aim at coarsening the granularity of a kernel function from the work-item to the work-group level.

Daloukas [21] explains that the selection of a work-group as the preferred degree of granularity for logical threads serialization may seem arbitrary. However, taking synchronization within a work group into account, it will become evident that other options may present hard to overcome complications in the presence of synchronization operations or multiple exit points within the kernel. At the same time, work-group granularity is usually explicitly set by OpenCL programmers, often considering data reuse, or matching the work-group data footprint to the capacity of specific levels of the memory hierarchy. Therefore, introducing different degrees of work granularity at the runtime, despite being semantically correct, might introduce performance side-effects.

16

OpenCL-to-C frontend applies three source-to-source transformations: threads serialization, elimination of synchronization functions, and variable privatization, each one explained in the remainder of the chapter.

2.3.1 Logical Threads Serialization

The main step in the OpenCL-to-C frontend 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, execution of work-items is serialized 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.

Threads serialization of kernel *Add_3D* (Figure 2.9*a*) produces the C function in Figure 2.9b. Input argument *local_size_array* is an array of size 3, and is used to store the dimensions of the work group to be used as boundaries in the triple nested loop.

2.3.2 Loop Fission

Thread serialization can lead to invalid execution of a kernel function if the OpenCL kernel body contains synchronization operations. In the presence of a barrier instruction, every work-item must execute that instruction before any work-item is

```
__kernel void Add_3D(__global int * A,
                     __global int * B,
                     __global int * C,
                     int W, int H)
{
  int id0 = get_global_id(0);
  int id1 = get_global_id(1);
  int id2 = get_global_id(2);

  int pos = id2*W*H + id1*W + id0;

  C[pos] = A[pos] + B[pos];
}
```
(*a*)

```
__kernel void Add_3D(__global int * A,
                     __global int * B,
                     __global int * C,
                     int * local_size_array,
                     int W, int H)
{                                    triple nested loop
  int i2, i1, i0;
  for( i2 = 0; i2 < local_size_array[2]; i2++){
    for( i1 = 0; i1 < local_size_array[1]; i1++){
      for( i0 = 0; i0 < local_size_array[0]; i0++){
        int pos = i2*W*H + i1*W + i0;

        C[pos] = A[pos] + B[pos];
      }
    }
  }
}
```
(b)

Figure 2.9: Logical Threads Serialization. (*a*) Add_3D OpenCL kernel adds two 3D arrays. The three runtime functions return the coordinates *(id0, id1, id2)* of the pixel computed by a work-item. (b) C function after threads serialization.

17

allowed to continue its execution. However, in the modified C 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 *loop fission* transformation that facilitates 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, Figure 2.10 depicts this transformation for the *MatrixMul* kernel of Figure 2.7. Since there is one synchronization statement, *barrier*, two triple nested loops are required to ensure correct execution of the C kernel function.

A similar problem occurs 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 2.11). For example, in Figure 2.11b, the *if*-statement works as a synchronization barrier. Hence, triple nested loops (*loops*) are created around each statement (S_1 and S_2).

2.3.3   Variable Privatization

Loop fission presents a complication for variables that are defined in one triple-

```
__kernel void MatrixMul(__global float* a, __global float* b,
                        __global float* c, int * global_id)
{
 int row, col, sum, j;

 __local float aTile[HEIGHT][WIDTH], bTile[HEIGHT][WIDTH];

 triple_nested_loop {
   row = global_id[1] + i1;
   col = global_id[0] + i0;
   aTile[i1][i0] = a[row*WIDTH + col];
   bTile[i1][i0] = b[row*WIDTH + col];
 }

 // barrier(CLK_LOCAL_MEM_FENCE);

 triple_nested_loop {
   row = global_id[1] + i1;
   col = global_id[0] + i0;
   sum = 0;
   for(j = 0; j < WIDTH; j++)
       sum += aTile[i0][j] * bTile[j][i1];

   c[row*WIDTH + col] = sum;
 }
}
```

Figure 2.10: Loop Fission example.
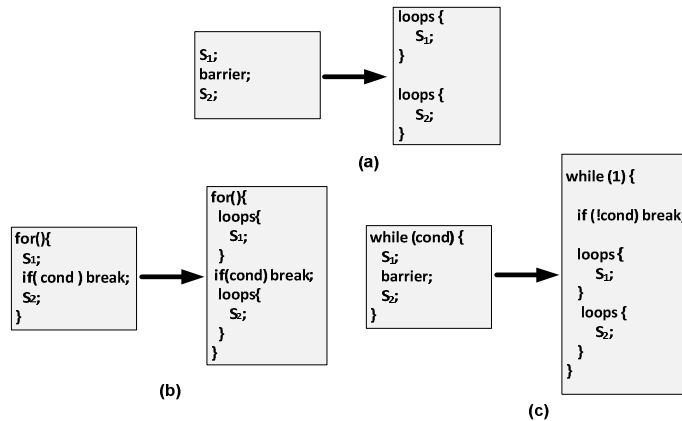
Figure 2.11: Barrier Elimination examples.

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.

SOpenCL compilation infrastructure conducts a live-variable analysis to identify the variables that are live beyond the boundaries of the loops introduced by loop fission. Next, 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.

Figure 2.12 shows an example of loop privatization. In Figure 2.12b, the variable k computed by each work-item (i.e. loop iteration) in the first nested loop, will be overwritten by other work-items (loop iterations). When the k variable is used in the second nested loop its value has been polluted with the last iteration of the first nested loop. Figure 2.12c shows the result of applying variable privatization on loop fission
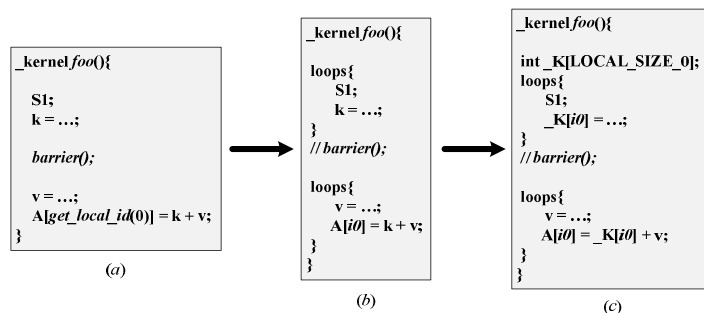
Figure 2.12: Variable privatization example. (a) Original OpenCL kernel. (b) Loop fission output (wrong). (c) Variable privatization output (correct).

19

output. A local memory array (_K) is allocated with size equal to the number of work-items per work group (*LOCAL_SIZE_0*). Each work-item stores its *k* value in the allocated array at a unique position to be later used in the second nested loop.

For further details on the OpenCL compiler transformations, the interested reader should consult [21].

2.3.4 Output C function structure

Figure 2.13*a* depicts an OpenCL kernel which implements LU Decomposition is used as a running example to explain the sequence of steps to generate the hardware accelerator. This kernel is part of the Rodinia benchmark suite [23].

LU Decomposition kernel consists of three parts, separated by barrier instructions. All work-items that execute the first part of the code, prefetch 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 writeback to array *m*.

Figure 2.13b 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).

One may assume that transforming the parallel OpenCL representation into the sequential C representation, we lose the desirable features of OpenCL language, i.e. explicit parallelism and data movement. However, the specific structure of the generated C functions and the knowledge of what each portion of the function represents, we can ensure that the desirable features of OpenCL are preserved. Multiple nested loops in the C function indicate the existence of synchronization commands within the OpenCL kernel. Multiple nested loops have to be executed sequentially, but their execution can be pipelined.

The body of a triple nested loop represents the workload of a single work-item, which leads to the conclusion that multiple iterations of a triple nested loop can correspond to multiple work-items, and hence, can be executed in parallel and out of order.

```
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) {                                          B_0_0
        idx    = loc_x ;  i0 = 0; i1 = 8;
        arroff = arroff0 = offset * m_d + offset;
        arroff1 = arroff + (b_x + 1)<<4 + idx;
    }
    else{                                                      B_0_1
        // similar settings
    }
    arr_offset = arroff0;
    for (i = i0; i < i1; i++) {                                L_0_2
        dia[i][idx] = m[ array_offset + idx];
        array_offset += m_d;
    }
    array_offset = arroff1;
    for (i = 0; i < 16; i++) {                                 L_0_3
        if ( loc_x < 16)  peri_row[i][idx] = m[ array_offset+idx ];
        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)                                   L_1_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) {                                         B_2_0
        array_offset = (offset + 1) * m_d + offset;
        for (i = 1; i < 16; i++) {                             L_2_0
            m[array_offset + (b_x + 1) * 16 + idx] = peri_row[i][idx];
            array_offset +=m_d;
        }
    }
    else {                                                     B_2_1
        array_offset = (offset + (b_x + 1) <<4) *  m_d + offset;
        for (i = 0; i < 16; i++) {                             L_2_1
            m[ array_offset + idx] = peri_col[i][idx];
            array_offset  += m_d;
        }
    }
}
```
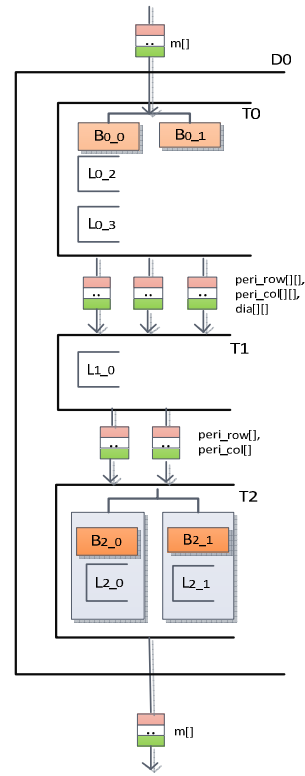
(*a*)   (b)

Figure 2.13: OpenCL kernel for LU Decomposition with marked loops (L_{i_j}) and basic blocks out of loops (B_{i_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.

Explicit local memory representations are transformed into local data arrays in the C function, and can be implemented as on-chip distributed memory blocks.

2.4   LLVM Compiler Infrastructure

LLVM compiler infrastructure [24] has been developed to provide a machine independent framework for program optimization, analysis, and refactoring. To provide support for multiple programming languages and different target architectures, LLVM adapts a three-step compilation flow (Figure 2.14). The LLVM

21
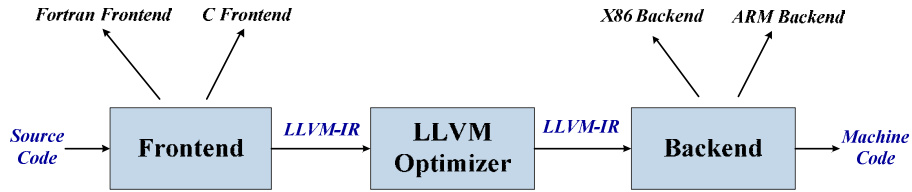
Figure 2.14: LLVM compiler Infrastructure

compiler model provides a RISC-style, yet rich, intermediate representation (LLVM-IR) between the frontend, optimizer, and backend.

The clarity and completeness of the LLVM-IR, provides a simple way for conveying information between multiple analysis and transformation passes as well between the frontend and backend. Using LLVM-IR, the compiler framework is a collection of libraries of transformations and optimizations can be used to build a compiler for any language and target architecture. In particular, LLVM-IR is both well specified and the *only* interface to the optimizer. This property means that all you need to know to write a frontend for LLVM is what LLVM-IR is, how it works, and the invariants it expects.

2.4.1   LLVM Intermediate Representation (LLVM-IR)

The LLVM-IR instruction set captures the key operations of ordinary processors but avoids machine-specific constraints such as physical registers, pipeline architecture, and low-level calling conventions. LLVM-IR provides an infinite set of typed virtual registers which can hold values of primitive types (boolean integer, floating point, and pointer). The virtual registers are in Static Single Assignment (SSA) form [58]. LLVM-IR is a load/store architecture: programs transfer values between registers and memory solely via load and store operations using typed pointers.

LLVM-IR uses SSA as its primary code representation (Figure 2.15). SSA is an Intermediate Representation (IR) used in several compilers (including LLVM compiler). In SSA each instruction is assigned a unique register name and each use of a register is dominated by its definition. In the example of Figure 4.4, the two assignments for the register *x* is transferred into two assignments on two different registers.

22

```
bb1:                    bb2:
   x = add a, b            x = mul a, 3


              bb2:
                 y = mul x, x
```
(*a*)

```
bb1:                    bb2:
   x1 = add a, b           x2 = mul a, 3


              bb3:
                 j = Phi(x1, bb1), (x2, bb2)
                 y = mul j, j
```
(*b*)
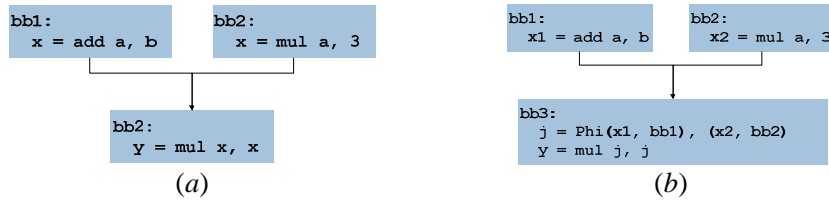
Figure 2.15: SSA Representation (a) Code portion without SSA representation. (b) Code with SSA representation.

A key feature of SSA IR is the *Phi* instruction which selects the proper value to pass to the next blocks, depending on the last control transfer event. In the example given, the assignment to register *y* does not read the values of the assignments for *x1* and *x2* directly but, instead, the proper value is passed through the *Phi* instruction assignment.

Memory locations in LLVM-IR are not in SSA form because many possible locations may be modified at a single store through a pointer, making it difficult to construct a reasonably compact, explicit SSA code representation for such locations.

SSA form provides a compact *def-use* graph that simplifies many dataflow optimizations and enables fast, flow-insensitive algorithms to achieve many of the benefits of flow-sensitive algorithms without expensive dataflow analysis. Non-loop transformations in SSA form are further simplified because they do not encounter anti- or output dependences on SSA registers. Non-memory transformations are also greatly simplified because registers cannot have aliases.

Figure 2.16 shows an example of LLVM-IR generated for a C function that clips a value in the range [0, 255]. A function in LLVM-IR consists of one or more basic


```
char ClipValue(short a) {

  char b = a;

  if (a < 0) b = 0;
  else if (a > 255) b = 255;

  return b;
}
```
Input C program

```
define i8 @ClipValue(i16 %a) {
entry:
  %tmp1 = cmp lt i16 %a, 0;          ← 16-bit integer data type
  br i1 %tmp1, label %exit, label %else

else:
  %tmp2 = cmp gt i16 %a, 255;
  br i1 %tmp1, label %exit, label %else2
                                   Instructions Basic Block
else2:
  br label %exit

exit:
  %tmp3 = phi [0, entry], [255, else], [%a, else2];
  ret i8 %tmp3
}
```
Output LLVM-IR

Figure 2.16: LLVM-IR Example.

23

blocks of instructions. A program in LLVM is represented as a module of code that includes one or more functions. A feature in LLVM-IR is its arbitrary bitwidth data type representation, which is convenient for hardware bitwidth optimizations.

2.5 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 majority of research efforts used a variation of C as their input programming language which was driven mainly by the existence of a large body of C programmers, and the extensive use of C in embedded applications. C-based architectural synthesis research can be classified into two categories: using a restricted format of C written in specific way, or extending extra language constructs and syntax to support hardware synthesis.

PICO-NPA [13], SPARK [25], Trident [26], and Streamroller [27] belong to first category. PICO-NPA is a synthesis system that generates non-programmable accelerators from a C function. PICO restricts a C function to consist only of a single perfectly nested loop. In addition to nested loops, PICO make use of C pragmas to pass application specific information to simplify program analysis. Those pragmas allow the user to declare no-standard data widths, to indicate that specific global variables are not live-in or not live-out. Also pragmas could be used to advise the compiler to create local memory for certain arrays, like lookup tables. PICO does not support recursion, and dynamic memory allocation.

SPARK and Trident impose no stylizations or modeling on the input C functions. The only restrictions in SPARK C model include function recursion and dynamic memory allocation. Trident imposes additional restrictions: the code cannot contain print statements, function arguments or returned values, calls to functions with variable-length argument lists, or arrays without a declared size.

Streamroller emulates the stream programming model by some extensions of the C language to capture parallelism and decouple communication from computation. The system takes as input the application written in C, expressed as a set of communicating kernels. The input program consists of two logical parts, a set of kernel specifications and system specification. A kernel is expressed as a single C

function. All inputs and outputs to the kernel have to be provided as arguments to the function. The body of the kernel has to be perfectly nested for loops. The system specification describes one "packet" forward flow through the pipeline. The system specification is expressed as a C function whose body contains a sequence of calls to the kernel functions.

The second category of C-based hardware synthesis research includes work that created new programming languages as variations of ANSI-C, such as Handel-C [28], Mitrion-C [29], haydn-C [30], and SA-C [31, 32] Handel-C retains most of the pure C syntax and sequential execution model. However, to support compilation for hardware, Handel-C supports several hardware implementation features like arbitrary bitwidth declarations of variables. Parallelism in Handel-C is supported through a "para" qualifier to declare a block of statements that will run in parallel. Handel-C provides a channel declaration to communicate between parallel blocks. RAMs and ROMs are declared in Handel-C like arrays, with exception that RAMs and ROMs are accessed once each clock cycle.

Haydn-C has many similarities to Handel-C. Like Handel-C, it uses parallel blocks of statements, VHDL-like components/entities to describe parallelism in the program. The Handel-C and Haydn-C are timed languages, i.e. require from the programmer to keep exact timing of the program execution, by defining the time execution of each expression as one clock cycle, and providing the user with a "delay" construct to control the timing of execution.

Mitrion-C main concept centers on parallelism and data dependencies and there is no order-of-execution; any operation may be executed as soon as its data-dependencies are fulfilled. To capture the custom features of hardware implementation, Mitrion-C enables the user to specify the exact variable precision by declaring the bit-width of the variable. Like other static single assignment languages, each statement in Mitrion-C is an expression, statements like FOR, WHILE loops return values, and each variable within a scope is assigned once. The single-assignment is required in Mitrion-C since statements within scope could run in parallel rather than sequential. In addition, since Mitrion-C targets FPGAs, it supports the use of RAM blocks and banks through a group of memory read/write functions.

25

SA-C differs from C in some important ways. It is an *expression-oriented*, functional language. Its scalar types include signed and unsigned integers and fixed point numbers with specified bit widths. It has no explicit pointers, and is non-recursive. It has true multidimensional arrays, including array sections similar to those in Fortran 90. It also allows any function, loop or conditional expression to return multiple values.

Other prior research based on C programming model chose to provide libraries of functions and types to support hardware synthesis instead of creating a new language. Stream-C [33] is a combination of annotations and library functions callable from C program. There are three distinguished objects declared in Stream-C program: process, stream and signal. Stream and signal carry data and control bits between processes. Processes are the computation kernels that implemented by hardware or host processor. Process declaration consists of head where the name and IN/OUT streams/signals are declared, and body encloses the computational operations. The body is written using callable functions and a subset of supported C.

Impulse CoDeveloper is an ANSI C synthesizer [34] based on the ImpulseC language. ImpulseC is distinct from standard C in that it provides a parallel streaming programming model for mixed processor and FPGA platforms. For this purpose, Impulse C includes extensions to C, in the form of functions and datatypes, allowing applications written in standard C to be mapped onto coarse-grained parallel architectures that may include standard processors along with programmable FPGA hardware. Using ImpulseC, an application could be described as a collection of parallel, pipelined processes, each of which has been described using one or more C subroutines.

At the heart of the ImpulseC streaming programming model are processes and streams. Processes are independently synchronized, concurrently executing segments of an application. Hardware processes are written using a subset of standard C and perform the work of an application by accepting data, performing computations and generating outputs. In a typical application, data flows from process to process by means of buffered streams, or in some cases by means of messages and/or shared memories. The characteristics of each stream, including the width and depth of the generated FIFOs, may be specified in the C application.

26

Another category of research efforts used the stream-programming model as their high level languages. In Proteus [35], a program consists of two objects: streams descriptors and stream data-flow graph (sDFG). A stream descriptor declares stream access patterns from main memory. The sDFG describes a computational kernel, and declares IN/OUT streams. Using those two objects a program can be written as a set of communicating sDFG blocks through streaming channels.

Optimus [36] takes programs written in StreamIt stream programming language. Programs in StreamIt are represented as graphs where nodes, called *filters* encapsulate computation, and edges represent FIFO communication. StreamIt is based on the synchronous dataflow (SDF) model of computation [50]. Each filter consists of a *work* function that repeatedly executes when sufficient data is available on its input FIFO (queue). The work function reads data from its input queue using *pop* operations, and writes data to its output queue using *push* operations. The work function can also inspect input without removing them from the FIFO using a *peek* operation.

Prior research has investigated the use of different programming models like MATLAB and Simulink. MATLAB and especially Simulink have traditionally been used for algorithm design. The availability of a mature tool with specialized modules (toolboxes, blocksets) along with the possibility of integrating C code makes the tool a very attractive development platform. Work in [16] presents a MATLAB-to- RTL compilation flow. One of the issues to be resolved in generating hardware from MATLAB is to figure out the type/shape of the variables since MATLAB variables have no notion of type or shape. To generate hardware, the compiler must determine the exact data type i.e. integer or floating point, or complex numbers etc. The compiler also needs to determine the shape i.e. how many dimensions the matrix (array) has, and what are the extents in each dimension.

The majority of current high level synthesis commercial tools use SystemC as input representation [14, 37, 38]. SystemC is a set of C++ classes and macros used to simulate concurrent processes, each described using plain C++ syntax. SystemC is closer to HDL languages VHDL and Verilog. A program in SystemC usually consists of several modules which communicate via ports. SystemC Modules include concurrent processes as the main computation elements. Modules communicate via

channels, which could be either wires or complex communication mechanisms like FIFOs or bus channels. SystemC libraries provide datatypes extensions like arbitrary bitwidth integer datatypes, and fixed point datatypes, in addition to C++ standard types.

Lately, research in architectural synthesis have focused on parallel programming languages such as FCUDA, a tool that converts CUDA kernels to synthesizable hardware [39]. CUDA is a parallel programming model developed by Nvidia for graphics processing. A CUDA kernel implicitly describes multiple CUDA threads that are organized in groups called thread-blocks. Thread-blocks are further organized into a *grid* structure similar to that of OpenCL. FCUDA is based on source-to-source transformation that generates a C function for each CUDA kernel. The generated C code is annotated with pre-processor directives (FCUDA pragmas) inserted by the FPGA programmer into the CUDA kernel. These directives control the FCUDA translation of the expressed parallelism in CUDA code into explicitly-expressed coarse-grained parallelism in the generated AutoPilot code. The FCUDA pragmas describe various FPGA implementation dimensions which include the number, type and granularity of tasks, the type of task synchronization and scheduling, and the data storage within on and off-chip memories.

The AutoPilot Compiler [15] 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 unrolls inner-loops to run concurrently when no across iterations dependencies are detected. AutoPilot allocates all arrays onto local BRAMs. It also supports arbitrary bitwidth data types to achieve optimized hardware implementations.

Jääskeläinen et al. [40] introduce a compilation infrastructure based on LLVM to generate transport-triggered architectures from OpenCL codes in an approach seemingly similar to our work. The 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

28

extensions, but perform extensive compile-time analysis instead, and granularity coarsening in order to avoid putting additional burden to the programmers.

Altera Inc. started an initiative to build FPGA-based systems from OpenCL programs [41]. The concept of Altera's OpenCL-to-FPGA is similar to that of Jääskeläinen et al.; OpenCL threads are mapped on customized processing cores. The system is populated with many of the processing cores on which the entire computations grid is mapped. An embedded on-chip RISC processor (e.g. Nios) plays the role of host processor that manages OpenCL threads. The processing cores are either custom pipelines or a VLIW/Vector processor.

Finally, OpenRCL platform utilizes OpenCL to schedule fine-grain parallel threads to a large number of MIPS-like cores [42]. OpenRCL does not generate customized hardware accelerators, although each MIPS core can be configured to match application characteristics.

29

CHAPTER 3

ARCHITECTURAL TEMPLATE

3.1   Overview

In a conventional hardware design flow, application functionality and structure determine the target design architecture. A hardware designer performs firstly a thorough analysis of the application functionality to extract parallelism and data communication patterns. Based on the analysis output, the designer partitions the application into a hierarchal structure of parallel tasks and subtasks each implemented separately, and determines the communication network connecting the set of tasks. Hardware designers exploit all kinds of available parallelism in the application like instruction parallelism, data parallelism, pipeline parallelism, and task level parallelism.. Moreover, each task implementation is optimized according to its specific computational patterns.

Figure 3.1 depicts the block diagram of a manual implementation of the motion compensation block in AVS video codec system [43]. A hardware designer typically partitions a complex task into multiple subtasks each performing a specific function: *chroma interpolation*, and *luma interpolation* (Figure 3.1*a*). Such partitioning exploits task level parallelism by concurrently executing chroma and luma interpolation, and pipeline parallelism by overlapping the execution of multiple blocks of data (called macroblocks in the context of video codecs). The designer may go further by partitioning each subtask into smaller blocks each performing a specific functionality exploiting more task parallelism, pipeline parallelism and data parallelism (Figure 3.1b). At the low level partitions, a hardware designer will exploit computation patterns to build efficient circuits to perform the basic computations (Figure 3.1c). Hardware designers traditionally design separated components for data streaming and interfacing to overlap I/O data communication and computations.

30
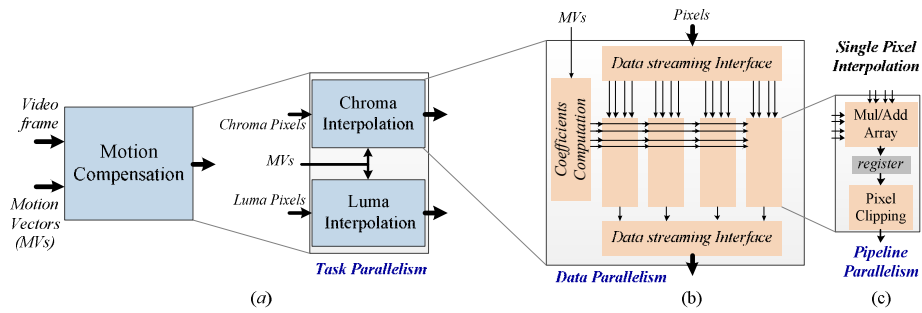
Figure 3.1: Motion Compensation Block Manual design.

In this work, the SOpenCL backend transforms a C function, corresponding to an OpenCL kernel, 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. In the following sections we will describe the structure and components of the architectural template, and how the C function is mapped onto it.

3.2   High Level Architecture

3.2.1   Hierarchical Structure

The use of an architectural template is necessary to relieve the programmer from specifying the tasks partitions and mapping by providing a systematic approach in partitioning and mapping the kernel code onto the hardware fabric while exploiting available parallelism. The proposed architectural template has a hierarchal structure that closely follows the computational hierarchy of the input kernel. Figure 3.2b shows the architecture of the hardware accelerator of the LU Decomposition kernel shown in Figure 3.2a. The architectural template is built mainly of two types of components: Processing Element (PE) and Control Element (CE). A PE is a customized architecture that executes an inner-most loop. A CE implements the functionality of the outer loops and loop invariant statements. Based on this classification, the kernel in Figure 3.2a translates into the accelerator of Figure 3.2b as follows:

- *Inner Loops*: Each of the inner loops {L_{0_2}, L_{0_3}, L_{1_0}, L_{2_0}, and L_{2_1}} is allocated a PE module.

31

- *Nested Loops*: Each of the nested loops {*T0*, *T1*, and *T2*} is allocated a CE module {*CE0*, *CE1*, and *CE2*}. Moreover, CE modules CE0 and CE2 are also used for processing outer loop basic blocks {B_{0_0}, B_{0_1}} and {B_{2_0}, B_{2_1}}, respectively.

- *Loop Invariant Code*: Loop invariant code outside any nested loops in the kernel body is allocated a CE module {*CE_g*}.

In this hierarchal structure a *parent-child* relationship exists between a CE module and another CE or PE module. In addition to executing outer loops and loop invariant code, a parent CE initiates the execution of its children. For instance, module CE0 is responsible for controlling execution of PE modules $PE(L_{0\_2})$ and $PE(L_{0\_3})$.

Local arrays in the kernel (*peri_row*, *peri_col*, and *dia* in Figure 3.2a) are each allocated a local memory implemented using dual port Block RAMs (BRAMs). Local memories could be either double buffered or work as a FIFO to enable pipeline parallelism of multiple PE and CE modules.

The architectural template allocates arbiters to manage data read and write

(a)   (b)
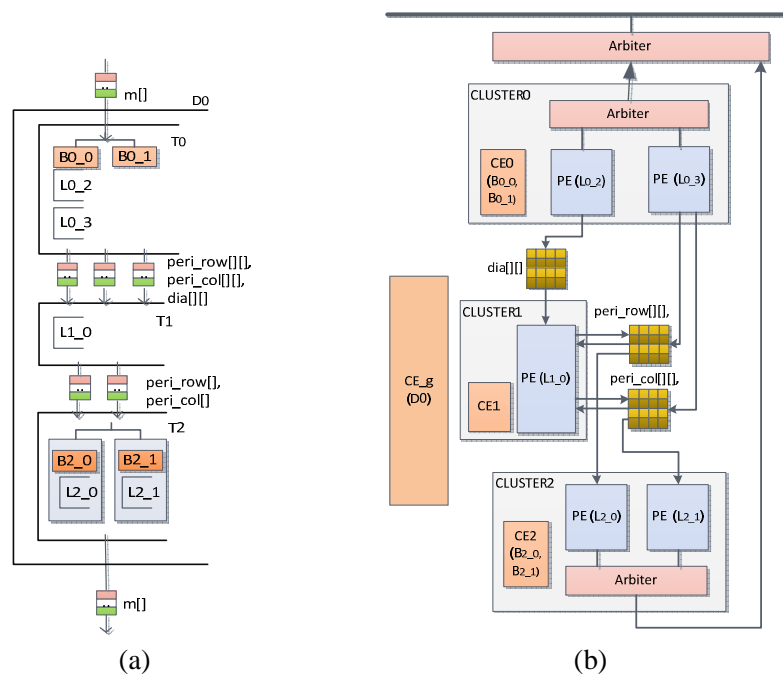
Figure 3.2: (a) Program structure of LU Decomposition kernel after coarsening the granularity to the equivalent of a work-group. (b) The block diagram of the automatically generated hardware accelerator for LU decomposition.

32

requests to global memories. Each separate memory port has its own arbitration logic. Multiple PE and CE modules that access the same global memory will compete to gain access to a global memory port.

The resemblance between the source code structure and the generated architecture provides several benefits:

- *Exploiting multi-level parallelism*: Multiple nested loops can be pipelined and hence execute in parallel. Multiple PE modules are allowed executing in parallel if they are independent or can be pipelined if they have cross iteration dependencies.

- *Full Customization*: An architecture that resembles the hierarchal structure of the kernel code captures every feature and characteristic of the code much better than a random RTL structure or a microprocessor-like architecture. Separate datapaths built to execute computations in different loops are designed more specifically to match the computational pattern of each loop, instead of having more generalized datapath for multiple loops.

- *Control distribution*: Control signal delay and logic becomes more critical when it covers large hardware blocks. Building architecture with multiple hardware blocks each executing independently and using a hand-shaking synchronization mechanism will localize control logic and reduce significantly the distance a control signal needs to travel within a single clock cycle.

3.2.2 Interconnection network

The interconnection network connecting all components uses FIFO channels between two components (PE or CE) that exchange data. The use of FIFO channels allows asynchronous execution and overlaps the execution of loop iterations, as will be described in Section 3.5.

Figure 3.3 depicts two types of data channels; scalar data point-to-point FIFO channel, and local streams buffer. Scalar FIFO channels are implemented using Flip Flops, and local stream buffers are implemented using FPGA Block RAMs (BRAMs). Multiple scalar FIFO channels are allocated for the same scalar variable if it has multiple consumers (Figure 3.3a). On the other hand, a local stream is allocated

33

only one local buffer channel shared by all producers/consumers of the same local stream (Figure 3.3b).

A scalar FIFO channel transfers scalar variables in the C function between producer and consumer components (PE, and CE modules). A producer continues to write data as long as all FIFO channels have free space (*full* signal equal to 0), and a consumer absorbs data as long as the FIFO is not empty. A FIFO channel will store incoming data if the *valid* signal is true, and will output data to the producer if the *absorb* signal is true. A consumer absorbs data tokens from the FIFO by setting the *absorb* signal to 1 (i.e. true), e.g. *consumer_0* sets *absorb_0* signal equal to 1 to absorb data from its own FIFO channel. The FIFO channel flushes one data token each clock cycle if the input *absorb* signal is true. Hence, if a consumer wants to read one data token from its FIFO channel, the *absorb* signal should stay true (equals 1) only for one clock cycle. The FIFO channel sets the *full* signal to 1 if there is no more space to store incoming data tokens (i.e. the FIFO is full), and forces the producer to stop generating new data tokens.

A local buffer channel is created for each data array which is local to a kernel. A local buffer channel is built using dual port Block RAMs providing separate Read/Write ports. A local buffer address space can be partitioned into two or more blocks (In Figure 3.3b local buffer has two blocks) to enable double buffering and
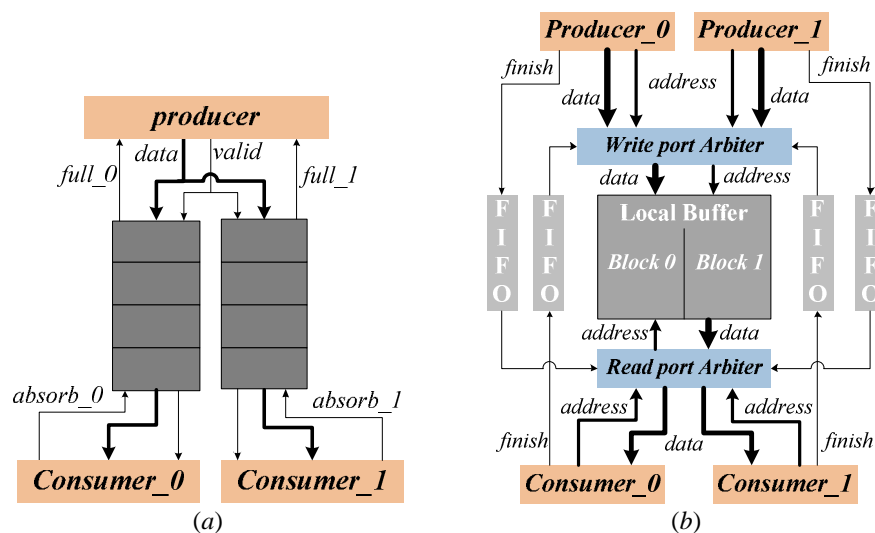
Figure 3.3: Interconnect communication channels. (a) FIFO channels. (b) Local Buffers channels. Local buffer two blocks are used for double buffering.

34

pipelined execution. In fact, a local buffer partitioned into multiple blocks is implemented as shared FIFO between the multiple consumers and producers. A producer first writes into *Block_0* address space of the local buffer, and when it finishes it sets its output *finish* signal to 1 so a consumer could start reading data from *Block_0*. While a consumer reading data from *Block_0*, the producer starts a new write session to *Block_1*, then it sets the *finish* signal again to declare finishing the second write session and starts a third write session to *Block_0* as soon as the consumers finished reading from *Block_0*. This switch forth and back between *Block_0* and *Block_1* allows overlapping read and write sessions to local buffers.

A producer/consumer generates a *finish* signal stored in a FIFO and used by buffer arbiters to enable/disable successors read/write requests. A *finish* signal becomes true once a consumer/producer submits as many read/write requests that fulfill its dependencies. For example, $PE(L_{0\_2})$ in Figure 3.2b, generates a true *finish* signal when the execution of the last iteration of loop T_0 terminates (and the write operation into *dia* local array). Likewise, consumer $PE(L_{1\_0})$ produces true *finish* signal when the execution of the last iteration of loop T_1 terminates. A *finish* signal is stored in a FIFO channel when it is equal to 1. A *read port arbiter* examines all FIFOs *finish* signals and allows a consumer to start reading data only when all its dependencies are fulfilled, i.e. all its predecessors produced a true *finish* signal. When a consumer finishes its reading session, the *read port arbiter* flushes the corresponding finish FIFOs of all its predecessors. The same operation also performed by the *write port arbiter*.

3.3 Processing Element (PE) Architecture

Figure 3.4 shows the architecture of a PE module, which is used to execute inner loop computations in a kernel. The PE architecture decouples and overlaps data movement and execution, by allocating separate modules for computation (*Datapath*), and data movement (*Stream Interface Unit*). The stream interface unit allocates a set of memory traffic management modules, including a programmable Address Generation Unit (*AGU*) for memory read requests. Separate modules are allocated for input and output streams to allow overlaping data read and write operations.

Institutional Repository - Library & Information Centre - University of Thessaly
06/05/2024 10:45:18 EEST - 3.144.45.101

The architecture favors streaming applications with regular and predictable memory access patterns by allocating separate modules for addresses generation and data computations and by processing memory read and write requests independently. However, in case memory access patterns are dependent on runtime computations, addresses and data computations are mapped on a unified-as opposed to decoupled-datapath. If irregular or a runtime-dependent RAW dependency exists, then separate input and output streaming units are also merged to preserve the execution order of memory read and write operations. This unified configuration of the PE architecture is more suitable for non-streaming applications with I/O traffic dependences that can be resolved only at runtime.

3.3.1 Datapath and AGU Modules

The *Datapath* module absorbs data tokens loaded from memory, performs computations, and then pushes output data tokens back to the streaming unit for write back to local or global memory. In a unified datapath configuration, it also performs address computations. An Address Generation Unit (*AGU*) aggressively generates
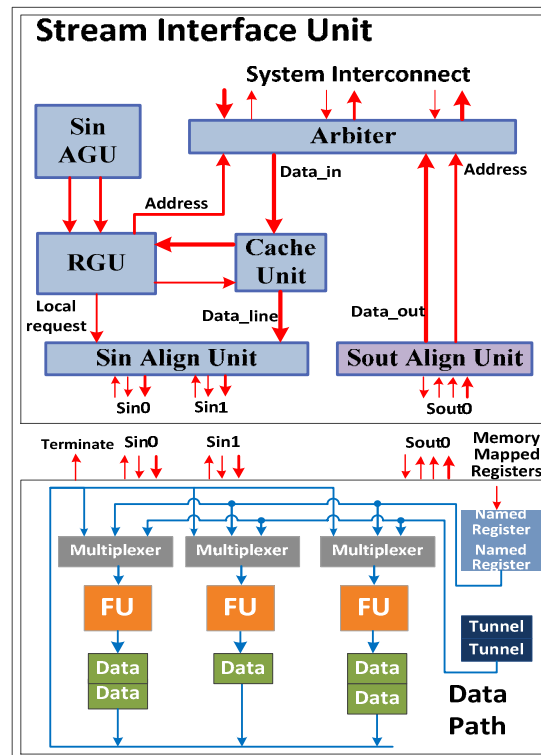
Figure 3.4: Processing Element (PE) architectural template.

36

addresses for data prefetching, and feeds them to the Requests Generation Module (*RGU*). The tool flow guides the generation of the AGU by first identifying the code slice responsible for data I/O, and then performing modulo scheduling on that code, as we will show in Section 4.3. 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 datapath, thus the same methodology is used to generate hardware in both cases. Figure 3.5 shows the datapath generated for *PE(L_{1_0})* module in Figure 3.2b. A datapath includes three types of components: functional units (FUs), storage units, and the control unit.

3.3.1.1 Functional Units

The datapath (and AGU) 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 (*Sin0*, *Sin1* and *Sout0* in Figure 3.4). Each FU is preceded by a multiplexer tree, which, at each time-slot, directs data elements into the correct input port. The multiplexers are driven by a periodic-count of the initiation interval (II) generated by the control unit.

Each FU supports the execution of specific operation type. SOpenCL tool supports a large pool of operation types classified as follows:
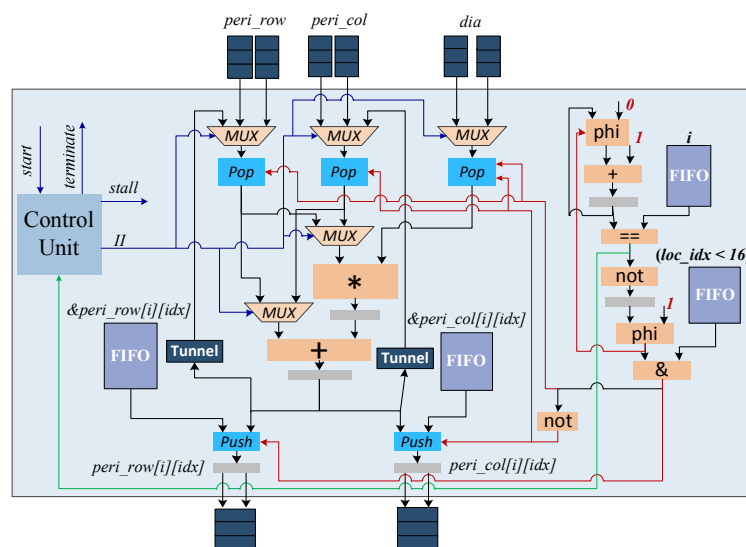
Figure 3.5: Datapath of the *PE(L_{1_0})* module in Figure 3.2b.

37

- *Primitive Integer Operations:* Basic arithmetic and logic operations like *Add, Mul, And, Shift*, and so on.

- *Single Precision Floating Point Operations:* SOpenCL uses a library of single precision IEEE-754 compliant floating point units (FP Unit). Multiple versions of each floating point operation are implemented. Each implementation is tagged with its precision, its latency, as well as the number of its pipeline stages. At compile time, the system selects and integrates the appropriate implementation according to precision requirements and the target initiation interval. We used FP units generated by FloPoCo [94] arithmetic unit generator.

- *Mathematical Operations*: The tool utilizes a library of commonly used mathematical operations, such as square root, exponent, sine, cosine, arctan, etc. The library includes two FUs supporting the *sine* and *cosine* operations: one FU implementation is based on Taylor series with latency equal to 28 clock cycles, and the second one implemented using CORDIC algorithm [95] with 40 clock cycles latency. The latter also supports the execution of *arctan* operation. The square root FU core uses a polynomial approximation with latency equal to 5 clock cycles. Square root and exponent FUs are generating by FloPoCo [94].

- *Application Specific Operations*: The tool flow analyzes computation patterns in the loop and extracts common computational expressions to implement then as custom FUs. Section 4.4 details the methods used in extracting application specific instructions.

The size and number of functional units and types of supported operations are configurable parameters, decided by the tool flow to achieve the computations requirements and user performance specifications.

3.3.1.2 Storage Units

The datapath also includes registers and FIFOs that hold loop invariant data generated by outer loops executed in parent CE modules. Figure 3.5 shows few of the data FIFOs generated, used to temporarily store incoming data from local arrays such as *peri_row* and *peri_col*, and inner-loop invariant variables like the outer-loop index *i*. The size of each FIFO is a configurable parameter that can be assigned to match the data rate at the specific FIFO channel.

38

Tunnels are storage elements used to bypass the streaming unit and channel data-tokens stored in earlier iterations to be used by loads in later iterations. Tunnels are generated wherever a load instruction has a RAW dependency with another store instruction with constant cross-iteration distance larger than or equal to one, for example for code portions like the following:

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

The tunnel size (i.e. the number of tunnel registers)is equal to the dependency distance, because once a valid data token leaves the tunnel, the corresponding *pop* FU starts reading data tokens from the tunnel and ignores data from the input FIFO channel (*Sin0, Sin1*, etc.) coming from the *Stream Interface Unit*.

Figure 3.6 shows the C code of the inner loop L_{1_0} in Figure 3.2a. Due to the need to accumulate values on the *peri_row* and *peri_col* arrays, the loop has two RAW dependencies with distance 1 in each of these two data arrays. Two tunnels are generated one for each with tunnel size equal to 1 as shown in Figure 3.5.

3.3.1.3 Control Unit

The control unit is responsible for initiating the execution of the datapath and generating a periodic count (II) used by the FU multiplexers to select proper input data at each time slot. The control unit stalls the *datapath* if any of the input data FIFOs (e.g. *i*, *loc_idx<16*) and streams FIFOs (e.g. *dia*) is empty, or any of the output streams FIFOs (e.g. *peri_row*) is full.

The control unit is also responsible for terminating the execution of the *datapath* by monitoring the loop termination condition, such as the comparator output in Figure 3.5. As soon as the termination condition turns true, the control unit waits for a

```
for( j = 0;  j < i;  j++){
   if(  loc_idx < 16 )
      peri_row[i][idx] += peri_row[j][idx] * dia[i][j];
   else
      peri_col[idx][i] += peri_col[idx][j] * dia[j][i];
}
```

Figure 3.6: L_{1_0} Loop C source code in Figure 3.2a.

39

predetermined number of clock cycles until the last loop iteration ends, and then it resets the *datapath*. Section 4.5.1.3 details how modulo scheduled loop is terminated.

3.3.2 Stream Interface Unit

The stream interface unit handles all issues regarding data transfers between the main memory and the datapath. These include data alignment, data ordering, and bus arbitration and interfacing. The streaming unit allocates multiple independent input and output streams processing modules. Those modules process generated addresses and prevent redundant or unnecessary requests from reaching local or global memory.

Local arrays (*peri_row, peri_col,* and *dia* in Figure 3.2*a*) or input arrays are considered distinct streams of data. Each stream of data is allocated its own set of processing units.

3.3.2.1 Input Streaming Units

Each input data stream is processed by a couple of tightly connected units: Requests Generation Unit (RGU) and Input Stream Alignment Unit (*SinAlign*). The *RGU* module receives addresses generated by the AGU and issues read requests to external memories, while *SinAlign* unit retrieves data tokens, and packs them in order to the datapath.

The *RGU* coalesces read requests generated by *SinAGU* (or the *datapath*) to the word width of the underlying memory interconnect (for example, a PLB bus for Xilinx FPGAs), or to burst size if bursting is enabled. The *RGU* aims to eliminate redundant transactions on the memory interconnect. Before issuing a transaction request to the *arbiter* it checks if the addresses aliases with previously requested ones or if the data are available in the cache (if the cache has been instantiated).

Figure 3.7 depicts how the *RGU* and *SinAlign* unit process each generated read address from the *AGU* (or the *datapath*) until the data token is loaded from the memory and presented to the datapath. The process flow can be summarized as follows:

40
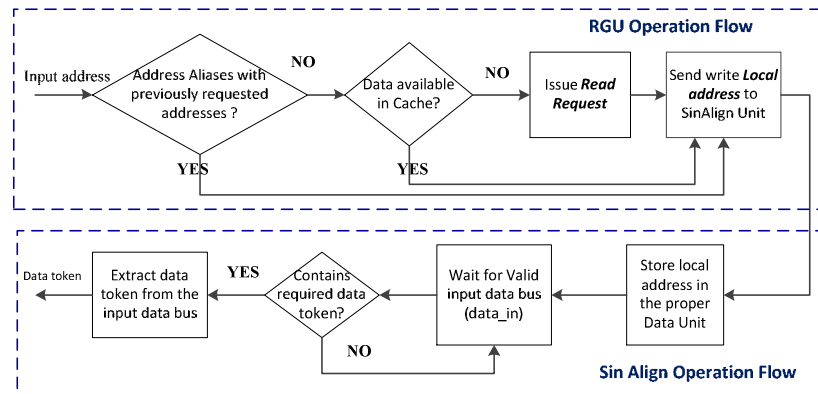
Figure 3.7: *RGU* and *SinAlign* modules operations flow.

- The *RGU* first checks if the input address aliases with previously issued addresses stored in *WReqs* and *SReqs* FIFOs (Figure 3.9) or not. If an address alias is found, the *RGU* issues a local address to the *cache* and the *SinAlign* unit to retrieve data token from input data line (*Data_line* in Figure 3.4). The *cache* uses the local address to store the incoming data line (*data_in* in Figure 3.4) and writing *Data_line* to the *SinAlign* unit.

- If no address alias is detected, the *RGU* checks if the cache has valid data (if the cache allocated) or not. If the cache has valid data then a local address is issued to the *cache* and *SinAlign* unit to retrieve the data token from input data line (*Data_line* in Figure 3.4).

- If the *cache* has no valid data, then the *RGU* issues a read request to the *arbiter*, and then issues a local address to the *cache* and *SinAlign* units to retrieve data token from input data line.

- The *SinAlign* unit stores input local address in the corresponding *Data unit* (Figure 3.9) and then waits for incoming data line (*Data_line*). A local address is shortcut of the complete address consists of two components: *Offset* and *Code*. Figure 3.8a shows a 5-bit local address. The *code* component is a unique ID given for each read request stored in the *WReqs* and *SReqs* FIFOs. The *SinAlign* unit compares this ID with the incoming data line tag (*Data_line_tag* signal not shown in Figure 3.9, accompanies *Data_line*) to check if the incoming data line contains the required data token. If true, the *SinAlign* unit extracts the proper data token from the input data line using the *offset* component. The *offset* component

41

4	3	2	1	0
Code		Offset		

(a)

Offset	000	001	010	011	100	101	110	111
	0	1	2	3	4	5	6	7

Data Bus

63   0

(b)
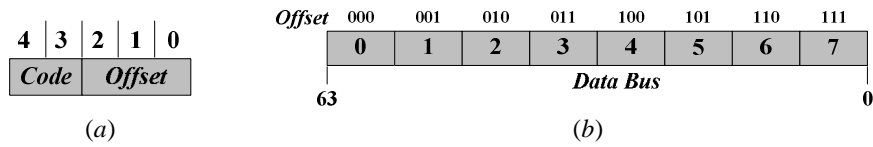
Figure 3.8: Local Address Encoding. (a) 5-bit local address. (b) Offset values for 1-byte data token in 64-bit Data Bus.

is used to retrieve the proper data token bytes within a data line. For example, for a 1-byte data token, Figure 3.8b shows the offset value for each byte in a 64-bit data bus.

The *RGU* module can be configured to process multiple addresses in parallel or once a time. The *RGU* module takes different shapes depending on the data stream type and characteristics. Figure 3.9 shows three basic shapes of the *RGU* module. For a global or local input data stream, the *RGU* follows the configuration in Figure 3.9a. The *Cache Access Logic* block is not used for local data streams, as well for streams that don't use the cache. A data stream of constants will use a much simpler *RGU*; each input address port is allocated a ROM that stores the array of constants (Figure 3.9b).
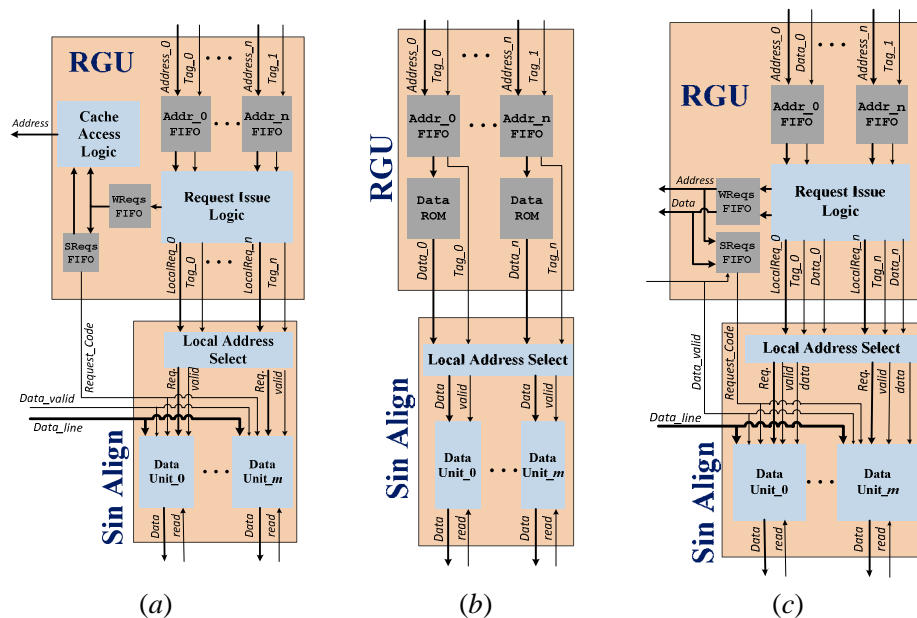
Figure 3.9: *RGU* and *SinAlign* modules configurations for (a) cached and non-cached data streams. (b) stream of Constants. (c) Streams with runtime RAW dependencies. *WReqs FIFO* refers to *Waiting Requests FIFO*. *SReqs FIFO* refers to *Sent Requests FIFO*.

42

The *RGU* module could serve data write requests (Figure 3.9c) as well in the special case of irregular or runtime dependent RAW dependencies as in the following code:

$$for\ (i = 1;\ i < N;\ i++)$$
$$a[c[i\text{-}1]] = a[c[i]] + b[i];$$

In this special case the *datapath* will be responsible for generating read and write addresses. The scheduler will consider the available RAW dependency and produce a correct schedule. However, since the *SoutAlign* Unit and *RGU* are completely independent, and the *datapath* does not wait for write acknowledge signal, there is no guarantee that the read/write requests order generated by the datapath will be preserved on the interconnect bus. Hence, both read and write requests, are served by the same *RGU* module which preserves their execution order. Moreover, the *RGU* module will exploit address coalescing resources to retrieve data tokens from a write request, and prevent unnecessary read request from reaching the interconnect bus.

The *SinAlign* module retrieves data from the cache unit or the *data_in* incoming data in case the data stream is not cached, and presents them in-order to the datapath. For each load instruction in the loop, the *SinAlign* module allocates separate alignment logic and FIFOs (*Data Unit_m* in Figure 3.9). This allows the *SinAlign* modules to serve multiple load instructions in parallel and out of order.

The *Local Address Select* block in Figure 3.9 works as demultiplexer by directing each incoming local address to the proper *Data Unit*. The *Tag* signal that accompany each local address indicates to the corresponding load instruction produced the address, and hence to which *Data Unit* the local address should be directed.

The *SinAlign* Unit is tightly coupled with the *RGU* module, and variations on its configuration follow closely any variations on the *RGU* configuration. For global and local data streams, the *Align Path (Data Unit* in Figure 3.9) includes a FIFO that store local addresses and retrieves data tokens. A single *Data Unit* can retrieve multiple data tokens simultaneously, if multiple local addresses — stored in its FIFO — have the same request *code* component. For a data stream of constants the *Data Unit* is a FIFO that stores only data tokens obtained from the ROM.

43

Each output data stream is allocated its own Output Stream Alignment unit (*SoutAlign*). The *SoutAlign* unit aligns the output data tokens coming from the datapath in a FIFO of data-lines of bus-width bytes (*Align Data FIFO* in Figure 3.10).

The operation of the *SoutAlign* unit can be summarized as follows:

- For each incoming write request (which includes address and data token), the *Align Logic* unit (Figure 3.10) checks if the input write address aliases with previous addresses stored in the *Align Data FIFO*. If an alias found, the proper data line in the *Align Data FIFO* is updated with the input data token.

- If an address alias is not found, the *Align Logic* unit stores the input address and data token in an empty line in the *Align Data FIFO*. If the *Align Data FIFO* is full, then the *Align Logic* unit sets the *issue* signal to true. The *Issue Request* unit then issues a write request to the *arbiter* (or a local memory) to make a space in the *Align Data FIFO*.

- When the datapath terminates, all data in the *Align Data FIFO* is written to the memory before new write requests stemming from the *datapath* are written in the
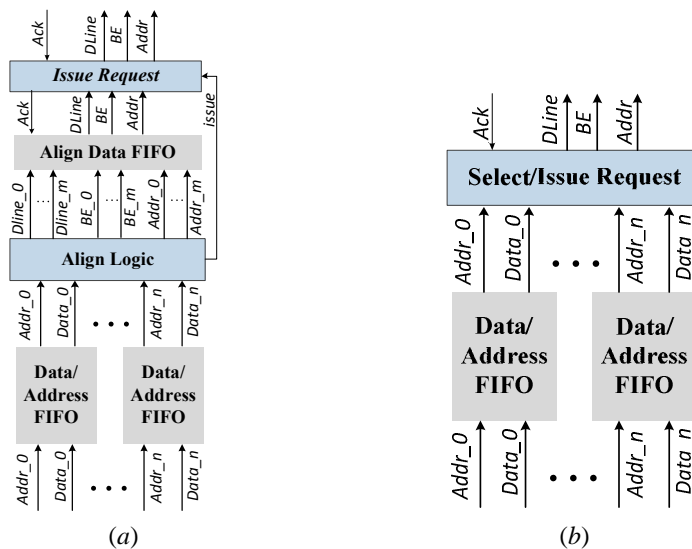
Figure 3.10: SoutAlign module. *BE* refers to the *Byte Enable* bus signal. (a) Generic *SoutAlign* unit configuration. (b) *SoutAlign* unit configuration when no address aliases detected at compile time.

44

Align Data FIFO.

The *Align Logic* unit provides parallel alignment capability, by writing aligned data tokens to multiple data lines in the *Align Data FIFO*, and by writing multiple data tokens in the same data line simultaneously.

The *SoutAlign* unit eliminates repetitive writes to the same memory location by overwriting old data tokens in the *Align Data FIFO* with newly produced data tokens. The mechanism of overwriting old data in the *Align Data FIFO* is applied until the datapath terminates or the *Align Data FIFO* is full and a data line (where data is overwritten) must be written to the memory to make space in the FIFO. Also if the *Align Data FIFO* is full RAW dependencies are not violated by the overwriting of old data. Regular RAW dependencies are served using tunnels, and irregular dependencies are served by directing write requests through the *RGU* module and removing the *SoutAlign* unit. Write-after-write (WAW) dependencies are considered by the scheduler, and since they pass through the same *SoutAlign* unit, their execution order is preserved.

The *SoutAlign* unit follows a simpler configuration (Figure 3.10b) if the SOpenCL detects no aliases between successive addresses at compile-time, and hence, remove the *Align Logic* unit and *Align Data FIFO*. The *SoutAlign* unit in this configuration simply works as arbiter serving one data token each clock cycle.

3.3.2.3 Local Cache

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 *SinAlign unit* can be served simultaneously.

A cache line is equal in size to the bus width. The *cache* unit is not instantiated if compile time analysis determines that the input memory access pattern has limited reuse. The cache unit is configured as a set of data blocks possibly with different sizes. Each distinct data stream stored in the *cache* is allocated a number of data blocks with specific size determined by SOpenCL, as will be discussed in Chapter 4. Compared to conventional caches, the cache unit has the following differences:

45

- It is a read only cache; data transferred from main memory to the *cache* but not the other way.

- A block of data is allocated a space in the cache but no read operation on the whole block is performed. A data line in the block will be transferred from the main memory only if a data request to a data token in that line is generated by the *SinAGU*. In other words, a data line is read on demand.

- The cache is accessed only by the PE module associated with it. No other PE modules have access to that cache.

- The lifetime of a data stream in the cache ends when another PE or CE module starts a write transaction to the data stream in the main memory.

It is not necessary that all input data streams utilize the *cache*. SOpenCL will detect data streams with temporal and spatial locality and recommend whether a *cache* will be instantiated as part of the architecture.

3.4   Control Element (CE) Architecture

The control element (CE) serves as the glue connecting all the accelerator components by directing the execution flow. The CE module implements and executes outer loops and loop invariant statements. In Figure 3.2b, CE modules *CE0*, *CE1*, and *CE2* execute the statements (blocks of instructions) in outer loops *T0, T1*, and *T2*, respectively. Figure 3.11 outlines the architectural template of the CE module. The architecture consists of three types of components:

- *Computational components:* functional and storage units.
- *Control FSM:* A finite state machine used to control the execution flow and provide synchronization information for the CE children (PE and other CE modules).
- *Streaming and memory interface:* a set of streaming units used to issue read/write requests, and retrieve data tokens and acknowledgements.

46
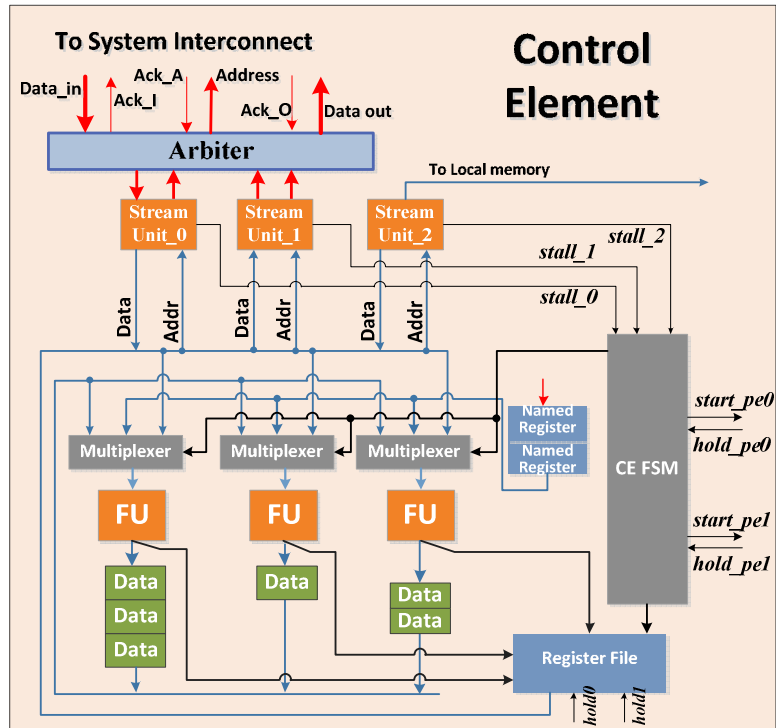
Figure 3.11: Control Element Architectural Template.

3.4.1 Functional and Storage Units

Computational components include a network of functional units (FUs), multiplexers, registers and queues. The instructions blocks within the loops are statically scheduled on the allocated FUs, and the multiplexers are configured at compile time to fulfill the interconnection requirements of the scheduled FUs, i.e. direct the proper FU output or registered data to the proper FU input port at each time slot in the schedule period.

The CE module supports the same types of functional units mentioned in section 3.3.1.1. However, the amount of FU resources allocated is typically less than the resources allocated for a datapath. The storage units in the CE module include scalar data static registers and FIFOs (similar to the ones described in section 3.3.1.2), and a register file. The register file holds scalar variable with lifetime outside the boundaries of a basic block. Figure 3.12 shows some of the Loop *T0* statements mapped on *CE0* (Figure 3.2), and the register file generated for *CE0*. In Figure 3.12, variables *r0*, and *r3* in block *bb0* (not shown) are used in block *bb5*, hence, they are

47

saved in the register file to be used later, because the queue of an FU is reset after block execution finishes.

3.4.2 Control Unit

The Control Unit implements the control transfer logic between blocks of instructions as well as with successor CEs and PEs. The transition between FSM states is guided by the execution of the control transfer instructions (*br* instructions in Figure 3.12) in the current executing basic block.

The FSM state drives the generation of control signals such as the schedule length, and trigger signals of children modules such as *start_pe0, start_pe1,* etc. Schedule length is the number of clock cycles required to finish the execution a block of instructions, e.g. the schedule length of block *bb3* in Figure 3.12 equal to 4 clock cycles. The value of schedule length is computed at compile time after instruction scheduling. The FSM selects the proper schedule length value depending on the block currently executing.

Similar to the control unit in the PE datapath, the FSM control unit stalls CE module execution when there is a read/write request waiting in a stream unit to be served, and when input scalar data is not available or the register file is stalled by a *hold* signal from another PE/CE module.
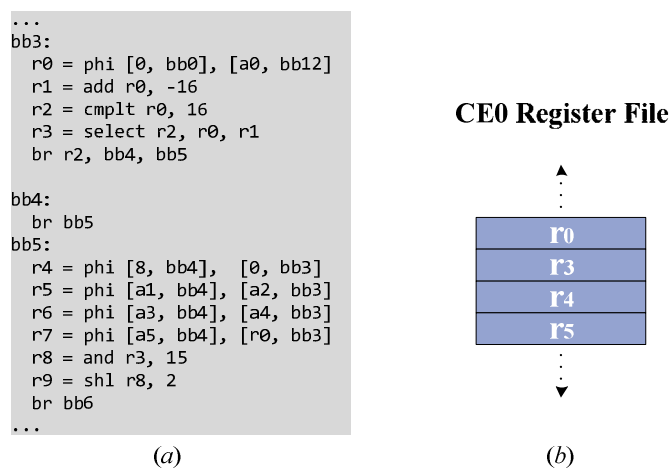
Figure 3.12: CE Register File allocation. (a) Part of the outer loop statements of Loop *T0* in Figure 3.2. (b) Snippet of the Register file of module CE0 in Figure 3.2.

48

3.4.3 Streaming Interface

The streaming interface in the CE architecture consists of a set of stream processing units and off-chip memory arbitration. Each data stream is allocated its own *Stream Units*. The stream units have the simple task to issue read/write request address and to retrieve data tokens or write request acknowledgement. Unlike PE architecture streaming units, the CE streaming units serve one read/write request each time; there is no address coalescing, no reuse mechanisms, and no cache support (Figure 3.13). The assumption is that the CE memory traffic is very small compared to that of the PE module; as the PE module normally has more data traffic executes N times the number of its parent CE execution iterations (where N is the loop trip of the inner loop executed by the PE module).

In Figure 3.2, *CE1* module executes the statement *peri_col[idx][i] /= dia[i][i]* where two read and one write operations are performed on local data arrays *peri_col* and *dia*. Hence, *CE1* module allocates two input stream units (as in Figure 3.13a) for read operations from *peri_col* and *dia* local streams, respectively, and one output stream unit (as in Figure 3.13c) for write operation to *peri_col* local stream. The stream units in a CE module share the same local buffer or global memory ports with PE modules. In Figure 3.2, *CE1* shares *peri_col* and *dia* local buffers with PE modules (interconnects are not shown in the Figure for clarity).

The CE streaming interface includes an arbiter that manages requests to an off-chip memory; all stream units accessing an off-chip memory assigned an arbiter that manages their requests and acknowledgements. A stream unit that processes data
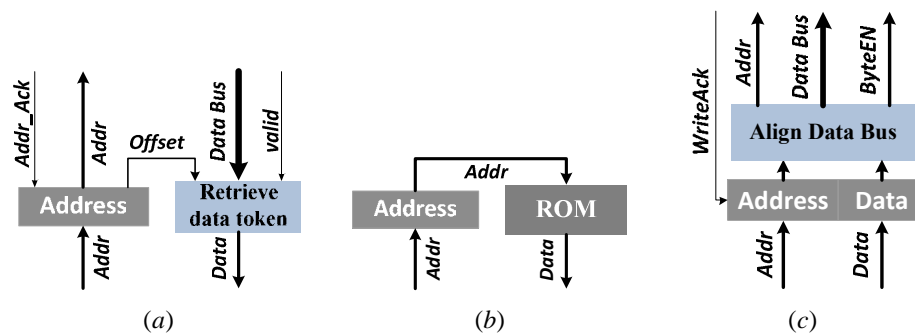
Figure 3.13: CE Stream Unit Configurations. (*a*) Typical input stream streaming unit. (b) Stream unit supports array of constants. (*c*) Typical output stream streaming unit.

49

arrays in local buffers has a direct link to the local buffer system. The stream units also support arrays of constants (Figure 3.13b). Like the *RGU* module, the stream unit allocates ROM storing the array of constants.

3.5   Execution Model

Figure 3.14a shows a synopsis of the FSM of *CE0* in Figure 3.2b. In a sequential execution model, a control transfer occurs (FSM state changes) when a basic block of instructions (e.g. *B00, B01*) finishes execution and a control transfer operation (*br, switch*) is executed. According to this model, a CE will not initiate a new execution of a successor module (PE or CE) until that successor finishes previous execution. A PE (or CE) emits a true finish signal to transfer control back to its parent CE. For example *PE02_finish* and *PE03_finish* signals used in Figure 3.14a FSM are generated by $PE(L_{0\_2})$ and $PE(L_{0\_3})$ (Figure 3.2b), respectively. This FSM model will reduce the architecture into a sequential processor consisting of multiple hardware units executing one at a time. Figure 3.15a depicts the sequential execution flow of all architecture components.

SOpenCL uses a concurrent execution flow, instead of the slower sequential model. A control transfer from a basic block occurs when it finishes execution, but a control transfer from a successor PE or CE will not wait for a finish signal, given the destination is known at compile time.

Figure 3.14b shows a synopsis of the FSM with concurrent execution model. When the FSM state reaches states *PE02*, and *PE03*, *CE0* children $PE(L_{0\_2})$ and

```
…
case ( fsm_state )
   B00  : next_state = (B00_finish)?  B00_destination : fsm_state;
   PE02 : next_state = (PE02_finish)? PE03            : fsm_state;
   PE03 : next_state = (PE03_finish)? B01             : fsm_state;
   B01  : next_state = (B01_finish)?  B01_destination : fsm_state;
   EXIT : next_state = (start)?       B00             : fsm_state;
endcase
…
                              (a)
…
case ( fsm_state )
   B00  : next_state = (B00_finish)? B00_destination : fsm_state;
   PE02 : next_state =  PE03;
   PE03 : next_state =  B01;
   B01  : next_state = (B01_finish)? B01_destination : fsm_state;
   EXIT : next_state = (start)?      B00             : fsm_state;
endcase
…
                              (b)
```

Figure 3.14: Synopsis of the FSM of CE0. (a) Sequential execution mode FSM. (b) Concurrent execution mode FSM. The FSM in (b) drops signals *PE02_finish* and *PE03_finish* in states *PE02* and *PE03,* respectively.
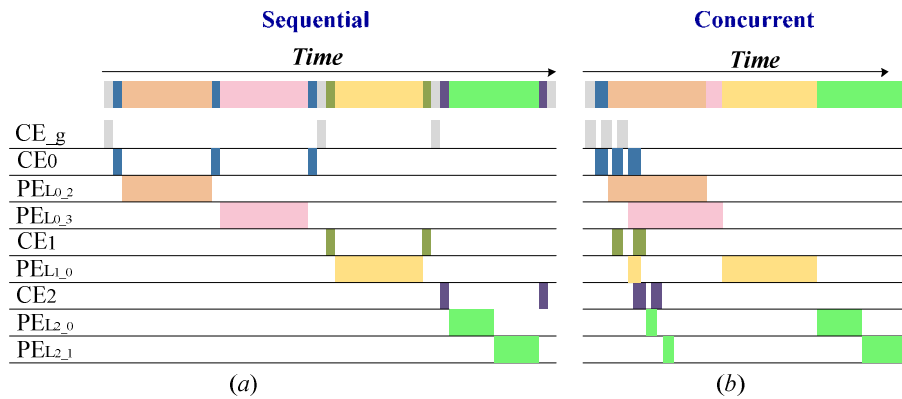
Figure 3.15: Timing for a work-item execution for the architecture of Figure 3.2b using (a) sequential execution flow, and (b) concurrent execution flow.

$PE(L_{0\_3})$ are triggered. The FSM in Figure 3.14b drops signals *PE02_finish* and *PE03_finish* in states *PE02* and *PE03* respectively. Both $PE(L_{0\_2})$ and $PE(L_{0\_3})$ will be triggered with distance one clock cycle. In other words, both modules will execute in parallel as long as there are no data dependencies between them. Figure 3.15b depicts the concurrent execution flow.

Concurrent flow requires a mechanism to preserve data dependencies between multiple PE and CE modules. A simple handshake synchronization mechanism is used. Two PE or CE units that have either a memory or scalar data dependency will exchange two signals: *Finish* and *Hold*, and if they have multiple dependencies they exchange multiple pairs of *Finish* and *Hold* signals one for each dependency. A producer will emit a *Finish* signal as soon as it finishes data computations required by other PEs and CEs. A consumer scans the *Finish* signal continuously and saves the incoming data in a FIFO when the *Finish* signal is true. If the data FIFO at a consumer is full, the consumer will emit a *Hold* signal and the producer will stall execution until the consumer can absorb the data. For memory dependencies, the consumer (reader or writer) will save the *Finish* signal itself in the FIFO since the data saved either in local or global memory.

Adopting the concurrent execution model allows parallel execution of multiple independent PE and CE modules. One major benefit is hiding prologue and epilogue latencies of inner most loops (Figure 3.16). In the PE module, the *AGU* and *datapath* run as separated entities. Figure 3.16a shows the sequential execution model,

51

according to which the next iteration of an outer loop can be initiated only after the last iteration of the inner loop. The sequential model creates execution bubbles at the prologue and epilogue of each outer loop iteration (*ET0* and *ET2*, respectively), during which computing resources remain idle, thus causing unnecessary execution delays. *ET0* refers to the execution time of computations in the outer loop executed before a PE module is initiated. *ET1* refers to the execution time of the PE module. And *ET2* refers to the execution time of computations in the outer loop executed after the PE module finishes execution. *Tin* in Figure 3.16 refers to the time required to initialize the *datapath* (and the *SoutAlign* unit) with input data. In the sequential execution model, at least one of the PE module components (*AGU*, *datapath* or *SoutAlign* unit) stays idle.

Using the concurrent execution model we can ameliorate the sequential execution model inefficiency. By initiating the next outer loop iteration, the parent CE will retrigger the successor PE while it still executes the work load of previous iterations. In Figure 3.14b, the FSM state will reach the *PE02* and *PE03* states while the corresponding PE children still executing previous iterations. This early trigger of a child, forces the *AGU* and *datapath* to start execution of next outer loop iterations as soon as it finishes previous ones (Figure 3.16b).
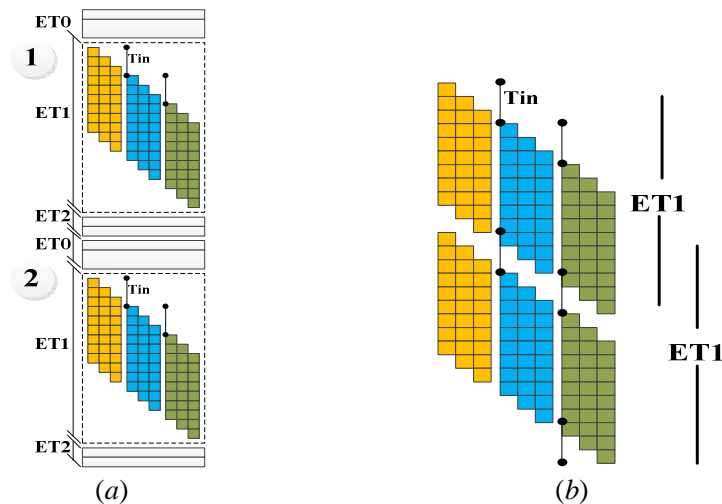
Figure 3.16. Nested loop execution model (a) when there is no overlap between successive outer loop iterations (sequential model) and (b) when successive outer loops overlap (concurrent mode). *SinAGU*: yellow, *Datapath*: blue, and *SoutAlign*: green

52

3.6   Related Work

Prior research in architectural synthesis has investigated a variety of hardware accelerators architectures. The variations between the introduced architectures resulted mainly from the way each architecture partitioned the input specification into multiple blocks and the interconnect between them.

PICO-NPA [13] generates a Non-Programmable Accelerator (NPA) for a C function comprising a single perfectly nested loop. The NPA architecture consists of an array of multiple instances of a datapath processor, a memory controller, a control unit, and an interface to the host processor (Figure 3.17a). The architecture includes also local memories shared by the datapath processors. A datapath instance implements a modulo-schedule of the inner most loop in the loop nest (Figure 3.17b). The PICO-NPA compiler distributes outer loops iterations over the allocated datapath processors equally. It is the responsibility of the host processor to initiate processors execution, initialize processors with data and loops indices.

The PICO-NPA architecture is a paradigm for a coprocessor with a host processor as its central control unit. While this paradigm provides an efficient implementation of a coprocessor and can speedup loop execution, shifting the control logic to the host processor restricts parallelism between multiple NPA coprocessors, and reduces NPAs to application specific execution units in a VLIW processor.

The Trident system [26] synthesizes a hardware accelerator from a C function with one or more arbitrary loop nests. Trident performs if-conversion (predication) to generate hyper blocks of instructions. A hyper block is created by removing all

(a)   (b)
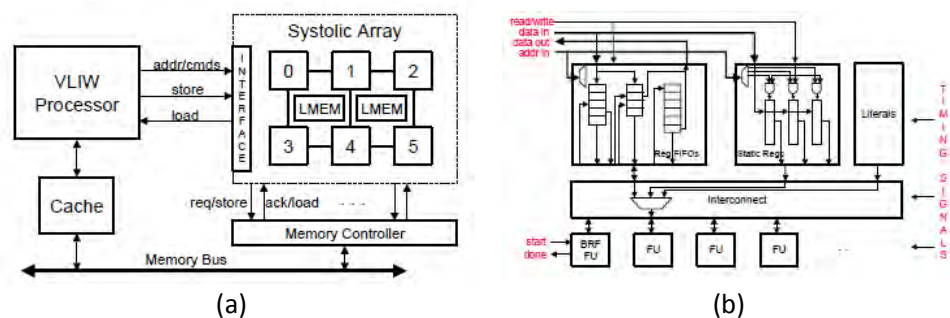
Figure 3.17: PICO-NPA system. Figure copied from [13]. (a) NPA architecture: systolic array of processing cores. (b) Processing core datapath.

53
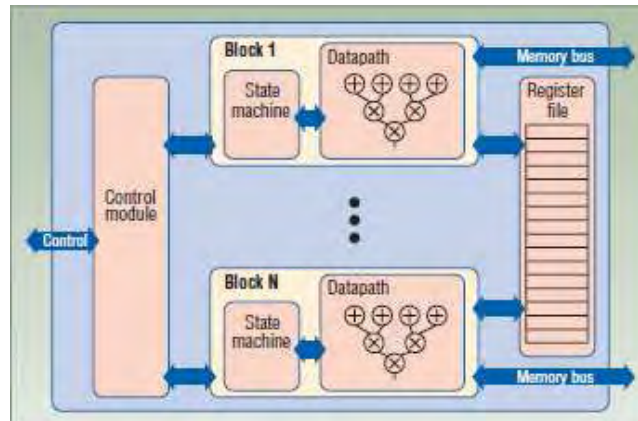
Figure 3.18: Trident system target architecture. Figure copied from [26].

branch instructions between a set of basic blocks and putting their instructions in a single block. Trident uses If-conversion to create hyper blocks. The Trident system generates an architecture consisting of multiple subcircuits each implements a hyperblock (Figure 3.18). A subcircuit consists of a state machine and a datapath. All subcircuits share a single file register to store scalar variables. The architecture top circuit includes a control module that manages control transfers between hyper blocks and exchange control signals with a host processor. Trident is one of the few synthesis tools that support floating point operations using multiple libraries.

Like PICO-NPA, Trident system doesn't provide any sort of synchronization mechanism between multiple hyper blocks, hence, blocks of Figure 3.18 execute sequentially. On the other hand, Laura [44] architecture utilizes sophisticated synchronization mechanisms allowing multiple processing units to run in parallel. Laura architecture (Figure 3.19) follows closely a Kahn Process Network (KPN) specification [45]. Laura uses the *Compaan* compiler [46] to generate a KPN specification from Matlab applications. The work in [47] builds upon Laura framework to support C functions.

A KPN computation model assumes concurrent autonomous virtual processes (VP) that communicate in point to point fashion over unbounded FIFO channels. In KPN model, a VP is a perfectly nested loop. KPN computation model is applicable on streaming applications with regular data streams. The streaming feature of KPN models allows pipelining producer-consumer VPs. To overcome the issue of

54
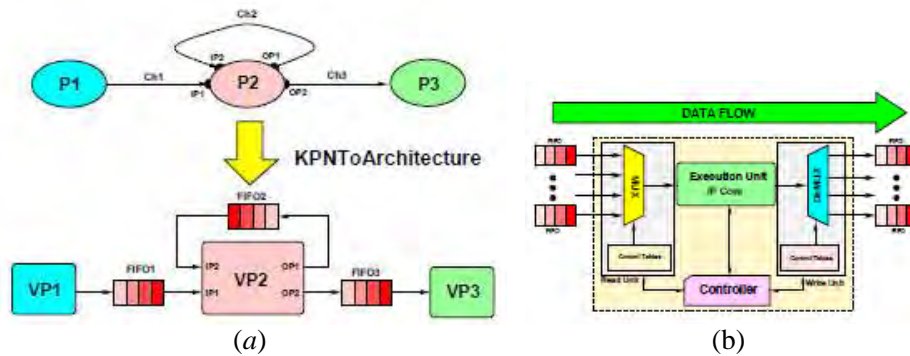
Figure 3.19: Laura target architecture. Figure copied from [44]. (*a*) Network of KPN virtual processes. (b) Architecture of a VP process.

unbounded FIFO channels in a KPN, Laura supports the use of bounded FIFO channels by applying blocking write synchronization and blocking read synchronization mechanisms.

Virtual process architecture includes three units; read unit, execute unit, and a write unit. In [44], PICO system is used to generate the hardware for the execute unit. Read and write units pop and push data from the proper FIFO channel without the need for address generation. A VP starts execution once all its input data are valid.

ROCCC compiler [48] implies architecture similar to Laura architecture. ROCCC architecture consists of a network of modules, in which each module implements a C function. According to ROCCC programming model, a C function consists of an I/O interface represented as a data structure and an instantiation of a function performs the computation. ROCCC module architecture (Figure 3.20a) decouples memory accesses from datapath computations. Since ROCCC supports regular memory accesses known at compile time, memory accesses are configured at compile time. A smart buffer handles data reuses by keeping data tokens for their lifetime. This requires the compiler to perform data reuse analysis and configure the buffers at compile time.

Similar to Laura and ROCCC architectures, Optimus [36] generates uses an architectural template called *filter* (Figure 3.20b). Optimus stream programming model represents a program as communicating filters. The template consists of five main components: input queues, output queues, memories, the filter itself, and the controller. Input and output queues are used to send and receive data. Each filter can

55

(*a*)	(*b*)
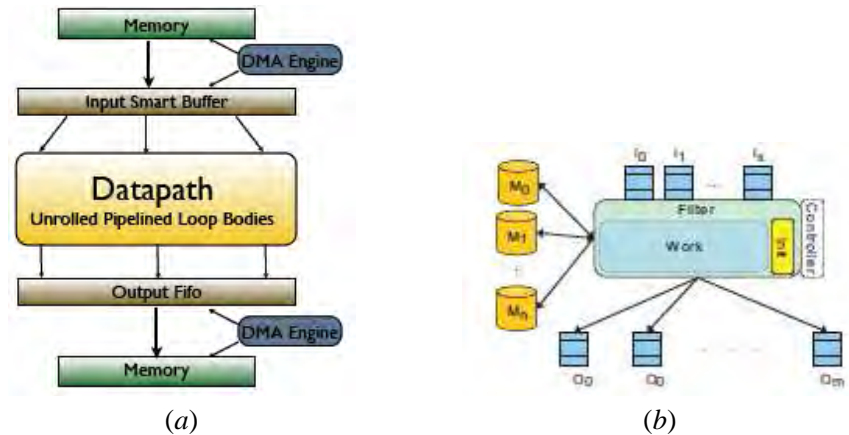
Figure 3.20: (*a*) ROCCC Module architecture model. Figure copied from [48]. (b) Optimus Filter template. Figure copied from [36].

be connected to several memory components. All the memory modules are local to each filter. The hardware block implementing the filter consists of the *work* module (datapath) which performs the computations and an optional *init* module which executes once to initiate the filter. The controller makes sure that the *init* function gets executed only once before the first invocation of the work function.

Contrary to Laura and ROCCC interconnect model, MARC system [49] uses many-core style architecture. The architecture consists of a C-core (Control processor), and many A-Cores (Arithmetic cores) as depicted in Figure 3.21. Each core has its own private/local memory (P/L), and access to global multiport memory through the interconnect network. The datapath of an A-Core can be a simple RISC style processor with 5-stage pipeline, or an application specific core. MARC system builds application specific A-Core datapaths each supporting a set of Super Instructions. A super instruction is a cluster of simple instructions that have a common computation pattern. The scheduler is responsible for mapping statically scheduled instructions on proper A-Core datapaths.

MARC architecture allows as many A-cores to execute in parallel as soon as each core has all its input data available. To exchange data, A-cores will go through global memory, because there are no registers between A-cores. Instructions executing on the same A-Core, share data through A-Core private and local memory. While application specific A-Cores achieve a significant speedup in computations, the absence of point-to-point communication between A-Cores increases the pressure on

56

global memory. Scheduling instructions on A-Cores should be done carefully to minimize the number of data dependencies between multiple cores.
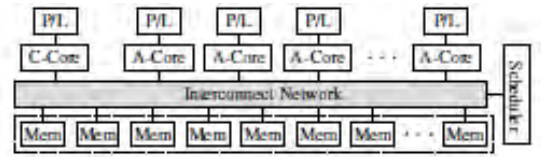
Figure 3.21: MARC System Architecture. Figure copied from [49].

57

CHAPTER 4

SILICON OPENCL BACKEND

SOpenCL backend applies a series of transformations prior to hardware generation (Figure 4.1). These transformations are used for hardware optimizations and are used as a means for generating customized hardware accelerators based on the template described in Chapter 3. Each transformation has a corresponding hardware support in the architectural template of Chapter 3 as will be explained in this section.

4.1 Bitwidth Optimization

General purpose processors (GPP) include functional units, such as ALUs, multipliers, etc. of standard size, (32 or 64 bits). As a result, compilers targeting GPP based platforms produce assembly instructions of the same bitwidth. However, when we design a customized hardware accelerator for a given application, we can control the size of each allocated functional unit. Hence, it is important to remove any redundant bits in every instruction size to minimize the size of functional units, and reduce overall area.

Bitwdith optimization has been developed as a separate LLVM optimization pass to compute the minimum number of bits needed to represent every integer variable (i.e. instruction) in the application. On the other hand, floating point variables are
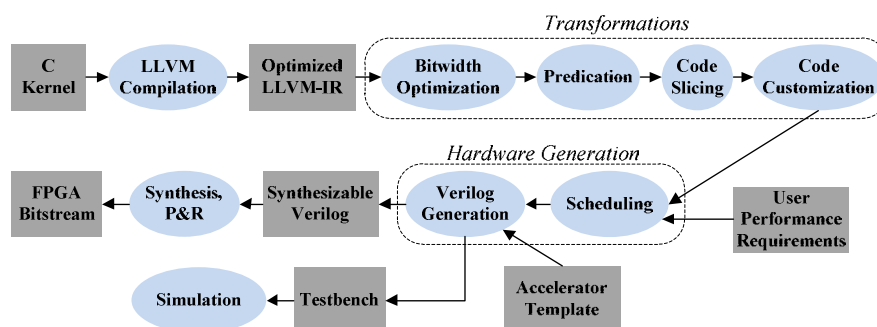
Figure 4.1: SOpenCL backend transformations.

58

IEEE-754 compliant and use the 32-bit for single and 64-bit for double precision, respectively.

Bitwidth optimization for integer variables is a value-range propagation problem. The value range (e.g. 0 to 255 for char variables) of a variable is propagated through the program data flow graph (DFG) to compute the value range of subsequent variables. The bitwidth optimization algorithm uses three types of information as input to the value propagation engine:

- *Variable data type:* Data types like *char, unsigned char* indicate a value range [-128,127] and [0,255], respectively.

- *Static Array Size:* Static arrays size like *A[256]* can be used as an upper bound on array index variables.

- *Loop carried linear expressions and loop trip count:* a loop carried expression, like most loop iteration index variables (e.g. k += 2), can be solved provided that the loop trip count is known and the expression is linear.

As an example, refer to Figure 4.2. Input data stream *A*, and *B* have *char* data type with value range [-128, 127]. Propagating their value range to variables *s0* and *s1* leads to value range [-256, 254] and [-255,255], respectively. The static array *C[16]* size places a bound on the variable *N*, hence the value range [1, 16] . The variable *i* value range is computed using its loop carried expression, and value range of *N*, hence the variable *i* takes value range [1, 16]. Using the computed value range for each variable, we compute the number of bits required to represent that value range

```
                          [-128 : 127] 8-bits

            int foo ( char * A, char * B, int N ){

                int i, s0, s1, C[16];

    9-bits      C[0] = 0;
[-256 : 254]→s0   = A[0] + B[0];
                                            5-bits
    [1 : 16]────────────┐         ┌────[1 : 16]
     5-bits     for ( i = 1; i < N; i++ ){
[-255 : 255]→    s1   = A[i] - B[i]
    9-bits         C[i] = s1 * s0 + C[i - 1];
                }

                return C[N-1];
            }
```
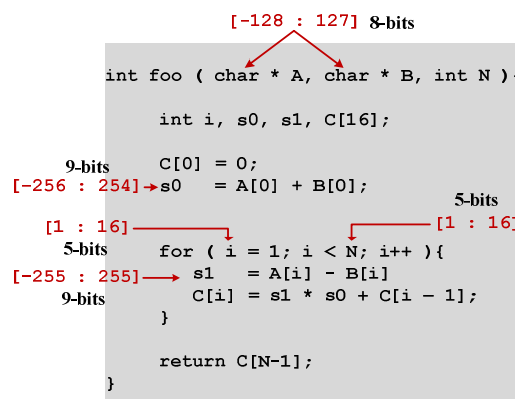
Figure 4.2: Bitwidth optimization example.

(as shown in Figure 4.2).

Bitwidth optimization significantly reduces the size of functional and storage units. Instead of 32×32 multiplier, we need only a 9×9 multiplier to compute *s1*s0*, and instead of a 32-bit adder, a 9-bit adder suffices to compute *s0 + s1*.

4.2 Predication

4.2.1 Overview

Wide-issue architectures require a sufficient amount of instruction level parallelism to achieve peak performance. Control transfer instructions impose a significant restriction on available Instruction Level Parallelism (ILP), and hence, lead to a serious restriction on performance.

Many studies proposed predicated execution as a method to increase ILP [51, 53, 55]. Predicated execution eliminates control transfer instructions and replaces them by predicate-defining instructions and guarding instructions. This transformation replaces control dependencies with data dependencies. An instruction is executed as soon as its data operands and predicates are available. Compilers support predicated execution by applying *If-Conversion* transformation, in which code with multiple basic blocks of instructions is translated into a single block Figure 4.3 shows a simple example of the outcome of If-conversion. Instructions *I0* and *I1* define predicates, while *I2* and *I3* are predicated instructions. The effect of a predicate on the instruction is *to validate (allows it to write its result) or invalidate its output*. In cases of load/store instructions, a predicate qualifies memory accesses.

Predication offers many benefits. ILP is increased by allowing separate control paths to be executed in parallel. Some optimizations like modulo scheduling are difficult to be applied on code segments with control-flow. Optimizations like
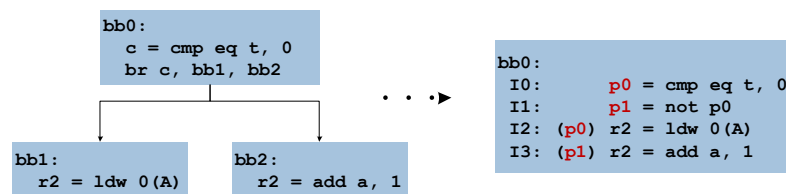
Figure 4.3: IF-Conversion using LLVM assembly. Multiple blocks of instructions are merged into a single block.

60

redundant and dead instructions elimination will be more effective on a code free of control instructions.

4.2.2 Prior Work

For many years, If-conversion studies have been conducted by simulating code generated by experimental compilers. Recently, predicated execution is supported on almost all high performance processors VLIW/EPIC processors. Multiflow-200 architecture [53] provided a *Select* instruction to select a data outcome from multiple control paths. Later Multiflow architectures supported conditional-write for store and floating point instructions [53]. Many architectures adopted conditional move instruction (CMOV) as in DEC/Compaq Alpha and SUN SPARC V9 [55, 57].

Cydra5 was the first architecture that fully supported word-wide instruction predication. Every wide-word instruction can be made conditional on a bit in the predicates register file (Iteration Control Register) [53, 55] Intel IA-64 (Itanium) was the first general purpose architecture that fully supported predication. Each instruction specifies a 1-bit predicate register, and if the value is true the instruction is executed, otherwise, the instruction will have no effect [52, 54]. Predicate registers are set by compare instructions, where each compare instruction is specified with the predicate registers to update.

4.2.3 Predication Algorithm

SOpenCL implements If-conversion as a separate pass in LLVM compiler. If-conversion is used to transform control dependencies in inner-most loops into data dependencies in order to facilitate modulo scheduling and increase ILP.

4.2.3.1 If-conversion algorithm

Algorithm 4.1 depicts the pseudo code of the used If-conversion algorithm. The algorithm first put the blocks of the inner most loop in execution order, i.e. a block comes in the list after all its predecessors. The algorithm then iterates the ordered blocks and for each block it first computes the block predicate using *computeBlockPredicate* function. Then, it process block instructions by replacing *Phi*

instructions, removing *branch* instructions and computing destination blocks partial predicates. At the end, it computes the loop header block predicate.

The *replacePhiInstruction* function replaces a *Phi* instruction in block *Bi* with a sequence of *Select* instructions using partial predicates computed for block *Bi*. A partial predicate is a predicate of block *Bi* generated from only one of its predecessors. Block *Bi* will have as many partial predicates as the number of its predecessor blocks. In Figure 4.4c, *c1* instruction is a partial predicate of block *bb4* corresponding it its predecessor block *bb1*. A *Phi* instruction is replaced by a sequence of *select* instructions each selecting an input data token if its condition (i.e. partial predicate) is true, or the previous data token *select* instruction. In this sequence only a single partial predicate will be true, and so the true data token will be passed. In Figure 4.4c, the *Phi* instruction *r3* is replaced by a sequence of two *select* instructions: *t0* and *r3* in Figure 4.4d.

The *computePartialPredicates* function removes a branch instruction and computes partial predicates of destination blocks using the branch instruction

Algorithm 4.1: If-conversion algorithm.

Input: Inner loop code in LLVM assembly code with multiple instructions blocks.
Output: Inner loop with single block of instructions.

```
 1: BB    → List of Inner loop Blocks
 2: PP    → Blocks Partial Predicates List
 3:
 4: // Main If Conversion algorithm
 5: IfConversion( BB ){
 6:
 7:  BB` = ExecutionOrder(BB);
 8:
 9:  foreach block Bi in BB` do
10:    p = computeBlockPredicate(Bi, PP);
11:    foreach instruction I in block Bi do
12:       if I is Phi instruction then
13:          replacePhiInstruction(I, PP);
14:       else if I is Branch instruction then
15:          computePartialPredicates(p, I, PP);
16:       else
17:          copyInstruction(I);
18:       end if
19:    end for
20:  end for
21:
22:  ComputeHeaderPredicate(PP);
23: }
```

```
for (i=0; i<4; i++) {
    A = interpolate(Ref, i);
    if ( A > 255 )
        A = 255;
    else if( A < 0 )
        A = 0;

    Dest[i] = A;
}
```
(a)

```
pre:
  br bb0
bb0:
  i  = phi (0, pre), (r6, bb5)
  r0 = call(interpolate, Ref, i)
  r1 = icmp sgt r0, 255
  br r1, bb1, bb2
bb1:
  br bb5
bb2:
  r2 = icmp slt r0, 0
  br r2, bb3, bb4
bb3:
  br bb5
bb4:
  br bb5
bb5:
  r3 = phi (255, bb1), (0, bb3), (r0, bb4)
  r4 = getelementptr Dest, i
  store r3, r4
  r5 = icmp eq i, 4
  r6 = add i, 1
  br r5, exit, bb0
exit:
```
(c)

```
pre:
  br bb0
bb0:
      p  = phi (1, pre), c3, bb0)
(p)  i  = phi (0, pre), (r6, bb0)
      r0 = call(interpolate, Ref, i);
      r1 = icmp sgt r0, 255
      c0 = not r1
      r2 = icmp slt r0, 0
      c1 = not r2
      c2 = and c0, c1
      t0 = select r1, 255, 0
      r3 = select c2, r0, t0
      r4 = getelementptr Dest, i
(p)  store r3, r4
      r5 = icmp eq i, 4
      r6 = add i, 1
      c3 = not r5
      br r5, exit, bb0
exit:
```
(d)

(b)

Figure 4.4: If-conversion transformation for value-clipping example. (a) C code interpolator sample. (b) Control flow graph (CFG) of the LLVM code in (c). (c) Generated LLVM assembly code. (d) Predicated LLVM code after applying Algorithm 4.1.

condition operand. For the true destination block, the partial predicate is computed as the AND operation of the branch condition and the predicate of the source block. In Figure 4.4d, c_2 is the partial predicate for *bb4* computed from its source *bb2*. This is the AND operation of the branch condition negation c_1 from source block *bb2* and the predicate c_0 of *bb2*.

The *computeBlockPredicate* function computes the predicate defining instruction of block B_i as a logical OR of all the block partial predicates. In Figure 4.4d, block *bb4* has one source block *bb2* only, hence its partial predicate c_2 is also its predicate instruction. The same applies to block *bb2*, its only partial predicate c_0 is also as its predicate.

Even the loop header block *bb0* is valid at each loop iteration, we introduce the predicate p for the header block. The predicate p takes true value for the first loop iteration and for the rest of loop iterations it takes the negation of the loop exit condition c_3. The header block predicate is necessary for implementing loop termination and schedule flushing. Header predicate instruction is computed in Algorithm 4.1 using *ComputeHeaderPredicate.*

Note that we do not need to replace the *Phi* instruction of the loop header block, because the accelerator architectural template provides special function units to

63

implement such Phi instructions in the loop header block. In Figure 4.4 the initial value is *0*, and the loop carried value is *r6* which is the increment of the loop index.

Another issue to address in if-conversion is the multiple exiting points in the loop. The exit condition represents the predicate for the exit block, the block the loop reaches when it terminates (e.g. block *exit* in Figure 4.4). We compute this predicate as any other predicate by ORing its partial predicates. The example of Figure 4.4 has a single exiting point with a single exit condition (*r5*) which is used as predicate for the *exit* block. If another block in the loop reaches the *exit* block, the predicate of *exit* block is computed as the OR between the two partial predicates.

4.2.3.2 Architectural Support for Predication

The architecture template of Chapter 3 provides support for predicated instructions, by annotating each data token by a valid bit used to indicate whether the token carries valid data or not. This valid bit is used to support predicated execution.

The architectural support we propose is exemplified in Figure 4.5. We only apply predicate-bits (*predicate* signal in Figure 4.5) on a limited set of instructions, such as *phi*, *store*, and *load* instructions, beside instructions that have effects outside the loop. The *predicate* signal in Figure 4.5 is the *predicate* defining instruction of the load operation running on the *load* FU. A false *predicate* signal invalidates the *load* FU output data token. Similarly, a false predicated data token is ignored when it changes memory or output data register as in *store* operations. This is the same effect when a valid-bit equals 0. In the implementation of Figure 4.5, a falsely predicated instruction resets the valid bit of the FU output queue. The effect of invalidating data tokens propagates through the valid bits of each functional unit.
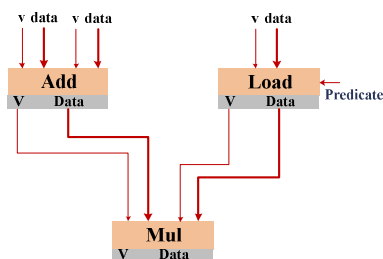
Figure 4.5: Predicated execution architectural support

64

4.3   Code Slicing

4.3.1   Overview

The aim of code slicing is to disassociate computation from data I/O and facilitate their overlap. Decoupled data movement and computations hide memory latency by prefetching data tokens required in later loop iterations while computations performed on early loaded data.

Code slicing has been early introduced by Weiser [60] to facilitate programs debugging. Later it has been used in software analysis and maintenance. According to Weiser's approach, a slice is computed by gathering consecutive sets of indirectly relevant statements, based on data and control dependencies. Two types of slices had been mentioned depending on the traversal direction of a data flow graph; *backward traversal* slices, and *forward traversal* slices. A backward slice consists of all program statements that affect a given statement in the program. A forward slice consists of all program statements that are affected by a given statement. Figure 4.6b shows a backward slice that consists of all statements affecting the statement `write(product)`.

The slice represents a precise portion of the program that produces correct results. Note that multiple backward (and forward) slices of a program will have replicated statements. For example, a backward slice that computes the statement `write(sum)` will include many of the statements appearing in the backward slice of Figure 4.6b.

```
(1)   read(n)                    (1)   read(n)
(2)   i = 1;                     (2)   i = 1;
(3)   sum = 0;                   (3)
(4)   product = 1;               (4)   product = 1;
(5)   while ( i <= n ){          (5)   while ( i <= n ){
(6)     sum = sum + i;           (6)
(7)     product = product * i;   (7)     product = product * i;
(8)     i = i + 1;               (8)     i = i + 1;
      }                                }
(9)   write(sum);                (9)
(10)  write(product);            (10)  write(product);

        (a)                               (b)
```

Figure 4.6: Code slicing. (a) Program Snippet. (b) Backward slice that computes product statement (10). Figure copied from [60].

65

The form shown in Figure 4.6 is known as *static* program slicing, performed statically, when all dependencies in a data flow graph are considered. Dynamic program slicing is a notion used when a program is sliced only according to dependencies occurring in a specific execution of the program.

4.3.2 Slicing Algorithm

SOpenCL implements static backward code slicing in each inner loop of the predicated C kernel as a separate pass in LLVM compiler. Code slicing is used to identify instructions responsible for computing the input (read) addresses in each

Algorithm 4.2: Code slicing algorithm. Output streaming kernel generation is similar to the input streaming kernel, with stores being the instructions of interest.

Input: Inner loop code in LLVM assembly code
Output: Two distinct modified kernels in LLVM assembly code

```
 1:  // Input Streaming Kernel generation
 2:  get_sin_kernel(inner_loop, InstructionList *sin_list){
 3:     sin_list = NULL;
 4:     foreach (instruction It in inner_loop)
 5:         if (It is a load instruction)
 6:                 add(It, sin_list);
 7:
 8:     It = select any instruction from sin_list;
 9:     while (It!= NULL) {
10:         foreach (predecessor(It) != NULL)
11:                 add(predecessor(It), sin_list);
12:         It = select any (predecessor(It)!= NULL);
13:     }
14:
15:     It = select any instruction from sin_list;
16:     while (It!= NULL) {
17:         pred = predicate(It);
18:         if (pred != NULL){
19:             foreach (predecessor(pred) != NULL)
20:                if (sin_list(predecessor(pred)) == NULL){
21:                 pred = NULL; break; }
22:             if(pred != NULL)
23:               add(pred, sin_list);
24:     }}}
25:  //Computational Kernel generation
26:  get_comp_kernel(inner_loop,  InstructionList *sin_list ,
26:                         InstructionList *comp_list){
28:     comp_list = NULL;
29:     foreach (instruction It in inner_loop)
30:         if ( It not in sin_list )
31:             add(It, comp_list);
32:             if (predicate(It)!=NULL)
33:                 add( predicate(It), comp_list);
34:  }
```

inner loop. *Code slicing* step partitions the code to two 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 (*SinAGU* module).

Computational Kernel: This is the core of the PE architecture, 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 the datapath in-order, a *pop* instruction consumes the next element from the input stream without the need to specify a memory address. *Push* instructions produce data to the output stream units in addition to the memory write address. The computational kernel drives the hardware generation of the datapath module.

Algorithm 4.2 depicts the pseudo code of code slicing for Input Streaming kernel and Computational kernel. All load instructions of the inner loop and all their predecessors, i.e. instructions used to compute memory addresses and their control predicates are allocated to the Input streaming units. In the computational kernel, these instructions are substituted by *pop* instructions used to stream data from the input streaming unit to the datapath.

Figure 4.7b depicts a slicing example of a chroma interpolation kernel (the LLVM-IR is shown in Figure 4.7a). The Input streaming kernel comprises all four *load* instructions, their address (*getelementptr* instructions in LLVM assembly), their predicates, and the instructions used to compute their addresses and predicates. In the computational kernel the *load* instructions are converted to *pop* instructions that sink data from input stream channels (*SIN0, SIN1, SIN2* and *SIN3*) without the need to generate address.

The code slicing process is applied only I/O addresses are known at compile time, i.e. they are not dependent on runtime information. Unless this requirement is not satisfied, the AGU cannot run ahead of the datapath since it needs to wait for data computations. In that case, irregular runtime read/write dependencies makes it impossible to pipeline input and output streaming units. As a result, the tool flow will skip code slicing and the unified datapath architecture generated will also be responsible for address generation as well data computations.

67

```
body:
        r34  = phi (1, pre), (r33, body)
 (r34)  i    = phi (0, pre), (r2, body)
        r0   = add a0, i
        gep0 = getelementptr x0, r0
        r2   = add i, 1
        r3   = add a0, i2
        gep1 = getelementptr x0, r3
        r4   = add a1, i
        gep2 = getelementptr x0, r4
        r5   = add a1, i2
        gep3 = getelementptr x0, r5
        r6   = add a2, i
        gep4 = getelementptr x1, r6
 (r34)  r7   = load gep0
        r9   = mul r7, a3
 (r34)  r10  = load gep1
        r12  = mul r10, a4
 (r34)  r13  = load gep2
        r15  = mul r13, a5
 (r34)  r16  = load gep3
        r18  = mul r16, a6
 (r34)  r19  = add r9, 32
        r20  = add r19, r12
        r21  = add r20, r15
        r22  = add r21, r18
        r23  = ashr r22, 6
        r25  = icmp gt r23, 255
        r26  = not r25
        r27  = icmp lt r23, 0
        r28  = not r27
        r29  = and r26, r28
        r30  = select r25, 255, 0
        r31  = select r29, r23, r30
 (r34)  store r31, gep4
        r32  = icmp eq r2, 8
        r33  = xor r32, true
        br r32, exit, body
```
(a)

```
Input Streaming Kernel:

body:
        r34  = phi (1, pre), (r33, body)
 (r34)  i    = phi (0, pre), (r2, body)
        r0   = add a0, i
        gep0 = getelementptr x0, i0
        r2   = add i, 1
        r3   = add a0, i2
        gep1 = getelementptr x0, i3
        r4   = add a1, i
        gep2 = getelementptr x0, i4
        r5   = add a1, i2
        gep3 = getelementptr x0, i5
 (r34)  SIN0 = load gep0
 (r34)  SIN1 = load gep1
 (r34)  SIN2 = load gep2
 (r34)  SIN3 = load gep3
        r32  = icmp eq r2, 8
        r33  = xor r32, true
        br r32, exit, body

Computational Kernel:

body:
        r34  = phi (1, pre), (r33, body)
 (r34)  i    = phi (0, pre), (r2, body)
        r2   = add i, 1
 (r34)  r7   = pop SIN0
        r9   = mul r7, a3
 (r34)  r10  = pop SIN1
        r12  = mul r10, a4
 (r34)  r13  = pop SIN2
        r15  = mul r13, a5
 (r34)  r16  = pop SIN3
        r18  = mul r16, a6
        r19  = add r9, 32
        r20  = add r19, r12
        r21  = add r20, r15
        r22  = add r21, r18
        r23  = ashr r22, 6
        r25  = icmp gt r23, 255
        r26  = not r25
        r27  = icmp lt r23, 0
        r28  = not r27
        r29  = and r26, r28
        r30  = select r25, 255, 0
        r31  = select r29, r23, r30
        r6   = add a2, i
        gep4 = getelementptr x1, r6
 (r34)  store r31, gep4
        r32  = icmp eq r2, 8
        r33  = xor r32, true
        br r32, exit, body
```
(b)

Figure 4.7: Code Slicing. (a) Predicated Chroma Interpolation kernel. (b) Input Streams and Computations code slices. Predicate variable *r34* is used to guard execution of load instructions in the Input Streaming Kernel, and pop and store instructions in the Computational Kernel.

Moreover, if control predicates in the Input Stream kernel are data dependent, the slicing algorithm will bypass adding control predicates to the Input streams kernel and will make *load* instructions always truly predicated. In that case, *load* instructions always generate valid addresses and read.

68

4.4 Instruction Clustering

4.4.1 Overview

One of the most challenging tasks of FPGA design is achieving fully routed circuits, especially in datapath dominated designs. According to our experimental analysis on a set of benchmarks, routing resources, in the form of multiplexers and interconnects occupy 70% to 80% of the design area and account for 90% of the signal delay in computationally intensive designs, such as the LDPC benchmark (described in section 6). Moreover, Placement and Routing (P&R) in modern FPGAs is a very computationally intensive process, even with the use of state-of-the-art routing algorithms. A placement and routing tool may take hours or even days to generate a fully placed and routed design, especially in the presence of routing congestion.

Given the routing complexity for large designs, the pressure is growing for techniques that address the placement and routing problem at a higher abstraction level. In a typical high level synthesis approach, the tasks of resource allocation, scheduling and binding are applied on a set of primitive operations (basic arithmetic and logic operations). The cost of routing resources per primitive functional unit is increasing rapidly in modern FPGAs. For example, the area cost of a 32-bit adder with a 4-input multiplexer on each input port is dominated by the multiplexers tree (67% of the FPGA slices).

Generation of application specific *macro-instructions* is a common practice among instruction-set extensions designers [61, 62, 63, 64]. Such *macro-instructions* can substitute a set of primitive operations and consume fewer resources. Regular computation patterns that appear repetitively in a program DFG are strong candidates to be implemented as macro-instructions. As an example, macro-instruction *K* in Figure 4.8b which consists of two successive additions results into a more compact and efficient circuit, requiring fewer resources (i.e. multiplexers) than the individual primitive ADD operations. A macro-instruction can be designed to optimize a set of different criteria, such as silicon real-estate or latency, compared with the set of corresponding primitive operations.
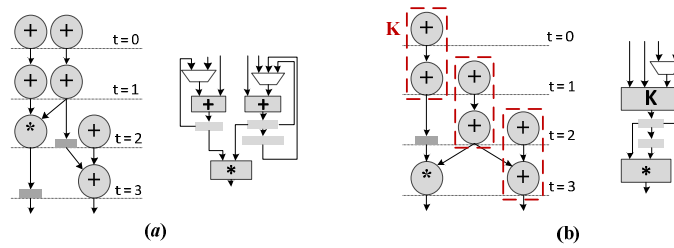
Figure 4.8: Scheduling and binding of a DFG with: (*a*) primitive instructions. (b) Mixture of primitive and macro instruction. Macro instruction *K* is scheduled on the Macro FU (MFU) *K* which is a pipelined 3-input adder.

The generation of application specific macro-instructions is a two steps process: a) candidate instructions identification, and b) candidate instructions selection. During candidate instructions identification, a space exploration of a given DFG results to the identification of a set of subgraphs, of primitive operations, each subgraph representing a potential macro-instruction that fulfills a specific set of constraints. In the next step, a subset of the candidate instructions is selected for the final implementation based on a number of optimality criteria, like latency and area. A variety of approaches have been used for the candidate instructions generation and selection problem, including subgraph enumeration methods and techniques based on pattern recognition [61, 62, 63, 64, 65, 66]. Our target is to exploit the characteristics of MFUs to reduce datapath complexity, and hence, reduce routing overhead and improve performance.

In this work we propose the use of a grammar-induction approach for macro-instructions generation and selection. Grammar induction is an established technique used in string and tree compression algorithms [67, 68]. It is a very efficient approach to extract repetitive patterns from a data sequence and to create hierarchical models of such patterns that can be readily understood, analyzed and applied in other domains. In this paper we extend a grammar induction technique called *Sequitur* [67], to identify and generate a set of candidate macro-instructions. The generated grammar is composed of a set of *non-terminals*, where a non-terminal is a subgraph of the DFG. A non-terminal can, in turn, be composed of other non-terminals and/or primitive operations.

Contrary to the thousands of subgraphs generated by enumeration and pattern recognition methods, the generated grammar has a regular hierarchal structure with

70

few non-terminals, each serving as a potential macro-instruction. This simple hierarchal structure results to a simpler and more compact form of macro-instructions. To keep routing overhead to minimum, a macro functional unit (MFU) closely follows the structure of a single type of macro-instruction (i.e. *non-terminal* in the grammar) and supports the execution of only this type. Making an MFU support the execution of different types of macro-instructions (rules with different subgraphs) requires adding internal configurable multiplexers on the internal edges of the MFU. This, in turn, would come at the expense of complexity and hence would limit the effectiveness of our approach.

One might reason that the reduction of inter-FU interconnects potentially leads to an increase of intra-FU interconnects. However, the increase of intra-FU interconnects does not translate into an area overhead. Intra-FU interconnects are multiplexers free and localized. They are short interconnects between neighboring logic slices. Moreover, intra-FU interconnects can be optimized out using the approach for pipelining MFUs we introduce in section 4.3. In fact, the transformation of costly, inter-FU interconnects into light weight intra-FU interconnects is the main technique exploited by the proposed grammar driven synthesis methodology to reduce area overhead.

4.4.2   Grammar Generation

In this section we introduce a grammar generation algorithm for systematically discovering all repetitive computation patterns inside the DFG, or equivalently identifying candidate sets of primitive operations to be implemented as macro-instructions. Our algorithm is based on the *Sequitur* grammar inference technique,
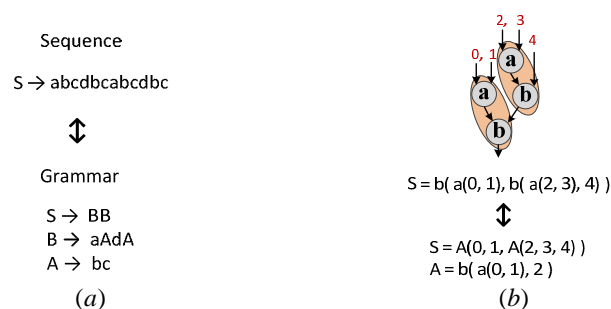
Figure 4.9: Grammar representation applied on (*a*) a sequence of data symbols, and (b) a data flow graph (DFG). Notation *x(y,z)* means that operation *x* has inputs *y* and *z*.

71

originally designed for data strings compression [67].

4.4.2.1 Grammar Representation

Figure 4.9*a* depicts an example of a grammar representation of a sequence of symbols. A grammar representation consists of a set of statements called *rules* or *non-terminals* (we will use both terms interchangeably through the rest of the paper). Each rule is a sequence of symbols that contains other rules and/or data symbols called *terminals*. In Figure 4.9*a,* rule *B* includes both non-terminal symbol *A* and terminal symbols, *a* and *d*. Rule *S* includes non-terminal *B* and rule *A* consists of terminal symbols *b* and *c*. The original statement *S* can be restored by substituting each non-terminal with its production, namely the right-hand side of the rule, until all non-terminals are eliminated.

In this work we extend grammar inductions to also represent data flow graphs. Figure 4.9b depicts a subgraph of a DFG represented as a compound statement *S*. A simple grammar can be deduced by introducing rule *A*. We treat each primitive instruction *a*, and *b* as a *terminal* symbol. A concern in using grammar representations for DFGs is the operand order for non-commutative operations, such as subtraction or division. We use clockwise numbering of input operands to denote their order. In a DFG that consists merely of primitive instructions, each rule can be considered as a potential compound macro-instruction.

A convenient property of grammar representations is their hierarchical structure, which inherently integrates multiple levels of granularity. Such a multi-granular representation of a DFG proves very handy when it comes to hardware implementation of computationally intensive algorithms. For example, assume the DFG subgraph *S* in Figure 4.9b is part of a larger DFG, populated with multiple subgraphs of type *S*. In this case, *S* can function as a *non-terminal* in the larger DFG. The synthesizer has the choice to implement either the macro-instruction *A* that represents a fine granularity computation, or the macro-instruction *S* which represents a coarser granularity computation.

An MFU that implements a macro-instruction with coarser granularity requires lower routing overhead because most interconnects tend to be within the FU, and not across the FUs. By reducing inter-FU routing, final datapath implementation tends to

suffer less from routing congestion and to require lower P&R overhead. However, a coarser granularity macro-instruction like *S* is not necessarily fitter for implementation. This is, for example, the case when the implementation of *S* requires many resources and at the same time there are just a few occurrences of *S* in the program to reuse the MFU that implements *S*. In this case, a finer granularity macro-instruction like *A* which costs less resources and may have many more similar patterns in the program seems to be fitter for implementation. In section 4.4.3 we will introduce a systematic method for selecting between different granularity levels.

4.4.2.2 Generation of Grammar-based DFG representation

The grammar generation algorithm traverses the DFG and discovers repetitive patterns by matching pairs of instructions. A pair of instructions *b*(*a*) denotes that the output of instruction *a* is an operand to instruction *b* as shown in Figure 4.9b. We call instruction *b* destination node and instruction *a* source node. The parenthesis in *b*(*a*)

Algorithm 4.3: Grammar Extraction Algorithm

Input: Data Flow Graph
Output: Set of Grammar Rules

```
 1: D[N]        → Data Flow Graph (DFG) nodes list
 2: N           → Number of DFG nodes
 3: M           → Set of matched node pairs
 4: G           → Grammar's rules set.
 5:
 6: Order D nodes in reverse topological order;
 7:
 8: index = 0;
 9: while (index < N) do
10:   R    = D[index];
11:   Max = 0;
12:   for each operand P of instruction R do
13:      Pair = R(P)
14:      if( ! check_output_ports( Pair ) ) continue;
15:      if( ! check_convexity( Pair ) )    continue;
16:      (Size, Mt) = find_matching_pairs( D, Pair );
17:      If ( Size > Max ) then
18:        M    = Mt
19:        Max = Size
20:      end if
21:   end for
22:   if ( Max > 0 ) then
23:      update_grammar( G, M );
24:      update_destination_nodes( D, M );
25:   else
26:      index += 1;
27:   end if
28: end while
```

is used to express the instruction-operand relationship of instructions *b* and *a.*

The rules of a grammar generated according to *Sequitur* share two properties:

(1) *Digram uniqueness*: A digram is a pair of adjacent symbols, each being a terminal or non-terminal e.g. *aA* in Figure 4.9a. Each digram should appear exactly once in the productions (right-hand side) of the grammar rules.

(2) *Rule Utility*: Each rule in the grammar should appear at least twice in the productions of other, higher-level rules. This property ensures that all rules are useful.

In addition to the above constraints we introduce the following constraints, specifically for data flow graphs:

(1) *Output ports number*: The number of outputs of a compound statement described by a rule S should not exceed an upper limit N_{out}. For N_{out} larger than one, MFU with multiple output ports (e.g. performs multiple computations in parallel) is feasible. This constraint helps reduce the complexity of the pattern identification and selection process by reducing the amount of feasible patterns.

(2) *Convexity*: A rule is a representation of a convex subgraph in the DFG. A subgraph S is convex if there is no path from a node $u \in S$ to a node $v \in S$ through a node $w \notin S$.

(3) *Data computation instructions only*: Load, store, and control instruction nodes cannot be included as terminals in the grammar rules.

Algorithm 4.3 outlines the pseudo code of the grammar generation algorithm and Figure 4.10 shows the steps using a motivational example. The algorithm starts by sorting the DFG nodes in a reverse topological order. In Figure 4.10a, each node is assigned a number indicating its reverse topological order.

Given the sorted DFG, the algorithm selects the first node, *n0* (*destination node*) in our example, and builds the template pairs for each operand of the node (*n0(n2)* and *n0(n3)* in our example). If a template pair satisfies the output ports number and convexity tests, the algorithm searches for additional instances of the template in the DFG, using the subroutine *find_matching_pairs*. The function returns a list M_t of

pairs of instructions matching the template pair.

A matching instance should have the same operations as the template pair and, generally, the same order of operands. The order of operands is ignored in case the destination node in the template pair is a commutative operation such as addition. From all the template pairs derived from *n0*, namely *n0(n2)* (Figure 4.10b) and *n0(n3)* (Figure 4.10c), we greedily choose to consider the template pair with the maximum number of instances for implementation as a macro-instruction. In our example (Figure 4.10d) we chose the template pair *a(b)* (corresponding to *n0(n3)*) which has 5 occurrences rather than the template pair *a(a)* (corresponding to *n0(n2)*) which has 2 occurrences.

When a template pair is chosen, the algorithm will update the grammar using the
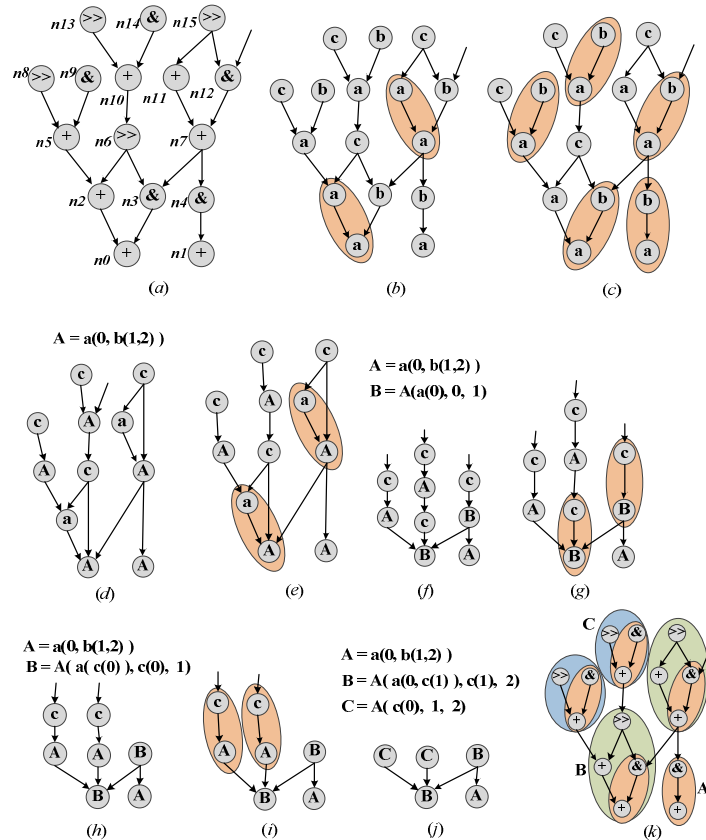
Figure 4.10: Motivational example showing the steps of Algorithm 4.3. In this case output ports number constraint is set to one ($N_{out} = 1$). The final generated grammar is depicted in (*k*).Three potential clusters of instructions can be implemented as a Macro FU.

75

subroutine *updategrammar* in one of two ways:

(1) If the destination node in the pair is a terminal, i.e. a primitive instruction, the algorithm generates a new rule. In Figure 4.10d we create a new rule *A* for the pair *a(b)* because *a* is a primitive operation.

(2) If the destination node in the pair is *non-terminal* (e.g. node *A* in Figure 4.10e), then;

    a. If all its occurrences in the DFG have a matching pair (e.g. *A(a)* in Figure 4.10e), we extend the *non-terminal rule* of the destination node.

    b. Otherwise, we create a new rule.

In Figure 4.10e, not all the occurrences of the destination node *A* have a matching pair *A(a)* (only 2 of the 5 occurrences of *A*), so we create the new rule *B*. However, in Figure 4.10g, all occurrences of the destination node *B* have a matching pair *B(c)*, so we extend the rule of *B* to include *c*.

After updating the grammar, the algorithm updates the destination node in each matching pair using the subroutine *update_destination_nod* as follows:

(1) Substitute the destination node of each matching pair by a *non-terminal* node. E.g. node *a* in the pair *a(b)* of Figure 4.10c becomes non-terminal node *A* in Figure 4.10d.

(2) Add the source node in the pair (*b* in the pair *a(b)* of Figure 4.10c) to the internal subgraph of the destination node. Each node marked as *non-terminal* has an internal subgraph which is a cut of the original DFG. In Figure 4.10d, non-terminal node *A* corresponds to subgraph *a(0, b(1, 2))*.

(3) Finally, the algorithm updates the operands list of the newly created non-terminal node to include the operands of the source node in the pair, and empties the operands list of the source node.

The process is repeated on the new state of the DFG, searching for templates (pairs of nodes) having the newly inserted non-terminal as destination. In Figure 4.10e, after merging terminal node *a* to non-terminal node *A*, the algorithm repeats the process of building template pairs and searching for matches using destination node *A* which now has two more operands: *c* and *A*, to node *b*. If the algorithm fails to

76
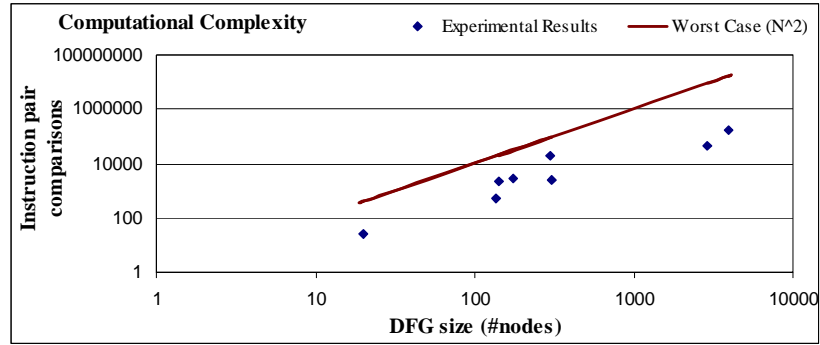
Figure 4.11: Experimental evaluation of the computational complexity of Algorithm 4.3. The data points represent the number of instruction pair comparisons observed experimentally on the benchmarks set of Table IV. The theoretically predicted worst case complexity is also depicted in the graph (continuous line). Both the x- and y-axis are in logarithmic scale.

find matching pairs having the newly inserted non-terminal as destination node, it continues with the next node in the sorted DFG list. The iterative process continues until there are no more nodes to consider as destination nodes.

4.4.2.3 Computational Complexity and Correctness

For a DFG with N nodes and E edges, the grammar generation algorithm computational complexity in the worst case scenario (where the DFG has no repetitive patterns) is $O(N^2)$. The computational complexity for the worst case scenario can be derived as follows:

(1) Each edge e_i in the DFG is compared with each other edge e_j in the DFG where $i \neq j$. Hence, the maximum number of search steps is $E*(E-1)$, in the case no patterns are detected. Otherwise, each time a pattern instance is substituted by a macro instruction, the total number of edges in the DFG is reduced by at least 2 (at least 2 instances of the pattern, involving at least 2 edges, are substituted by macro nodes), and the total number of search steps is reduced accordingly.

(2) For a DFG without recurrent circuits and, the total number of edges E in the DFG is linearly dependent on the number of nodes N, hence the maximum number of search steps is $O(N^2)$.

Figure 4.11 depicts the computational complexity (in terms of the total number of instruction pair comparisons) observed experimentally by applying the algorithm on the benchmark base used in the experimental evaluation (Section 5). Their

77

characteristics are summarized in Table IV. The graph also includes a plot of $f(N) = N^2$ (Worst Case). It is clear that in all cases, the overhead of the algorithm is lower than the $O(N^2)$ worst-case complexity. In fact in practice the worst-case upper bound proves overly conservative.

The computational complexity of the algorithm is significantly lower than that of enumeration based algorithms, which are characterized by exponential complexity. For all experiments described in Section 5, the execution time of the algorithm was less than 1 second. Moreover, the significant reduction in synthesis, placement & routing runtime for large values of N in the vast majority of the experiments overweighs the grammar generation runtime overhead, leading to overall reduction in the design generation runtime.

The algorithm does not remove DFG nodes, not even reorganize them. It just groups them together without changing their external or internal connections in the DFG, so essentially, the original and the compressed DFGs are equivalent. Therefore, the algorithm is correct.

4.4.3 Grammar-Driven Datapath Synthesis Flow

The hierarchical grammar representation of a DFG can be exploited in many practical problems such as DFG compression. Since each FU in a datapath can be typically reused for multiple DFG operations, a multiplexer tree is needed at the input ports of each FU to select among a multitude of inputs. Multiplexer trees may cost more in terms of area than the FU itself, specifically for simpler FUs that perform basic arithmetic and logic operations. For example, a 2-input 32-bit multiplexer consumes as many FPGA logic cells as a 2-input adder or a logic operator of the same bitwidth. Therefore, if a 2-input adder is driven by an 8-input multiplexer tree at each of its inputs, the cost of the adder will be smaller than the cost of the multiplexer tree. If all instances of a grammar rule are implemented as a macro functional unit (MFU), where the internal data flows are free of multiplexers, the area gain may be significant; furthermore, reducing routing complexity leads to reducing routing latency, and time the P&R tool chain requires to place and route the design.

78
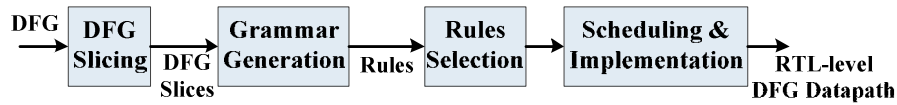
Figure 4.12: Grammar based datapath synthesis flow.

Figure 4.12 shows the complete grammar-driven datapath synthesis flow, including instruction clustering. For each input DFG we generate the datapath RTL that implements the DFG functionality. Given the original input DFG, the synthesis flow starts by slicing the DFG into one or more smaller subgraphs. Then, the grammar generation engine processes each DFG slice separately and generates the grammar. A subset of the non-terminal rules is selected to generate macro-instructions. Given the selected set of rules, the algorithm will produce a new DFG incorporating primitive instructions and macro-instructions.

4.4.3.1 Data Flow Graph Slicing

A preliminary step before grammar generation in our tool is the slicing of the given DFG into smaller DFGs (Figure 4.13). In some cases, for example when the DFG expresses computation of an unrolled, data-parallel loop, the graph consists of multiple strongly connected subgraphs (slices), each corresponding to a loop iteration. The objective of DFG slicing is to treat parallel data flows within a DFG independently in grammar generation, scheduling and binding. For grammar generation, the search space for matching pairs is smaller when applied on DFG slices rather than the original DFG, which will speed up the grammar generation algorithm. Another important benefit is the creation of isolated islands of resources (FUs, registers) by preventing an instruction in a DFG slice from being scheduled on resources of another DFG slice. These isolated islands of resources make the task of the placement & routing much easier.

DFG slicing corresponds to identifying the strongly connected components of the DFG. We use a modified version of the path-based strong component algorithm described by Cheriyan and Mehlhorn [69]. Starting from each leaf node of the original DFG, *n0*, *n1*, and *n2* in Figure 4.13, the slicing algorithm iteratively moves up the graph and tracks the operand nodes of each selected node. At first, both DFG slices A and B of Figure 4.13 include two common nodes: *c0* and *c1* (Figure 4.12).

79
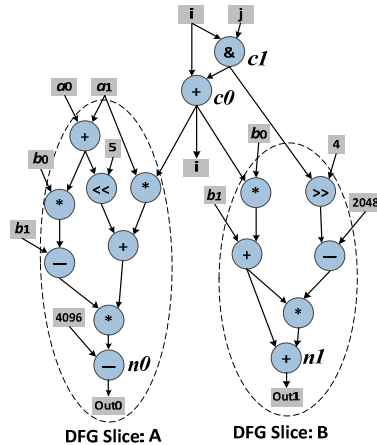
Figure 4.13: DFG slicing example. The original DFG is partitioned into two independent slices.

Since the two slices are strongly connected, we exclude the two common nodes from both slices. A slice of a DFG is created from each leaf node and DFGs with no common nodes belong to different slices.

4.4.3.2 Grammar Generation & Selection

Following DFG slicing, the flow continues with the grammar generation algorithm described in Section 4.4.2, which is applied independently on each slice. Hence, each DFG slice will end up with its own grammar representation.

Grammar-driven data compression algorithms normally use all the grammar rules to compress a sequence of data symbols. However, in our case, a subset of rules can be used to implement MFUs. As mentioned earlier, grammar rules correspond to candidate macro-instructions – which can be implemented as custom MFUs – at different granularities. Therefore, the synthesizer needs to select the optimal granularity for the generation of macro-instructions, according to a set of criteria.

The purpose of this step is to identify an optimal subset of grammar rules that minimizes routing density and reduces total area. Algorithm 4.4 summarizes the greedy selection heuristic we introduce in our work. The selection heuristic uses a fitness function to assign weights to each rule in the generated grammar. At each step, the rule with the highest fitness value is selected to be implemented as an MFU and all instances of the selected rule are removed from the grammar. Note that when a rule is selected, all grammar rules using this rule as a non-terminal in their

80

productions are essentially also removed from the grammar and they are no longer considered for implementation as MFUs. Otherwise, multiple different MFUs would be generated, executing the same primitive operations. After each step, the fitness function updates the fitness of the remaining rules. The process is repeated until the grammar is empty.

The fitness function (1) uses a set of metrics to estimate the gain from implementing rule i as an MFU. The metrics aim to rank the grammar rules based on their potential to reduce routing complexity:

$$W_i = CG_i * \left(LG_i + MUXG_i \right)$$

(1)

The following paragraphs detail the parameters of (1).

Coverage Gain (CG): The coverage gain for rule i is a normalized value of the total number of primitive instructions in the DFG covered by the specific rule. The metric is computed in (2). Higher coverage of the DFG nodes means fewer primitive FUs will be implemented individually, hence, smaller multiplexer trees. To compute a

---

Algorithm 4.4: Grammar Rules Selection

Input: List of Grammar Rules.
Output: Select set of grammar rules.

```
 1: G          → set of discovered Rules
 2: SR         → Selected set of Rules
 3: r_size     → Rule instances count
 4:
 5: computeMetrics( G, BWA );
 6:
 7: while ( G != ∅ ) do
 8:   OrderRules( G );
 9:   R = getMaxFitnessRule( G );
10:
11:   if ( R. r_size > 2 ) then
12:     add R to SR;
13:     foreach Rule S_k != R do
14:       if S_k uses R as non-terminal then
15:         remove S_k from G
16:       else if R uses S_k as non-terminal then
17:         remove all instances of S_k in R from G
18:       end if
19:     end for
20:   end if
21:   remove R from G;
22:   computeMetrics( G );
23:
24: end while
```

81

fair metric value, we compute the total number of primitive instructions that can be covered by a given rule, instead of relying only on the count of rule instances (*occurrences*) or the number of primitive instructions (*operations*) per rule instance.

$$Coverage_i = \left(\#Occurrences_i * \#Operations_i\right),$$
$$CG_i = \frac{Coverage_i}{\max\limits_{0 \le i \le RulesCount}\left(Coverage_i\right)} \tag{2}$$

The coverage gain factor functions as a multiplier for two metrics *LG* and *MUXG* that correspond to area gains. It is important to notice that the value of the coverage gain metric will change each time we select a rule to be implemented as an MFU. This happens because some of the rule instances are removed from the grammar if they appear as non-terminals in the production of a rule selected earlier. Also the current maximum coverage value will change, and hence, the normalized values of CG.

Multiplexers Gain (MUXG): This metric quantifies area gains due to reduction of number of multiplexers per instance of each *rule*. The metric is computed using (3). The nominator in (3) is the difference between the total number of inputs of all primitive FUs of an MFU ($\Sigma\#Operands$) and the number of the MFU inputs ($\#RuleOperands$). To quantify the gain from this difference, we divide it by "$\Sigma\#Operands$".

$$Ratio_i = \frac{\sum\limits_{p=0}^{RuleOps}\#Operands_p - \#RuleOperands}{\sum\limits_{p=0}^{RuleOps}\#Operands_p}$$
$$MUXG_i = \frac{Ratio_i}{\max\limits_{0 \le p \le RulesCount}\left(Ratio_p\right)} \tag{3}$$

Based on formula (3), we can find that the value of *MUXG* tends to increase when the number of primitive instructions in a rule increases. In other words, larger rules will have higher multiplexers gain. However, the algorithm does not always favor larger rules over smaller ones. A smaller rule with lower multiplexers gain per instance may be associated with a much higher coverage gain which makes it fitter for implementation.

82

Logic Gain (LG_i): This metric quantifies the potential for reduction of logic cells through packing of primitive instructions within an MFU (or equivalently a grammar rule). The metric is computed using equation (4). Consider an MFU implementing the function $f(x_0, x_1 ... x_n)$. The nominator in (4) quantifies the efficacy of fusing the logic cells of all the primitive FUs of the MFU. LUTs in FPGAs (an LUT serves as a function generator with limited number of inputs) have a limited number of inputs, hence, the more the number of MFU inputs increases the more difficult it becomes to map its function on fewer LUTs, and therefore, we divide by the number of the MFU input signals (*#RuleOperands*) in equation (4).

$$LogicGain_i = \frac{\sum_{l=0}^{RuleOps}(1 - A_l)}{\# RuleOperands}$$

$$LG_i = \frac{LogicGain_i}{\max_{0 \leq p \leq RulesCount}(LogicGain_p)}$$

(4)

The value of the parameter A_l in (4) is normalized in the range [0, 1] and is characteristic for each primitive instruction type *l*. It quantifies the difficulty to fuse this instruction with additional ones, in the same set of logic cells. A_l is dependent on the nature of the instruction, the FPGA architecture, and the synthesis, placement and routing tool chain. We developed a set of representative subgraphs, with various primitive instructions types and configurations, which can be used as micro-benchmarks for systematically estimating A_l on each target platform. A micro-benchmark is a subgraph synthesized to analyze primitive FUs resources requirements. Subgraphs *A*, *B*, and *C* in Figure 4.10 are examples of micro-benchmarks. This approach is described in detail in Section 4.4.3.3. For the Xilinx Virtex 6 FPGA family for example, the characterization assigned the A_l value 0.5 to *add* operations, whereas logical *and* operations have an A_l value of 0.20. The *shift* operation was assigned an A_l value of 1.0 indicating that its logic cells cannot accommodate additional operations, when the shift amount is variable.

Figure 4.14 shows how we apply rule selection on the grammar of the example of Figure 4.10. The left table of Figure 4.14 contains the normalized metric parameters and the corresponding fitness for each rule according to (1). After selecting the rule with the maximum fitness (*B* in Figure 4.10), we update the metric parameters, and

83

	CG	MUXG	LG	W		CG	MUXG	LG	W
A	1	0.5	0.95	**1.45**	A	1	0.75	1	**1.75**
B	**0.8**	1	1	**1.6**	C	1	1	0.75	**1.75**
C	0.6	0.67	0.72	**0.83**					

Select Rule **B** ⟶   Update Metrics ⟶   Select Rule **C** ⟶

Figure 4.14: The selection process of Rules in the grammar of Figure 4.10. The selected set of rules: {**B**, **C**}.

normalize their values again. Note that after removing rule *B* from the grammar, we also removed two instances of rule *A*, which appears now in only 3 instances. Rules *A* and *C* now have the same coverage since they both cover 6 instructions. After updating the metrics (right table of Figure 4.14), both rules *A* and *C* have the same weight. Since rule *C* is using rule *A*, *OrderRules* subroutine prioritizes rule *C* over rule *A*, and hence the algorithm selects rule *C* for implementation and removes 2 more instances of the rule *A*. Since rule *A* now appears in only one instance, we can no longer consider it for MFU implementation, because of the *rule utility* constraint: each rule must appear in the grammar with at least two instances.

4.4.3.3 Macro Functional Unit Pipelining

Since MFUs have a more complex structure than simple FUs, it is possible that they will stretch clock frequency if they are assigned a single cycle for execution. Prior to scheduling macro-instructions on the generated MFUs, we have to determine the pipeline depth of each MFU and therefore its cycle latency, aiming at retaining the same clock frequency as if we had no MFUs in the accelerator. Algorithm 4.5 drives the decision process of inserting pipeline registers between pairs of primitive FUs in a given MFU. The algorithm attempts to balance timing delay by placing FUs
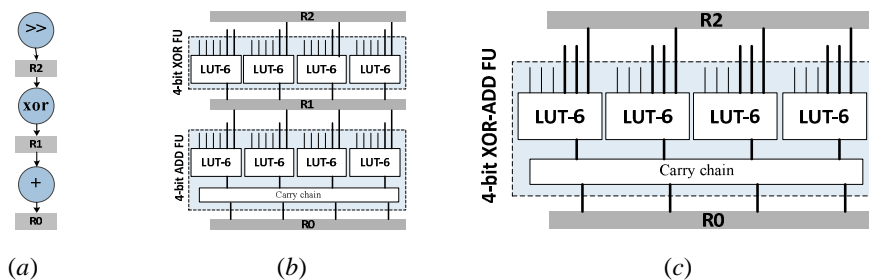
Figure 4.15: (*a*) reference pipeline scheme used as template for the pipelining algorithm. (*b*) Logic level of pipelined Xor and Add operators. (*c*) Fused Xor and Add operations logic level.

84

of approximately equal latency in each pipeline region.

The algorithm uses as a reference a default pipelining scheme for inserting pipeline registers in MFU. The default pipelining scheme blindly adds a pipeline register after each primitive FU, as in the case when primitive FUs implemented individually and not part of an MFU (Figure 4.15a). In this reference pipeline scheme, *the combinational path of a single primitive FU* (4-bit XOR and 4-bit ADD FUs in Figure 4.15b) is considered as one level of logic. Hence, using the default pipelining scheme, only one level of logic exists between two successive pipeline registers.

Algorithm 4.5 traverses the MFU subgraph and removes a pipeline register if its removal doesn't increase the levels of logic between two other pipeline registers. For example, in Figure 4.15a, pipeline register *R1* will be removed if it does not increase the levels of logic between pipeline registers *R0* and *R2*. Contrary to the intuition, the removal of a pipeline register doesn't necessarily increase the levels of logic on a combinational path between two registers on an FPGA. For example, in Figure 4.15c, the removal of pipeline register *R1* allowed fusing the logic cells of the XOR FU with the logic cells of the ADD FU. The removal of a pipeline register*R1* produces a new boolean expression that may be implementable using one level of logic cells (LUTs). In most cases, a primitive FU does not consume the whole capacity of its LUTs.

To determine if the removal of a pipeline register will increase the number of logic

---

Algorithm 4.5: Custom Instruction Pipelining

Input: Custom instruction subgraph.

Output: Pipelined Macro Functional Unit.

```
 1: N            → Rule's primitive operations
 2:
 3: reverse_topological_order(N)
 4: foreach node N₁ in N do
 5:    max = 0
 6:    foreach user Uₖ of Node N₁ do
 7:      if ( heights[k] > max ) then
 8:        max = heights[k]
 9:      end if
10:    end for
11:    if ( (max + A₁) < 1.0 ) then
12:      remove_pipeline_register( N₁ )
13:      heights[l] =  Aₖ
14:    else
15:      heights[l] =  max + Aₖ
16:    end if
17: end for
```

levels – in the form of LUTs –, algorithm 4.5 uses the same set of A_l parameters used in (4) to compute the logic gain metric LG_i. Parameter A_l quantifies an estimation of the percentage of the implementation capacity of the LUT taken by the primitive instruction l. Similarly, if two primitive instructions l and l' are fused on the same LUT, the summation of the corresponding area estimation parameters A_l and $A_{l'}$ provides a good estimation of the consumption of the LUT implementation capacity by both instructions.

In general, if the summation of area estimation parameters A_l in a DFG sub-path, is less than or equal to 1.0, we estimate that the corresponding primitive instructions can be fused and implemented on a single LUT, or equivalently, they require the same levels of logic as one primitive instruction. As a result, intermediate registers in the sub-path can be removed without affecting the timing characteristics of the circuit.

The value of the parameter A_l for each primitive instruction is derived by systematically applying an experimental method on a set of micro-benchmarks. The following subsection describes in details the experimental method we introduce.

The pipelining algorithm (Algorithm 4.5) is characterized by linear ($O(N)$) computational complexity for a single MFU type, with respect to the number of primitive FUs (N) in the MFU. For each FU node in the DFG, the algorithm examines one or more output edges (user node U_k in Algorithm 4.5). Since the maximum number of FU operands is 3 (for the select FU), the average number of output edges per node in the MFU graph is a constant, independent of N. Therefore, the total number of edges in the MFU is $O(N)$ and the computational complexity of the algorithm is $O(N)$ as well.

Algorithm 4.5 is essentially a heuristic that could potentially lead to timing errors if applied alone. However, the Xilinx toolchain, responsible for Synthesis, Placement and Routing, guarantees timing correctness by appropriately manipulating frequency. In Section 5 we present the experimental timing evaluation (Table 4.3) on a set of microbenchmarks (Figure 4.16) using both full and selective pipelining. Moreover, we present (Table 5.6) the frequency attained by the Xilinx toolchain on a set of kernels optimized using our approach. Both sets of experimental results prove that Algorithm 3 works efficiently.

4.4.3.3.1   Experimental Area Estimation.

The experimental method incrementally builds sets of micro-benchmarks, computes an initial estimate of the parameter A_l, and refines the initial estimations at a subsequent step. Algorithm 4.6 describes the steps of the experimental method.

Algorithm 4.6: A_l parameters estimation.

Input: Set of micro benchmarks.
Output: A_l parameters estimated values.

```
 1: N        → Primitive Operations Population.
 2: Âₗ       → The value of parameter Aₗ plus an error δₗ
 3: FUₗ      → Primitive FU performs only operations of type l
 4: MFU      → Macro FU composed of one or more primitive FUs
 5:
 6: // Step 1: Initial estimate of parameter Aₗ
 7: foreach primitive operation Nₗ in N do
 8:    Count  = 0
 9:    MFU    = FUₗ
10:    Lₗ       = getAreaLUTs( MFU )
11:    L_mfu  = Lₗ
12:    while ( L_mfu ≤  Lₗ ) do
13:      MFU   = addNewFU( MFU, FUₗ )
14:      L_mfu = getAreaLUTs( MFU )
15:      Count  += 1
16:    end while
17:    Âₗ = 1 / Count
18: end for
19:
20:  Order primitive operations in N from min to max Âₗ
21: // Step 2: Refine initial estimate of parameter Aₗ
22: foreach primitive operation Nₗ in N do
23:   MFU  = FUₗ
24:   Lₗ       = getAreaLUTs( MFU )
25:   foreach operation Nₖ in N where k less than l do
26:      if  Aₖ  < Âₗ    then
27:         Count   = 0
28:         MFU     = addNewFU( MFU, FUₖ )
29:         L_mfu   = getAreaLUTs( MFU )
30:         while ( L_mfu ≤  Lₗ ) do
31:           MFU   = addNewFU( MFU, FUₗ )
32:           L_mfu = getAreaLUTs( MFU )
33:           Count  += 1
34:         end while
35:         if   ( Count × Aₖ  + Âₗ ) > 1    then
36:            δₗ = Count × Aₖ  + Âₗ − 1
37:            Âₗ = Âₗ − δₗ
38:         end if
39:      end if
40:    end for
41:    Aₗ  = Âₗ
42: end for
```

Table 4.1: Experimentally derived values of the A_l parameter for primitive operations for Xilinx Virtex-6 and Virtex-4 FPGA families.

	And, Or, Xor, Not	Select	Add, Sub, Cmp	Mul, Div, Shift, FP operations
Virtex 6	0.2	0.4	0.5	1.0
Virtex 4	0.33	0.67	0.67	1.0

The initial estimate of A_l is computed by determining how many primitive FUs of the same type l can be packed in one level of logic of the same LUTs. The procedure *getAreaLUTs* performs synthesis, placement and routing on the given FU (or MFU) and returns the number of consumed LUTs (the combinational logic cells). The procedure *addNewFU* adds the given FU_l to the subgraph of the given MFU. The process of adding more FUs of the same type continues, until the resulting subgraph requires more LUTs for its implementation than the single, primitive LUT.

The initial estimate is a rough approximation that represents an upper bound for A_l. For example, for an addition operation, two adders can be packed in the same number of LUTs required for the implementation of one adder of the same bitwidth. If a third adder is added, it will occupy a different set of LUTs. Therefore, the initial estimate of A_{add} takes the value 0.5. If packing a third adder on the same set of LUTs succeeded, the estimate would be 0.33. Therefore, the real, accurate value of A_{add} has range [0.5, 0.33).

Given the computed initial estimates of parameters A_l, the algorithm performs a refinement step which attempts to reduce the range of error in the initial estimate. The second step refines the parameter A_l for primitive operation of type l by computing how many primitive operations of type k, with $A_k < A_l$, can be packed in the same LUTs already occupied by operation l. If the summation of parameters A_k and A_l of all successfully packed operations is larger than one, we conclude that the value of parameter A_l is over-estimated and needs to be reduced to approximate the real value.

The reason why we reduce the value of A_l not A_k is because the error in the value of A_k is smaller than that in A_l. Note that the algorithm refines operations with smaller A_l before others with larger A_l. This means the error in A_k has been already refined to approximate its real value before using it to refine a larger A_l. For example, from the parameter A_{add} value range the error is up to 0.17. On the other hand, for Bitwise logic operations the value range of parameter A_{logic} is [0.2, 0.17), and hence the error span

for A_{logic} equals 0.03 which is much smaller than that of A_{add}.

Table 4.1 shows the values of parameter A_l on two Xilinx FPGA architectures; Virtex-6 based on 6-input LUTs architecture, and Virtex-4 based on 4-input LUTs architecture. Figure 4.16 depicts a subset of the micro-benchmarks generated and tested using algorithm 4.6 (all FUs are 16-bit wide). Figure 4.16a corresponds to the reference fully pipelined configurations. Pipelined configurations according to algorithm 4.5 for Virtex-6 and Virtex-4 appear in Figure 4.16b and Figure 4.16c respectively.

Table 4.2 summarizes the consumed LUTs for each micro-benchmark (Figure 4.16a) when all pipeline registers are removed. The *Output FU* column in Table II refers to the area of the output FU (the one directly producing the output data) in the micro-benchmark subgraph: in the examples of Figure 4.16 this is the *Adder* FU for micro-benchmarks 1 & 2, and the *Select* FU for the rest.

The PR-Free Configuration column reports the LUTs required for the implementation of the full set of FUs in the micro-benchmark, whereas ΣAl is the sum of the Al area estimation parameters of all FUs participating in the benchmark. The results in the table are a testament of the accuracy of our area estimation approach, even after one step of refinement. A quick summation of the Al parameters is an excellent predictor of the area that will be required for the implementation of the compound instruction. Whenever ΣA_l exceeds 1.0, an additional set of LUTs will be required to implement the set of FUs. For example, micro-benchmark #1 has a ΣAl of

Table 4.2: Examples of the area (number of LUTs) consumed by a set of micro-benchmarks. All primitive operations are 16-bits wide. We use the notation introduced in section 4.4.2 to describe the micro-benchamrks. PR refers to Pipeline Register.

#	Micro-Benchmarks	Virtex-6			Virtex-4		
		Output FU (one instance)	PR-Free Configuration	ΣA_l	Output FU (one instance)	PR-Free Configuration	ΣA_l
1	*Add* (*Add* (0, 1), 2)	16	16	1.0	16	30	1.34
2	*Add* (*Add* (*Add* (0, 1), 2), 3)	16	32	1.5	16	46	2.01
3	*Sel* (0, *Sel* (1, 2, 3) , 4)	16	16	0.8	16	32	1.34
4	*Sel* (0, *Sel* (1, 2, 3) , *XOR*(4, 5))	16	16	1.0	16	32	1.67
5	*Sel* (0, 1 , *XOR*(2, 3))	16	16	0.6	16	16	1.0

1.0 for Virtex-6, hence our estimator predicts that it will fit in the same set of LUTs as a single Add operation.

The prediction is confirmed by the experiment. Moreover, if we implement the compound statement as a macro FU, we do not need to insert a pipeline register between the adders. However, for Virtex-4 ΣA_l equals 1.34, meaning the adders cannot be fused to a single level of LUTs (as again confirmed by the experiment). Therefore, if we decide to implement the compound statement as a macro FU, we will have to insert a pipeline register between the adders. In micro-benchmark #2 for Virtex-6, the third adder increases the summation of parameters A_l to 1.5 and hence we have to insert a pipeline register after the second adder.

The same can be seen in the other benchmarks. In micro-benchmark #4, the summation on the *Select-Select* path equals 1.34 for Virtex-4, so we do add a pipeline register. However, on the *Select-Xor* path the summation equals 1.0, so no pipeline register is not inserted. Observe also the case of benchmark #2 for Virtex-4: The ΣA_l marker has a value above 2.0. This indicates that even a second set of LUTs will not be enough, and a third set will be needed. The prediction is, once again, confirmed by the experimental results.

In Table 4.3 we compare the critical path delay of the reference fully pipelined micro-benchmarks (Figure 4.15a), with selectively pipelined configurations generated using Algorithm 4.5. In general, pipelined configurations according to our approach
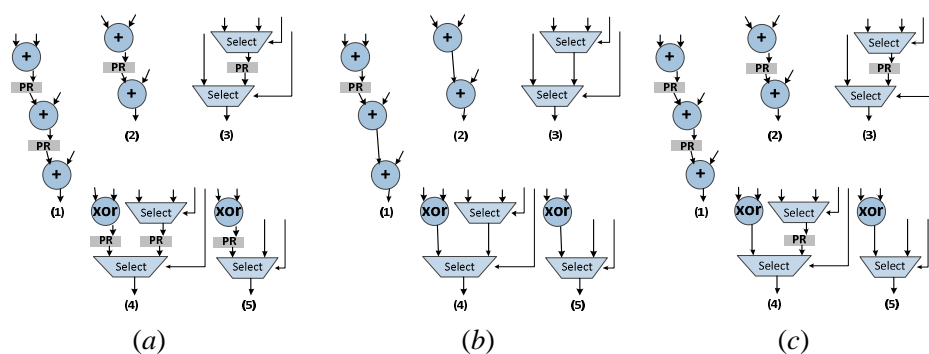
Figure 4.16: Experimental method micro-benchmarks. PR refers to Pipeline Register. (*a*) Fully pipelined configurations. (b) Configurations pipelined according to Algorithm 4.5 for Virtex-6 FPGA. (c) Configurations pipelined according to Algorithm 4.5 for Virtex-4 FPGA.

90

have a slightly longer critical path with very little effect on the clock frequency in the context of a large datapath. The critical path delay is composed of logic and route delay between the registers of inputs and outputs ports. Our analysis of the critical path delay, showed that the logic delay is the same, and the slight overhead comes from route delay. This can be expected, since when fusing two operations, more inputs are brought to the same LUTs, which may increase slightly the route delay of the farthest input source. Once again, the results of Table 4.3 are indicators of the accuracy of the automated, experimental area estimation approach we use as input to the selective pipeline registers insertion algorithm.

4.4.3.4 Scheduling and Implementation

Once a set of rules is selected for MFU implementation, each instance of a rule is converted to a macro instruction of the specific type. Each macro instruction type will be bounded to its own macro FU (MFU latency is computed after applying the pipelining algorithm described in Section 4.4.3.3.

After macro-instruction formation, the resulting DFG is scheduled using modulo scheduling. A macro instruction is scheduled only when all input data are available, so that the functionality and internal organization of MFUs does not need to be known to the scheduling algorithm. For example in Figure 4.10, when scheduling the macro instruction represented by rule *B*, all three input operands should be available. We use *Swing Modulo Scheduling (SMS)* to generate a schedule of the DFG nodes, as will be detailed in Section 4.5.

Table 4.3: Examples of some micro-benchmarks critical path (ns) for two cases: Fully pipelined configuration Figure 4.16a, and a configuration selectively pipelined using algorithm 4.5 (Figure 4.16b and Figure 4.16c for Virtex 6 and Virtex-4 respectively). All primitive FUs are 16-bits wide.

#	Micro-Benchmarks	Virtex-6		Virtex-4	
		Full-Pipelining	Selective Pipelining	Full-Pipelining	Selective Pipelining
1	*Add* (*Add* (0, 1), 2)	2.324	2.720	2.771	2.771
2	*Add* (*Add* (*Add* (0, 1), 2), 3)	2.460	2.770	2.766	2.766
3	*Sel* (0, *Sel* (1, 2, 3) , 4)	1.479	1.580	1.596	1.596
4	*Sel* (0, *Sel* (1, 2, 3) , *XO*R(4, 5))	1.570	1.740	1.669	1.709
5	*Sel* (0, 1 , *XOR*(2, 3))	1.523	1.562	1.650	1.661

91

4.5   Scheduling

Our infrastructure applies two types of scheduling algorithms: a modulo scheduling algorithm called Swing Modulo Scheduling (SMS) [87] which is applied on datapath and AGU kernels (input streams kernel and computational kernel), and As Soon As Possible (ASAP) scheduling [89] applied on basic blocks assigned to the CE modules.

Scheduling techniques are machine dependent algorithms. Scheduling instructions on the datapath or AGU requires first allocating a number of functional units (FUs) before scheduling applied. The amounts and types of functional units in each AGU and datapath are passed as an XML-based file representation specified by the user.

4.5.1   Modulo Scheduling

4.5.1.1 Overview

Modulo scheduling is a software pipelining technique typically applied for pipelining loop iterations. Software pipelining on loops overlaps the execution of successive iterations to increase throughput and to reduce the total execution time. A modulo scheduler produces a schedule for one iteration of the loop (after several unrolls if required), such that when this same schedule is repeatedly applied at regular intervals, no intra- or inter-iteration dependence is violated, and no resource usage conflicts arise between operations of either the same or distinct iterations. This constant interval between successive iterations is called the initiation interval (II).

A modulo schedule of a single iteration is divided into stages with stages' count recorded as *SC* [88]; each stage has a duration equal to the initiation interval. Successive iterations of the loop are initiated after each stage finishes or after II time slots. Figure 4.17 shows a modulo-schedule of a loop with 10 iterations and an II equal to 3. A schedule of a single iteration spans 4 stages. The full loop execution flow consists of three phases: prologue execution, kernel execution, and epilogue execution. The *prologue* represents a transient phase from the beginning of loop execution until all hardware resources become active. The *kernel* phase represents a steady state in the loop execution flow, which in Figure 4.17a, takes place when the fourth iteration is initiated. In steady state all resources are fully utilized by

92

instructions of different loop iterations. The number in the brackets indicates the loop iteration the instruction belongs to. The kernel pattern will repeat (Figure 4.17b) until no more loop iterations are launched. Then the *epilogue* phase begins, which gradually drains the pipeline.

The following steps summarize a generic algorithm to generate a modulo schedule. The next section describes SMS, the specific modulo scheduler used for SOpenCL.

1. Calculate a minimum II bound called MII. The minimum initiation interval (MII) is a lower-bound on the number of cycles required by any feasible schedule of the loop body.

2. Put the instructions population of a loop iteration in an ordered list.

3. Perform scheduling by picking instructions from the ordered list sequentially. Insert instructions in a free time slot in the partial schedule. If the partial schedule fails to accommodate more instructions, increment II and restart scheduling.

The computation of MII is not always adequate for correctness of the schedule, but to avoid trying II that is too small to succeed, thereby speeding-up the modulo scheduling process [87, 88]. MII is computed as the maximum of two parameters;
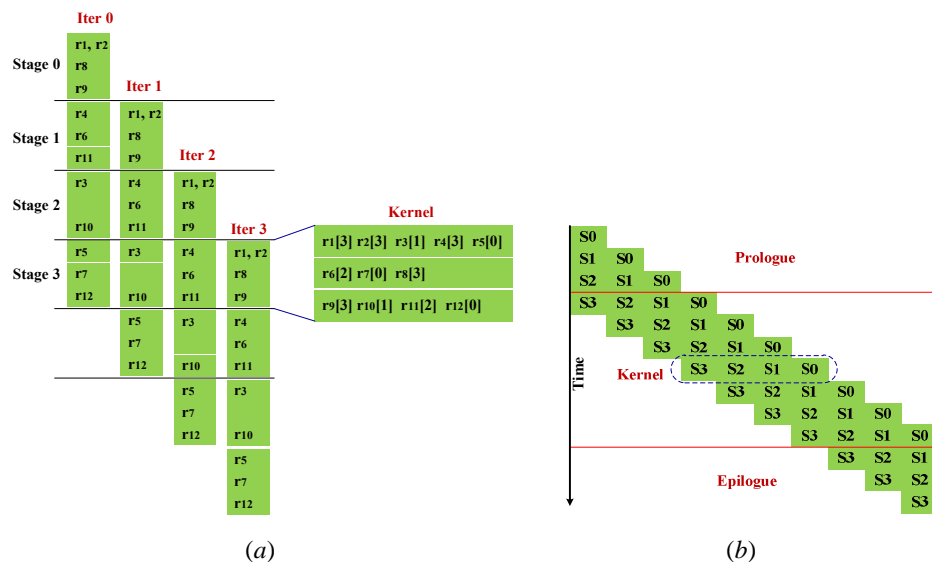
Figure 4.17: Modulo Scheduling. (*a*) Loop Schedule Sample. (*b*) Loop Execution time Flow.

Resources bound MII (*ResMII*), and Recurrence bound MII (*RecMII*).

ResMII is a measure of how many cycles are required to map all the instructions in a single loop iteration on the available resources (functional units) without any resource conflicts (regardless of dependences).

ResMII for FU type *f* is computed as the division of the total bitwidth allocated for FU type *f*, *allocatedBitwidth(f)* (e.g. 256-bits ALU), and the total bitwidth of instructions in the kernel supported by the FU type *f* (e.g. *add*, *sub* operations on ALU). ResMII is determined as the worst case constraint across all FU types.

$$\mathrm{Re}\,sMII(f) = AllocatedBitwidth(f)/TotalInstructionsBitwidth(f)$$

RecMII is derived from the latency calculations around elementary circuits in the dependence graph for the loop body. Assume that the sum of latencies along some elementary circuit *c* in the graph is *Latency(c)* and that the sum of the distances along that circuit is *Distance(c)*. RecMII for circuit c is computed as the division:

$$\mathrm{Re}\,cMII(c) = Latency(c)/Dis\tan ce(c)$$

The RecMII is determined by considering the worst case constraint across all circuits.

4.5.1.2 Swing Modulo Scheduling

Swing modulo-scheduling algorithm [87] is a modulo scheduling technique designed to minimizing registers requirements and critical path delay. The algorithm starts by building a DFG to represent all data dependences in the loop. Then, the DFG nodes are ordered in a list. The scheduler then run on the ordered list and tries to allocate the necessary time slots for each instruction.

Swing modulo scheduling differentiates from other modulo scheduling algorithm in its DFG nodes ordering algorithm. It starts by ordering recurrence circuits nodes giving the circuit with highest *RecMII* the highest priority. Then, it goes forth and back on the DFG (swinging) ordering predecessors of partially ordered nodes then successors, then predecessors and so. The later pattern of ordering is what minimizes variables lifetimes since nodes ordered for schedule near their predecessors and successors.

Table 4.4: Modulo Scheduled kernel example

Kernel	
t = 0	$i_2[3], i_{23}[3], i_8[2], i_{21}[2], i_{12}[1], i_{15}[0]$
t = 1	$i_0[3], i_{18}[3], i_9[2]$
t = 2	$i_4[3], i_6[3], i_7[3], i_{19}[3], i_{14}[1]$

4.5.1.3 Hardware Support

SOpenCL does not generate separate code segments for the prologue and epilogue portions of the modulo schedule but instead uses the concept of valid bits.

As described in section 4.2, each data token exchanged between functional units, or streamed in or out of the datapath is accompanied by a valid bit. That bit shows whether the value carried by the data token is valid or not. The operation carried out by a FU will only be valid, if all input data to the FU are valid. Since at the beginning of a loop execution, all data tokens are reset to invalid, only data sourced by the input streaming unit are valid. In each cycle, these data tokens spread to the rest of the datapath-in a movement reminiscent to a wave-thus gradually enabling execution on the FUs. This gradual triggering of the FUs implements the prologue schedule.

Figure 4.18 depicts the flow of valid bits over the whole loop execution duration for the kernel of Table 4.4. *Phi* instructions always become valid (*green*) at the first loop iteration, while the rest of the instructions become valid once all their input operands are valid. After 10 cycles all instructions become valid, i.e. the schedule
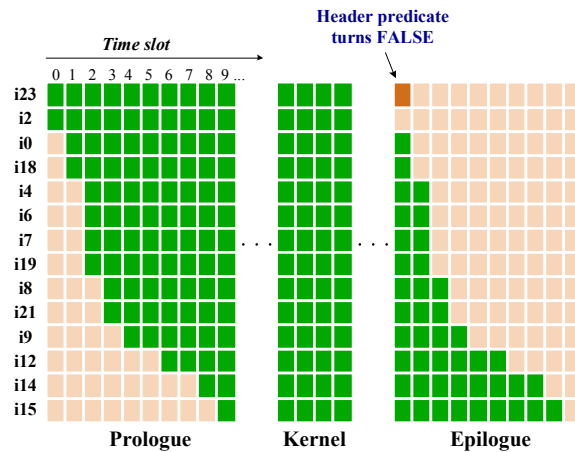
Figure 4.18: Valid-bit flow over the loop execution duration for the kernel of Table 4.4. Green for true valid-bits and red for false valid-bits.

95

execution reaches the steady state.

Loop termination occurs once the *header predicate* (predicate of the loop header block) became false. The header predicate is always a *phi* instruction (*i23* in Table 4.4) with its back-edge value is the negation of the termination condition (*i21* in the Table 4.4). In Figure 4.18, when the header predicate *i23* becomes false it invalidates the output of the other phi instructions (*i2, and i0*). The false valid bit of the phi instructions propagates for few cycles (10 cycles) until all instructions output is invalid, then the loop terminates.

4.6   Cache Instantiation

The target of the cache in the PE architecture is to exploit temporal and spatial locality in the access pattern of each input stream of the inner loop. A cache will be instantiated only if at least one input stream is deemed to be able to benefit from the use of a cache. The decision is taken independently for each input data stream, however all input data streams eventually use the same physical cache resources.

An input data stream is a candidate to use the cache, only if it has a predictable, regular memory access pattern, and accesses off-chip memory. Local arrays mapped on on-chip memories are excluded because of their very low latency compared to off-chip memories and similar to the cache latency. An input data stream with an irregular or dynamic access pattern is not expected to benefit significantly from a cache, since cache size is essentially just a few kilobytes due to resource limitations.

4.6.1   Memory Addresses Profiling

SOpenCL backend uses profiling of memory read accesses to determine cache requirements. The profiler computes all addresses generated for each read operation in the inner loop code over all the iterations of the nested loop. Then the addresses are placed in blocks of continuous addresses. In a block of continuous addresses, the distance between two addresses does not exceed the width of the system data bus width (in bytes), otherwise a block of cached data will have gaps of data lines never used. Since allocated cache has limited size (few kilobytes) and we only allocate

96

cache for regular data streams, hence, the gaps between useful data lines in the access pattern will appear regularly, such gaps are expensive and are avoided.

Figure 4.19b shows the set of generated addresses blocks for the C code sample of Figure 4.19a. In the given example a continuous addresses block represents *all* continuous addresses generated for single outer loop iteration. The profiler produces output only for data streams with regular access pattern. A data stream considered regular if all the generated blocks of addresses have the same size, and have an identical addresses distances. Generated addresses blocks annotated with the outer loop index. This annotation is used later to compute the cache reuse distance while determining the cache configuration.

4.6.2   Cache Configuration Computation

To determine whether a cache should be instantiated or not, the hardware should check whether a data stream is a candidate for being stored in the cache. This happens if it is a read-only stream and has a regular access pattern which can be determined from the profiler output. In more detail:

- Compute stream cache configuration: for each candidate data stream estimate the degree of data reuse, reuse distance, and the cache size required to effectively host reused data.

- Select a subset of the candidate streams for being supported by the cache.

For each candidate stream the tool computes two parameters: *reuse ratio* and *cache configurationReuse Ratio (reuse):* For a data stream, the *reuse* parameter measures how many repetitive addresses generated over the loop trip as in (1). The
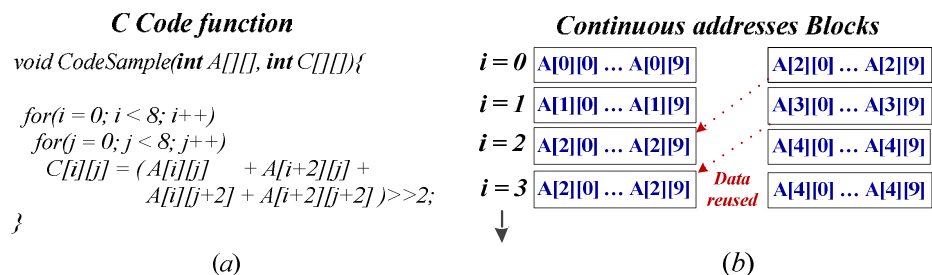
Figure 4.19: Example of data reuse across outer loop iterations. (a) C code sample with row wise access pattern. (b) Memory accesses profiler output, set of continuous addresses blocks.

97

reuse parameter value is in the range [0, 1].

$$reuse_i = \frac{\#TotalAddresses - \#UniqueAddresses}{\#TotalAddresses} \tag{1}$$

Cache Configuration (Size): Each data stream has its own preserved space in the cache unit that cannot be used by other data streams. The tool flow decides the space size and configuration for each data stream to host the amount of data reuse computed earlier. A cache configuration consists of two parameters: Data Block Size (*DBSize*) and Data Blocks Count (*DBCount*). The cache space size allocated for the stream is the multiplication of both values as shown in (2).

$$Size_i = DBSize_i * DBCount_i \tag{2}$$

A data block size (*DBSize$_i$*) is computed from the size of a continuous addresses block generated by the profiler. The size computed as the distance between the minimum and maximum addresses. Then the distance is rounded to the nearer upper power of 2. In Figure 4.19b, the data block size computed initially equal to 40 bytes rounded up to 64 bytes. The *DBSize$_i$* size is rounded to a power of 2 value because the addressing scheme of cache data blocks dictates that. A cache data block is assigned an address space that spans a power of 2 bytes. For example a 256-byte data block is assigned a base address *0x******00*. The specific address space simplifies the process of detecting valid/invalid data in the cache.

The count of data blocks is equal to the cache *reuse distance*. Conventionally, cache reuse distance [96] is the number of distinctive data elements accessed between two consecutive uses of the same element. In our design flow, we apply a slightly different definition: the cache reuse distance is the number data blocks written to the cache before a data reuse occurs. In Figure 4.19b, after 2 outer loop iterations a data reuse occurs and 4 blocks are loaded to the cache, hence, *DBCount$_i$* equals 4.

The reason behind choosing *DBCount$_i$* to be equal to the cache reuse distance is the regular access pattern of a candidate data stream. Because a candidate data stream has a regular access pattern, data reuse occurs at regular distances. Hence, once we reach the iteration where data reuse starts, data blocks loaded earlier will be reused regularly. So we need to keep all loaded data blocks until a data reuse starts, because after that we can replace old blocks with new ones.

98

The regular access pattern of a candidate data stream also drives the *replacement policy* of the cache data block. Initially, the cache blocks are empty; the cache fills an empty block for each read request that has no data in the cache. When the cache is full, the oldest block in the cache with no pending read requests is evicted and the block is allocated for the new read request. If all blocks have pending read requests, the first block finish serving its current pending requests is allocated for the new read request.

Given the computed parameters *reuse* and *size* for each data stream, we solve the problem of maximizing the amount of data reuse within an upper bound constraint on the cache size as in (3). We use exact enumeration techniques to solve the problem in (3). An enumeration of all possible combinations is performed and the combination with maximum total reuse is selected.

$$\max_{i \in SIN} \sum_i reuse_i \ , \ So \left(\sum_i Size_i \le Cache\_Size \right) \tag{3}$$

4.7 Local Buffers Synchronization

A key feature of the proposed architectural template of chapter 3, are the asynchronous interconnect channels between a producer and a consumer, namely scalar data FIFO channels and local streams buffers (see section 3.2.2). In the case of scalar data FIFO channel, the dependencies appear as instruction operands, hence the datapath (or AGU) and CE modules allocate the proper data FIFOs (as discussed in chapter 3, section 3.3.1). However, the generation of local streams buffers requires dependency information extraction through memory access pattern analysis, in order to build a dependency graph and guide the generation of synchronization signals.

Figure 4.20 depicts the dependency graphs generated for each local data stream in the LUD kernel (Figure 3.2). SOpenCL generates dependency graphs for each local stream by analyzing memory dependencies between individual load/store operations in each PE and CE module.

A dependency graph consists of nodes, where a node is a PE or a CE module. Each node is labeled by its memory access type for the specific data stream: Write (**W**), Read (**R**), or Read/Write (**R/W**). A dependency can occur between two nodes as

99

long as at least one of the nodes performs a write operation. The dependency is represented by a directed edge labeled by the dependency distance. The latter is the cross-iteration interval at which the dependency occurs. For example, in **Error! Reference source not found.**a, the dependency $PE(L_{0\_3}) \rightarrow CE1$ with distance 0, means that *CE1* cannot start read operation until $PE(L_{0\_3})$ finishes its write operation. On the other hand, the dependency $PE(L_{0\_3}) \leftarrow CE1$ with distance 1, means that $PE(L_{0\_3})$ waits for *CE1* to finish its read operation before starting a write operation for the next iteration. An edge with distance 0 is called a *forward edge*, whereas an edge with distance greater than 0 is called a *backward edge*.

After building the dependency graph for a data stream, the tool performs a redundant dependency elimination optimization in order to reduce the number of synchronization channels corresponding to dependency edges. **Error! Reference source not found.** depicts the pseudo-code of this optimization. The algorithm first generates an ordering of the graph nodes such that a node comes after all its
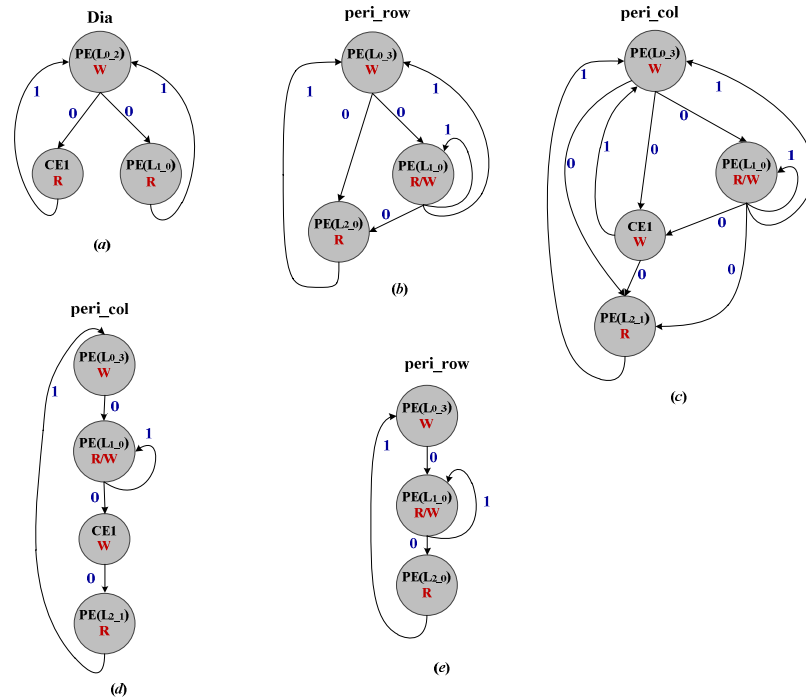
Figure 4.20: Memory Dependency Graphs for LUD OpenCL architecture in Figure 3.2. **W:** refers to Write memory, **R:** refers to Read memory. (a) *Dia* local stream dependency graph. (b) Non-optimized *peri_row* local stream dependency graph. (c) Non-optimized *peri_col* local stream dependency graph. (d) Optimized *peri_col* local stream dependency graph. (e) Optimized *peri_row* local stream dependency graph.

predecessors. Then, for each node *Ni* the algorithm performs elimination of forward and backward predecessors (incoming edges) separately. A forward edge from predecessor *Pi* is eliminated if there is a path PP_i between *Pi* and any of the node predecessors (excluding *Pi*) fullfils the following constraint:

$$Distance\ (PP_i)\ \leq\ Distance\ (Pi \rightarrow Ni)$$

Where *Distance()* returns the summation of distance label on all edges of the given path. The distance constraint on the path PP_i ensures that the dependency implied by the path PP_i occurs before or at the same iteration as the eliminated dependency edge

Algorithm 4.7: Redundant Dependency Elimination.

Input: Memory dependency flow graph.
Output: Optimized memory dependency graph.

```
 1:  G   → Dependency flow graph.
 2:  //
 3:  eliminate_redundant_edge( G ){
 4:      G' = predecessor_first_order( G );
 5:      foreach node Nᵢ in G' do
 6:          eliminate_forward_edges(Nᵢ, G');
 7:          eliminate_backward_edges(Nᵢ, G');
 8:      end for
 9:  }
10:  //
11:  eliminate_forward_edges(Nᵢ, G'){
12:
13:      foreach predecessor(Nᵢ) Pᵢ do
14:          if( distance( Nᵢ, Pᵢ ) == 0 )then
15:              foreach predecessor(Nᵢ) Pᵢ' != Pᵢ do
16:                  if( has_path(Pᵢ ,Pᵢ') ) then
17:                      delete Pᵢ
18:                      break;
19:                  end if
20:              end for
21:          end if
22:      end for
23:  }
24:  //
25:  eliminate_backward_edges(Nᵢ, G'){
26:      foreach successor(Nᵢ) Sᵢ do
27:          if( distance( Nᵢ, Sᵢ ) > 0 )then
28:              foreach successor(Nᵢ) S'ᵢ do
29:                  if( has_path(S'ᵢ , Sᵢ) ) then
30:                      delete Sᵢ
31:                      break;
32:                  end if
33:              end for
34:          end if
35:      end for
36:  }
```

101

$Pi \rightarrow Ni$.

A backward edge to successor S_i is eliminated if there is a path between any of the node's successors and S_i such that, the maximum distance on such a path should be less than or equal to the edge $Pi \leftarrow Ni$. **Error! Reference source not found.**d and Figure 4.19e shows the result of applying **Error! Reference source not found.** on dependency graphs of Figure 4.19c and Figure 4.19b respectively.

The equivalence of the new dependency graph to the old one can be verified as follows: for each eliminated direct dependency edge $Pi \rightarrow Ni$, there is at least one path in the dependency graph from node Pi to node Ni, that fullfils the distance constraint. For example, in Figure 4.20c, the edge $PE_{(L_{0\_3})} \rightarrow CE1$ is eliminated. In the optimized graph of Figure 4.20d, the path "$PE_{(L_{0\_3})} \rightarrow PE_{(L_{1\_0})} \rightarrow CE1$" is equivalent to the eliminated one and both has distance equal to 0 which fulfils the distance constraint.

Dependency graph optimization simplifies the local buffers synchronization.. Each dependency edge is served by a 1-bit *finish* signal (refer to Figure 3.3) and a FIFO 1-bit wide. Redundant dependency edges elimination leads to eliminating corresponding *finish* signal and its FIFO. While the FIFOs cost is small, eliminating *finish* signals affects significantly the routing complexity and control signals computation at each node. For example, the dependency graph of Figure 4.20c produces a network of 10 synchronization *finish* signals spreading all over the accelerator, while the optimized graph in Figure 4.20d has only 4 finish signals flow in a pipeline pattern.

Once we have the optimized dependency graph for each local data stream, the backend allocates as many Block RAMs required for each local array. For example, in the LUD kernel (Figure 2.14) each of the local arrays, *peri_col, peri_row,* and *dia* has size equal to 256 floating point elements, hence 1 KB of memory space is required for each local array. To support doubling buffering we allocate a 2 KB local buffer for each local array. The backend then uses the optimized dependency graph for each local array to generate synchronization logic. A *finish* signal and a FIFO is generated (as in Figure 3.3) for each dependency edge. Then the hardware generator

102

builds the local buffer Read/Write ports arbitration considering double buffering and computed dependencies.

4.8 Related Work

Research in architectural synthesis traditionally applied a series of optimizations to achieve efficient hardware designs. Prior research that avoided arbitrary bitwidth datatypes extensions employed a sort of bitwidth analysis to compute the minimum bitwidth to represent a variable [70, 71, 72, 73, 74].

The majority of previous work applied a series of loop transformations. PICO-NPA [13] performs loop tiling. The compiler selects the best tile shape and size to reuse already loaded data. Additionally, the tile size should match the possible available registers and local memories resources.

SPARK compilation framework [25] applies a variety of transformations including code motion using percolation scheduling, ,, and speculative code motion. Transformations like dynamic renaming while reordering operations and dynamic common subexpression elimination (CSE) also have been applied to reduce the size of required resources.

Traditional compiler optimizations have been used with all works compilation frameworks. Optimizations include dead code elimination, common sub expression elimination, constant propagation, array value propagation, and function inlining.

Extracting regular computation patterns has been the focus of prior research in behavioral datapath synthesis [65, 66, 76, 77, 78]. Regularity extraction also has also been used for custom instruction generation [61, 62, 64, 63, 79]. The proposed approaches can be categorized based on how they resolve candidate subgraph generation and candidate subgraph selection.

Candidate subgraph generation. Early work used variations of enumerations techniques augmented with a set of constraints or a guide function to prune the search space.

Atasu [62] exhaustively enumerated all possible subgraphs in the DFG using a binary tree representation. To prune the search space, Atasu used convexity and upper

limit of inputs/outputs as constraints to generate a candidate subgraph. Atasu considered single and multiple outputs subgraphs as candidates, weakly connected subgraphs also considered as a class of multiple outputs subgraphs. Goodwin [80] adapted the work of Atasu to generated fused operations for application specific processors. Goodwin added the subgraph latency constraint in addition to the number of inputs/output operands constraints used by Atasu. A less expensive enumeration technique was proposed in the work by Bonzini [79]. The proposed algorithm uses the same set of inputs/outputs and convexity constraints used in previous works, and achieves a polynomial time complexity with respect to the input/output port number.

Yu [63] proposed a more efficient enumeration approach that produces all possible subgraphs using a two phase process. In the first phase, it enumerates all upward and downward cones in the DFG, and in the second phase a union operation is applied on the generated set of upward and downward cones to produce more complex subgraphs. Yu also used the convexity and inputs/outputs number constraints to eliminate illegal subgraphs. The approach of Yu can run faster than that of Atasu because it eliminates illegal subgraphs, early in the first step. Both enumeration techniques have a worst case exponential time complexity.

Cong [81] used the method of cones enumeration. Instead of considering upward and downward cones, Cong restricted the enumeration process to upward cones only, hence supporting single output subgraphs. Cong used the number of input operands and execution time as constraints on feasible upward cones. Our algorithm also considers upwards cones only, similarly to Cong *et al.*, however without constraining the number of input operands, thus allowing us to generate the maximal patterns. Cong considers any cut of a feasible cone to be a feasible candidate subgraph. In our approach, a cut T of a candidate upward cone C_i (i.e. a grammar rule C_i) is a feasible candidate subgraph (i.e. translated into new rule) in two cases: if the cut T pattern appears in other candidate cones (i.e. in other grammar rules productions), or if the cut T pattern appears more than once within the same candidate cone subgraph. For example, candidate Rule B → AA, includes two instances of rule A. Otherwise, for our purposes of multiplexers size reduction, implementing a candidate upward cone is more efficient than just implementing a cut of its subgraph. Hence we dismiss generating such patterns in our grammar structure.

104

Work by Clark [64] examines each node in the DFG and uses it as a *seed* for a candidate subgraph. This seed is grown downwards along dataflow edges to create new candidates. A guide function is used to determine which nodes are the best directions to grow, and when to stop growing a subgraph. The guide function assigns a priority to each edge in the DFG based on its criticality, latency, and area.

Another set of early work used pattern recognition techniques to extract computations regularities in a DFG. Rao et el. [65] used string pattern recognition techniques on a DFG to extract regular computation patterns. First, he converts the DFG into a string of characters (operations), and then a string matching algorithm is used to find regular patterns of characters. User-defined patterns library also used in work [76] to improve quality of logical synthesis at the behavioral level. Other interesting work used predefined patterns library include scheduling and binding algorithms based on patterns matching [77, 78].

Cong [66] proposed a pattern-recognition based approach for FPGA resources reduction. According to Cong *et al*., a pattern type includes instances not completely identical. In our grammar approach, instances of a pattern (represented by a grammar rule) are completely identical. Cong *et al*. approach produces MFUs with extra multiplexers on intra-FU interconnects. The extra multiplexers cost increases the area overhead of MFUs. Moreover, multiplexers on the intra-FU interconnects would prevent generating compact, optimized MFU circuits using our pipelining algorithm.

The pattern recognition approach Cong *et al*. used is based on exhaustive subgraph enumeration. First, each DFG node is considered as a candidate pattern. For each node, all possible subgraphs are enumerated by adding one neighboring node (predecessor or successor), thus creating subgraphs of size 2. The algorithm then traverses the current pattern types set and adds the created subgraph to a matching pattern. If the subgraph does not match any previously created pattern, a new pattern is created, as long as it satisfies the convexity constraint. After processing subgraphs of size 2, subgraphs of size 3 are created from subgraphs of size 2 and the previous process is repeated. The algorithm continues until patterns cannot be grown any further. If the instances of a pattern are less than a pre-defined number, the pattern and all its instances are removed from the search space. Cong *et al*. also remove patterns totally encapsulated within a larger pattern (called maximal pattern).

105

Our grammar generation algorithm also grows patterns from each DFG node incrementally, however moving only upwards (add predecessors), unlike Cong *et al.* patterns which grow in any direction. In our case, since we start growing patterns upward from nodes at the bottom of the DFG, there is no need to grow patterns downward. This unidirectional growth reduces the complexity of the search space and thus of the algorithm. Contrary to Cong *et al.* our algorithm considers patterns completely contained in other larger patterns and dismisses patterns partially contained in other larger patterns. In fact this feature is the basis for hierarchical grammar structure. Our experimental study indicated that such patterns characterized by a finer computations granularity could often be fitter for implementation than larger, coarser patterns.

Cong *et al.*, pattern recognition algorithm generates a large number of patterns (in the order of thousands) covering all possible patterns in the DFG. While their approach is complete and more efficient than others, it still produces a large amount of unnecessary and redundant patterns and takes minutes to process a DFG with a few hundreds of nodes. Our grammar-based algorithm produces just a handful of patterns within one second, for DFGs with thousands of nodes. At the same time, it achieves a similar reduction in area (~20%) to that achieved by Cong *et al.* algorithm.

Several papers used candidate generation algorithms based on iterative combination of primitive operations [61, 82, 83, 84]. The basic idea behind iterative combination of primitives is to use a profiling approach to find the frequency of a combination of two operations in the input program, replace them with new super-node and repeat the process until a stopping condition is met. Brisk [61] extracts regular computation patterns from a DFG by examining each edge in the DFG and record the number of occurrences for each edge type. Consequently, the most frequently edge types are converted to super-nodes. The process I s repeated iteratively until a stopping condition (like graph coverage) is met. Work by Bennett [83] considers the combination of two operations that occur in subsequent line of code to reduce static code size. This technique is irrespective of the dataflow graph and is used mainly for code size reduction.

Our work utilizes the same concept of replacing a combination of two operations (or an edge) with a super node (i.e. rule). The work of Brisk *et al.* destroys a

106

previously created super node *S* when a new super node *A* is created contains the super node *S*. Such behavior prevents building hierarchical models of super nodes (i.e. rules). An instance of super node *S* will be available in the final set of super nodes only if not all its instance have been destroyed when the algorithm reaches the stopping condition. Our approach preserves all instances of a super node allowing creating hierarchical models of supper nodes. Such super nodes may be more fit for implementation than their parent super nodes. Removing them during the creation of their parent super nodes we lose the opportunity to exploit them leading to sub-optimal design.

Candidate subgraph selection. All previously mentioned papers approached the candidate subgraph selection problem in a similar manner: a cost function and a set of metrics have been used to weigh the performance gain and the feasibility of a candidate subgraph. Previous research that has targeted application specific processors and instructions set extension [61, 62, 63, 64], where the concern is increasing processors performance, metrics that estimate latency, area, and inputs/outputs number have be used. Clark used a greedy selection algorithm based on dynamic programming. A ratio of cycles savings and area is computed for each candidate subgraph and used as a priority metric for selection.

Cong [66] used metrics that estimate multiplexers cost reduction and latency to reduce FPGA resources. The latency metric gives higher priority to flat subgraphs to reduce latency overhead. Our patterns selection algorithm has few similarities with that of Cong *et al*. Both algorithms are greedy and use metrics for area reduction estimation. In our case however, latency is not a primary concern at the instruction clustering phase. The critical path latency is actually effectively reduced during MFU pipelining. However, using the flatness metric of Cong *et al*. could help reduce the variables lifetime overhead

Work described in [85] uses a speedup analysis to select an optimal set of subgraph candidates. Speedup analysis is performed by comparing the approximate subgraph execution time in software, as a sequence of instructions, with the approximate time the subgraph takes if implemented in hardware, as a single special instruction. The most promising candidates are then passed for hardware mapping.

107

The enumeration techniques in previous research have a worst case exponential computational complexity. Moreover, the generated set of candidate subgraphs is typically very large (thousands of subgraphs) for large DFGs, and most of them are redundant or cannot produce optimized designs. Our grammar-driven approach performs a very fast search and produces a small number of subgraphs by focusing only on repetitive patterns as candidates. Another distinct difference is the clear hierarchal relationship among the generated grammar rules. On the other hand, in enumeration based approaches, only a portion of subgraph nodes may be members of tens other subgraphs. This complex relationship among the subgraphs and their large number increases the complexity of candidate subgraph selection algorithm.

Prior work addressed the problem of multiplexers size reduction in a variety of ways. The majority of works are based on resources binding techniques in datapath synthesis. Huang *et al.* [97] developed a weighted bipartite matching approach to minimize the multiplexers following a step-by-step method. First, variable-register binding is applied, followed by an operation-FU binding step. The register binding method tries to minimize the total number of operation types with outputs bonded to the same register, and at the same time minimize the total number of input registers used by operations with outputs bonded to the same register. The FU binding method tries to minimize the number of new input registers required when assigning an operation to an FU instance. Chen *et al.* [98] enhanced Huang methods and updated the method of calculating the weighted bipartite graph. Moreover, they applied the register-binding algorithm after FU binding.

Cong *et al.* [99] apply a similar algorithm to Huang *et al.* on a distributed register file architecture. The proposed architecture model consists of one or more islands of registers and functional units. The binding algorithm concentrates on reducing inter-island interconnects and multiplexers.

The drawback of previous binding algorithms is that they fail to exploit regular patterns and rely solely on iterative algorithms to minimize the multiplexers overhead generated during resources binding.

Our work tackles the problem of multiplexers area overhead earlier in the design flow, similarly to Cong *et al.* [66], by identifying and exploiting regular patterns in

the problem DFG. Cong *et al.* uses a multiplexer area overhead metric that favors MFUs with less internal multiplexers and does not consider the overall reduction in multiplexers count. Exploiting regular patterns we create islands of primitive FUs (i.e. MFUs) with multiplexers-free internal interconnects. Since we only support MFUs with no multiplexers on internal interconnects, the rules selection algorithm (Algorithm 2) uses a metric (MUXG in equation 3) that favors MFUs which result to a higher reduction in the total number of multiplexers in the design. This objective is similar to that of binding algorithms.

Few research papers addressed the problem of MFU implementation. Works in the field of custom instruction set generation [64, 86] considered implementations of MFU that support different types of macro-instructions. Clark proposed a *wildcard* approach to share resources between different subgraphs. Wildcards are subgraphs that have a similar shape, but operations in one node may differ. This approach increases routing complexity of the MFU when internal multiplexers introduced to support different types of subgraphs. Pothineni [86] proposed a heuristic that accounts for internal multiplexers in merged subgraphs. The heuristic merges multi-cycle subgraphs, by first decomposing them into single cycle subgraphs that can be merged during the binding process.

CHAPTER 5

EXPERIMENTAL EVALUATION

In this chapter we present our experimental evaluation and analysis of SOpenCL. We examine independently the impact of asynchronous execution model, bitwidth optimization, instruction clustering and cache utilization on performance and area.

5.1 Benchmark Suite

Table 5.1 outlines the set of benchmarks used in our experimental evaluation. Some of the kernels base source is OpenCL and others are from C source origin. The kernels are from a variety of fields: multimedia, cryptography, telecommunication and linear algebra. Following is a brief description for each kernel highlighting its specific characteristics.

CMC is the Chroma motion interpolation kernel of the AVS video standard. CMC performs pixels interpolation on the chrominance pixels in a video frame. CMC uses a 2-dimensional sliding window of size 2×2 to compute the interpolation of a single pixel. The coefficients of the interpolation filter are derived from the motion vector for each Macroblock (16×16 block of pixels) [90]. The Chroma component interpolation (Figure 5.1) follows the equation:

Table 5.1: Applications used for experimental evaluation.

Application	Description	Source	Data
CMC	AVS Video Decoder Chroma motion interpolation [90]	OpenCL	Int
LMC	AVS Video Decoder Luma motion interpolation [90]	C	Int
DCT	H.264 Video Encoder 8x8 Integer DCT [91]	OpenCL	Int
SEAL	Seal cryptography kernel [8]	C	Int
CN	Forward Error Correction (FEC) decoder CheckNode Kernel [92]	OpenCL	Int
BN	Forward Error Correction (FEC) decoder BitNode Kernel [92]	OpenCL	Int
LUD	LU Decomposition-Perimeter [23].	OpenCL	FP
Deblocking	AVS Video Decoder Deblocking Filter [93].	C	Int

110
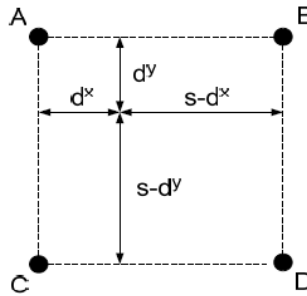
Figure 5.1: Sub-pixel Chroma interpolation in AVS Motion Compensation.

$$pred'[x][y] = (8 - dx)(8 - dy)A + dx(8 - dy)B + (8 - dx)dyC + dxdyD$$
$$pred[x][y] = (pred'[x][y] + 32) >> 6$$

The CMC kernel consumes 10 pixels per loop iteration and produces 4 pixels per iteration.

LMC is the Luminance motion interpolation kernel of the AVS video standard. LMC performs pixel interpolation on luminance pixels in a video frame. Like CMC, LMC kernel uses sliding window for interpolation, but the size of the sliding window is variable (1×4, 1×5, 4×1, 5×1, 4×4, 4×5, and 5×4) depending on the motion vector of each Macroblock. LMC kernel consumes up to 20 pixels per loop iterations and produces 1 pixel per iteration.

Discrete Cosine Transform (DCT) kernel, used in H.264 video encoder among others, converts 2D 8x8 pixel blocks in an image frame to frequency coefficients each time it is invoked. . The kernel consists of a nested loop which encapsulates two inner loops. The first inner loop processes the input pixels block and produces a partially transformed 8×8 block stored in a local array. The second inner loop operates on the partially transformed block and completes the DCT computations.

SEAL is a fast, software-oriented encryption algorithm. SEAL is a stream cipher, i.e. incoming data to be encrypted are streamed in the algorithm and continuously encrypted. SEAL encryption uses a random 160-bit encryption key and has a longer initialization phase during which a large set of tables is done using the Secure Hash Algorithm. An invocation of the SEAL kernel encrypts a 4KB plaintext message. The algorithm is divided in two steps: Tables generation, and a pseudo-random function execution. Tables generation is typically performed once for a communication

111

session. Given the generated tables and a 32-bit position index n, the pseudo-random function stretches n to L-bit pseudo-random string. L can be made arbitrarily large ranging from a few bytes to thousand of bytes. In our SEAL kernel, L equals 4 KB.

In terms of implementation characteristics, the C code includes an inner loop which forms a recurrence circuit limiting the initiation interval (II) to 60 in all configurations. To make things worse, the memory access pattern in the SEAL kernel is runtime dependent, i.e. read addresses computation depends on data loaded from the memory. As a result, a unified PE architecture (the datapath performs addresses computation) is generated for the SEAL accelerator.

BN and CN kernels are forward error correction kernels used in the DVB-S2 standard (Digital Video Broadcasting – Satellite second generation). The standard is based on, and improves upon its predecessor DVB-S. It uses a new coding scheme based on a modern LDPC code. It also uses VCM (Variable Coding and Modulation) and ACM (Adaptive Coding and Modulation) modes, which allow optimizing bandwidth utilization by dynamically changing transmission parameters. Both BN and CN kernels have a 1-dimensional computations grid. The kernels are computationally intensive. For example the CN kernel DFG has 3962 nodes. The kernels require a significant memory bandwidth: BN kernel consumes 128 Bytes per loop iteration, and CN kernel consumes 96 Bytes per iteration.

Deblocking Filter is a video filter applied to blocks in decoded video to improve visual quality by smoothing the sharp edges between macroblocks. Video frames normally partitioned into macroblocks, which further partitioned into smaller blocks processed independently, a process leads to distortions at the blocks edges. Each block edge is assigned a boundary strength based on whether it is also a macroblock boundary, the coding (intra/inter) of the blocks, whether references (in motion prediction and reference frame choice) differ, and whether it is a luma or chroma edge. Stronger levels of filtering are assigned by this scheme where there is likely to be more distortion. The filter can modify as many as three samples on either side of a given block edge. In most cases it can modify one or two samples on either side of the edge. Deblocking kernel has a RAW memory dependency across outer loops iteration of distance equals 1 preventing pipelining and overlapping the execution of successive outer loop iterations.

112

LU Decomposition is an algorithm to calculate the solutions of a set of linear equations. The LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix. The product sometimes includes a permutation matrix as well. LU decomposition is a key step in several fundamental numerical algorithms in linear algebra such as solving a system of linear equations, inverting a matrix, or computing the determinant of a matrix. LU Decomposition kernel consists of three nested loops: the first and third nested loops perform data prefetching and write back, respectively. The second nested loop performs the main LU Decomposition kernel computations. The three nested loops have a clear forward dependency flow (prefetch → compute → write) that allows for execution pipelining.

5.2 Methodology

The aforementioned backend transformations and hardware generation algorithms in chapter 4: If-conversion, code slicing, instructions clustering, scheduling and cache instantiation have been implemented as separated passes in the LLVM compiler.

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. However, for some kernels as **BN** and **CN**, hundreds of instructions scheduled per FU produce very large multiplexers, hence multiple FUs are allocated. C_C configuration 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

Table 5.2: Experimentation Data Set Size.

Application	Data Set
CMC	VGA Frame: 640×480
LMC	VGA Frame: 640×480
DCT	VGA Frame: 640×480
SEAL	4 KB Plaintext message.
CN	32400 Data points
BN	64800 Data points
LUD	128×128 Data matrix
Deblocking	VGA Frame: 640×480

113

application to achieve the average II between the two extremes. For applications with little computation in each loop (such as LUD) the C_B configuration proved similar to C_C. In SEAL kernel a recurrence circuit limited the II value to 60 cycles for all configurations.

Besides the three resource configurations, architectural exploration also considers parameters such as sequential/concurrent execution, instruction clustering, bitwidth optimizations and cache availability. For the evaluation of our design we used the Xilinx Virtex-6 LX760 FPGA and Xilinx ISE 12.4 toolset for synthesis, placement and routing. The Virtex-6 LX760 device includes 118560 slices, 720 RAMB36 Block-RAMs, and 864 DSP48 modules. The tool flow generates a testbench (Figure 5.2) used for simulation and verification. Table 5.2 summarizes the data set size used in verification/simulation of each benchmark.

5.3   Execution Model Evaluation

The concurrent execution model adopted in the proposed architectural template increases the utilization ratio of the allocated resources and reduces the duration each component stays idle through overlapping the execution of multiple components. In this section, we experiment with the concurrent execution mode for each of the three configurations C_A, C_B, and C_C All other optimizations are enabled by default.

Table 5.3 summarizes the area results after the synthesis performed for the benchmarks of Table 5.1. The general trend is that area requirements increase from configuration C_A to configuration C_C when the loop body encompasses enough computations to exploit the additional resources. Concurrent mode configurations tend to consume more slices than sequential ones. The additional hardware is used to implement the synchronization FIFOs of the PE and CE modules and synchronization
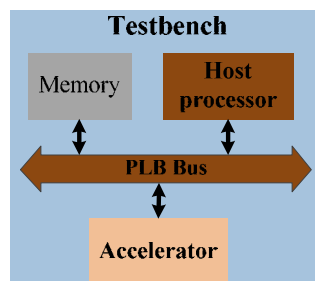

Figure 5.2: Simulation and Verification Testbench.

114

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), and it depends on the number of scalar variables FIFO channels and Local buffers synchronization signals available in the architecture. For example, in the LUD kernel, there are 25 scalar variables (LLVM instructions) computed in different parent CE modules and passed to children PE modules. Note that most of the scalar variables here are LLVM assembly instructions that do not change during the course of the inner most loop iterations, the backend applies loop-invariant code motion and move them to outer loops, hence they computed in CE modules and must passed through FIFOs to the consumer PE modules. Each scalar variable uses a FIFO channel of size equals 3. The total increase in slices in the LUD kernel (around 800 slices) is a combination of the scalar variables channels and local buffers synchronization channels for each one of the streams *dia, peri_col,* and *peri_row* as depicted in Figure 4.19.

An additional overhead stems from the routing overhead of the synchronization signals *valid* and *hold* in each channel. The use of the *valid* and *hold* in the control mechanisms at each module (e.g. stall execution at *hold* signal) increases control

Table 5.3: Concurrent/Sequential modes area results for the benchmarks implemented on Xilinx Virtex-6 LX760 device.

	CMC							LMC					
	Concurrent			Sequential				Concurrent			Sequential		
Config.	C_A	C_B	C_C	C_A	C_B	C_C	Config.	C_A	C_B	C_C	C_A	C_B	C_C
Slices	2051	2074	3421	1596	1652	2947	Slices	2989	3540	5395	2909	3447	5304
RAMB36	1	1	1	1	1	1	RAMB36	1	1	1	1	1	1
DSP48	12	12	20	12	12	20	DSP48	5	10	18	5	10	18
	LUD							DCT					
	Concurrent			Sequential				Concurrent			Sequential		
Config.	C_A	C_B	C_C	C_A	C_B	C_C	Config.	C_A	C_B	C_C	C_A	C_B	C_C
Slices	4788	4895	4895	3908	4191	4191	Slices	3481	3615	5323	2916	3074	4416
RAMB36	3	3	3	3	3	3	RAMB36	1	1	1	1	1	1
DSP48	17	19	19	17	19	19	DSP48	14	14	14	14	14	14
	Deblocking							SEAL					
	Concurrent			Sequential				Concurrent			Sequential		
Config.	C_A	C_B	C_C	C_A	C_B	C_C	Config.	C_A	C_B	C_C	C_A	C_B	C_C
Slices	2464	2736	3379	1868	2157	2714	Slices	2089	2112	2112	1905	1945	1945
RAMB36	0	0	0	0	0	0	RAMB36	0	0	0	0	0	0
DSP48	3	3	3	3	3	3	DSP48	0	0	0	0	0	0
	BN							CN					
	Concurrent			Sequential				Concurrent			Sequential		
Config.	C_A	C_B	C_C	C_A	C_B	C_C	Config.	C_A	C_B	C_C	C_A	C_B	C_C
Slices	22304	25692	32168	22268	25640	32150	Slices	20675	27390	22044	20640	27350	22005
RAMB36	0	0	0	0	0	0	RAMB36	0	0	0	0	0	0
DSP48	4	4	4	4	4	4	DSP48	2	6	10	2	6	10

115

complexity and routing overhead.

The area overhead in the asynchronous configuration is very small or none exist if there are no scalar variables exchanges between multiple PE and CE modules, and no local streams synchronization is required. This is the case for BN and CN kernels. The LMC kernel also has a very small area overhead since only four scalar variables are exchanged between a CE and a PE module and each variable is 13-bits wide.

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 a cache, while the rest of the benchmarks use their Block RAMs to implement local buffers for local arrays. In LUD, each of the local arrays *dia, peri_row*, and *peri_col* is allocated a Block RAM of 36Kbit. In all applications, the Block RAMs are configured as 512 lines in size, each size 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 write and read transactions.

Figure 5.3 depicts the execution time (in ms) and clock rate for four benchmarks under different configurations for the work data set shown in Table 5.2. As expected, performance increases moving from configuration C_A to configuration C_C when there is enough memory bandwidth to serve the datapath I/O requirements. The limited memory bandwidth problem appears in the DCT benchmark for the concurrent configurations. The memory bandwidth of 8 bytes/cycle fails to support the datapath I/O requirements 16 bytes/cycle and 32 bytes/cycle for configurations C_B and C_C respectively.

As expected, the concurrent mode implementations in all benchmarks achieve higher computational rate and reduced execution time compared to configurations supporting sequential mode. Sequential operation (without data prefetching) frequently throttles the throughput of PE modules. Concurrent operation tends to become performance critical when II is small. This is typically the case in the C_C configuration. Faster datapaths 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.

116

The performance of concurrent 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 between $PE(L_{1\_0})$ and its parent $CE1$ as appear in dependency graph discussed in Chapter 4 (Figure 4.19), 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 concurrent execution outperforms synchronous one.

Figure 5.4 shows of the rest four benchmarks that achieved very limited performance gain using the concurrent operation. Deblocking filter (Figure 5.4a) achieves limited performance mainly because of data stream dependencies. The inner most loop of the deblocking kernel has a RAW memory dependency across outer loops iterations with distance equal to 1. In the generated architecture, the input

Figure 5.3: Execution Time (bars in *ms*) And clock frequency (lines in *MHz*) for concurrent and sequential configurations.

117

streaming units wait for a *finish* signal from the output streaming units before sending read requests. The minimal execution time improvement is due to overlapping the execution of the PE module with its parent CE module.

SEAL benchmark has a unified PE architecture; no AGU modules generated because addresses computations can be computed only at runtime. Hence, the PE module consists only of a datapath and input/output streaming units (*RGU*, *SinAlign* and *SoutAlign* units). As a result, successive outer loop iterations will not promote data prefetching since addresses generation for later iterations cannot start until the datapath finishes computations of earlier iterations. As in the Deblocking filter case, the limited reduction in execution time came from overlapping the execution of the PE module with its parent CE module.

BN and CN kernels in Figure 5.4c and 5.4d show another case where concurrent operation achieves no performance gain. BN and CN kernels have 1-dimensional computational grid. In other words, the trip count of the outer loops of the triple
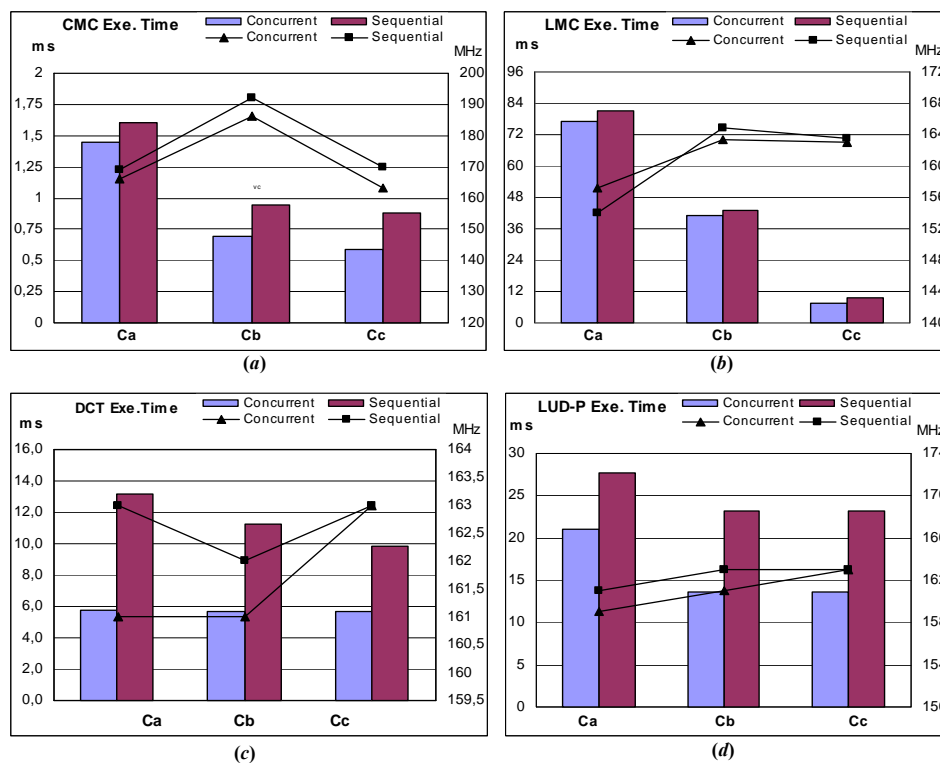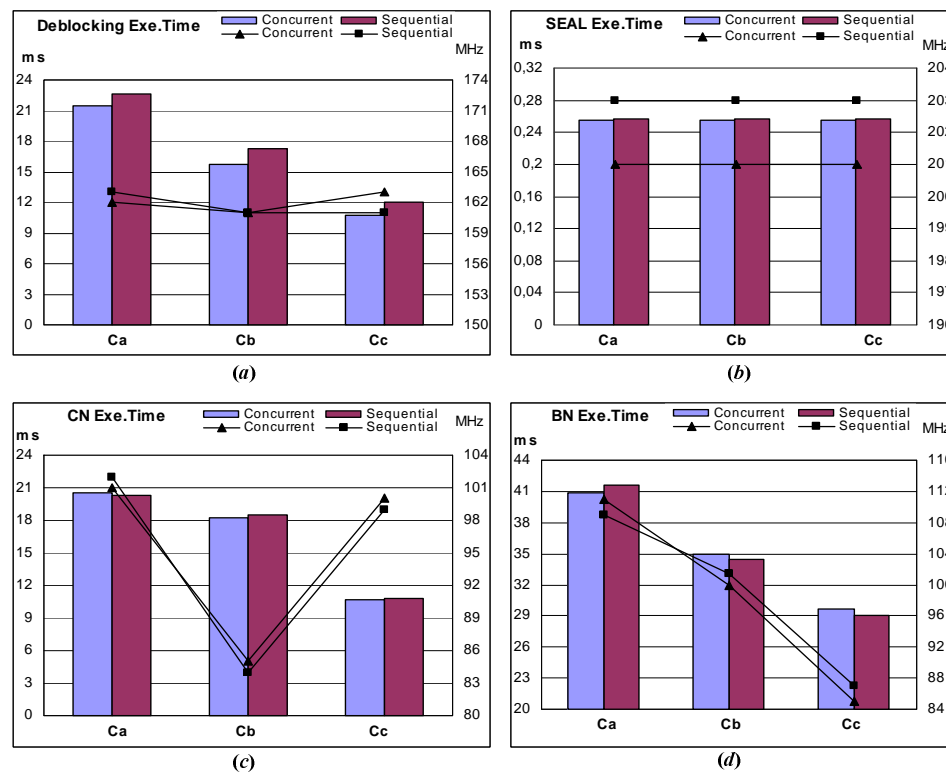

Figure 5.4: Execution time (bars, in *ms*) And clock frequency (lines in *MHz*) for concurrent and sequential configurations.

118

nested loop is one; hence the PE module (the inner loop) is initiated just once for execution.

Analyzing data dependencies and grid dimensions, the tool flow can determine if the concurrent operation could possibly improve performance or not. Figure 6.4 compares the maximum performance gain (decrease in execution time) achieved in using concurrent operation for each benchmark to its corresponding area overhead (increase of consumed resources) in each benchmark. The comparison of performance gain to the area overhead reveals the efficiency of the concurrent operation compared to the cost. Figure 5.5 shows that, the 4 benchmarks of Figure 5.3 that achieved respectable performance gain (over 30%), did so at much less area overhead. On the other hand, area overhead surpassed performance gain for benchmarks with limited concurrent operation. Performance gain and area overhead are computed as follows:

$$PerformanceGain = \frac{ExecTime(Sequential) - ExecTime(Concurrent)}{ExecTime(Sequential)}$$

$$AreaOverhead = \frac{Slices(Sequential) - Slices(Concurrent)}{Slices(Sequential)}$$

One can conclude that efficiency of concurrent operation is dependent on the application characteristics.

Concurrent operation has a mixed effect on clock frequency. A FIFO channel

Figure 5.5: Concurrent operation performance gain and area overhead

119

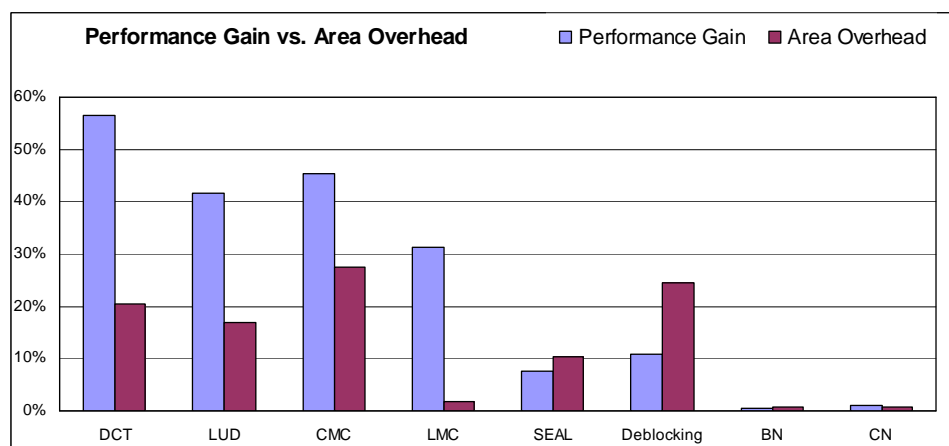plays a role in balancing the routing delay between a producer and a consumer. On the other hand, the increase in routing and control signals computation overhead caused by the synchronization signals like *valid* and *hold* produces a negative effect that may degrade clock frequency.

5.4 Bitwidth Optimization Evaluation

In this section, we experimentally evaluate bitwidth optimization for the three different target configurations. All other optimizations, i.e. asynchronous execution, instruction clustering and cache allocation are enabled, by default. In Conventional compilers targeting architectures with standard FU size (i.e. 32- and 64-bits wide), the result value is represented in 32-bits format, while 8-bits format is enough for its representation. Figure 5.6 shows the area results for each of the benchmarks with bitwidth optimization enabled (*optimized* case) or not (*original* case). As we expected, bitwidth optimization succeeds in reducing the amount of consumed resources. In the figure we can see that up to 36% reduction in area has been achieved. The negative percentage values indicate the ratio of area reduction for each configuration.

In particular, deblocking Filter (Figure 5.6h) achieves most gains from bitwidth optimization. Filter computations operate on pixel variables with char data type which is automatically extended to 32-bits by the LLVM compiler. Moreover, many kernel operations have one of their operands to a constant value equal to 2, 3 or 4. The bitwidth optimization (similar to instruction clustering) performs efficiently on computations which contain small constants, such as CN and BN as well as SEAL kernels.

In the case of LUD benchmark, bitwidth optimization affected the FIFO channels width because many scalar variables are exchanged between multiple CE and PE modules. As a result, both FUs and the FIFO channel width are optimized.

Moreover, the effects of bitwidth optimization vary from one configuration to another, since, for example, Configurations with lower II value (such as C_C) are more successful in reducing area overhead. Higher II values force the scheduler to allocate fewer functional units which should be wide enough to serve multiple instruction

120

bitwidths, hence, instructions with small bitwidth (e.g. 8-bits), could be scheduled on FUs wider than their instruction bitwidth.

Bitwidth optimization has also a positive effect on clock frequency (Table 5.4). In BN and CN kernels, the reductions in functional units width significantly reduced
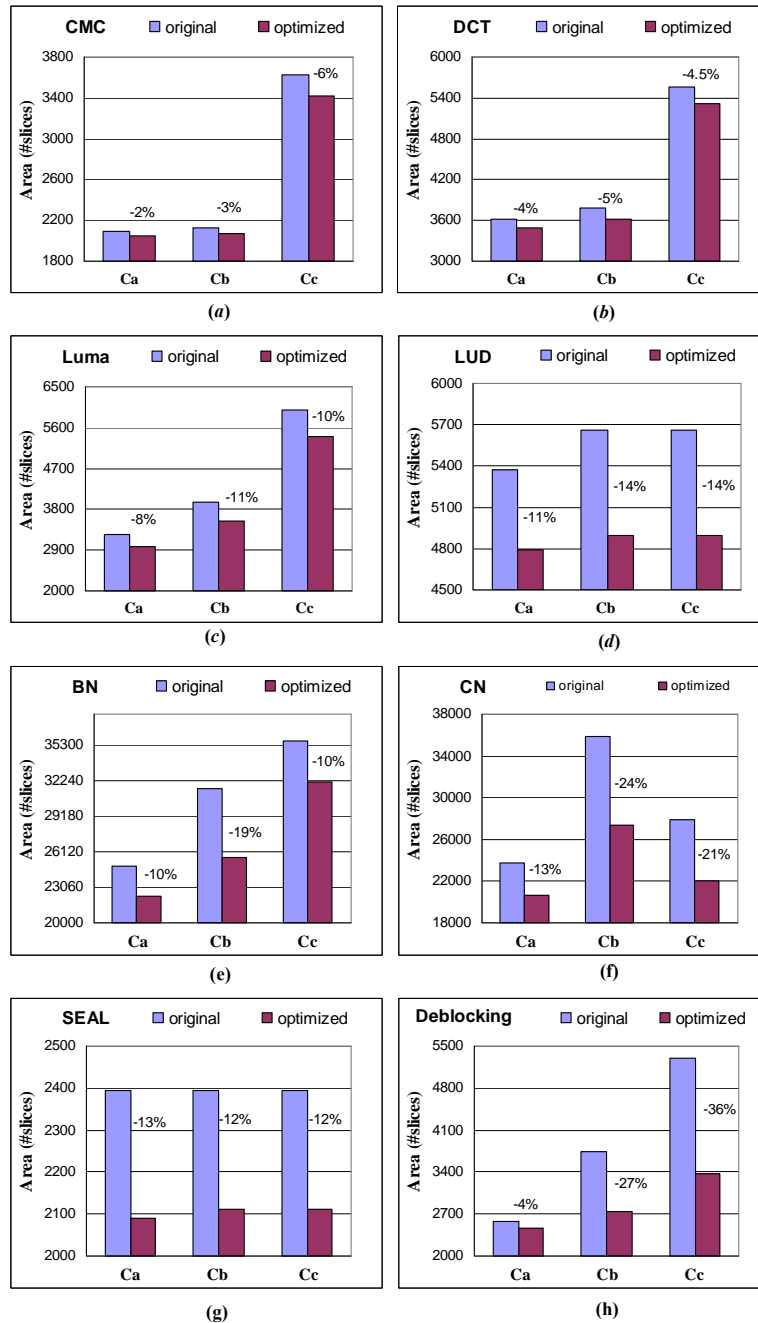

Figure 5.6: Area results for Bitwidth optimization. The percentage value above the bars indicates the percentage of Area reduction.

121

Table 5.4: Bitwidth optimization Frequency (MHz) results for the test kernels on Xilinx Virtex-6 LX760.

App.	Original			Optimized		
	C_A	C_B	C_C	C_A	C_B	C_C
CMC	165	179	161	166	186	163
LMC	160	160	162	158	164	164
DCT	134	134	134	161	161	163
SEAL	184	184	184	201	201	201
LUD	158	161	160	159	161	163
Deblocking	160	158	162	162	161	163
CN	69	66	66	101	85	100
BN	71	66	67	111	100	85

datapath routing complexity of the giving more room for the router and hence increasing clock frequency. Another noticeable improvement on clock frequency appears in the DCT benchmark. The main source for clock delay in the DCT is 32-bit multiplications. With bitwidth optimization, 20-bit multiplication is only required reducing significantly the clock delay.

5.5   Instruction Clustering Evaluation

Instruction clustering is a powerful optimization aiming at reducing area overhead and routing complexity especially in computation bound designs. In this section, we experimentally evaluate instruction clustering optimizations for the three different target configurations. All other optimizations are enabled, be default.

Table 5.5 summarizes DFG statistics after grammar generation and rule selection. Column "*#Rules*" lists the grammar size in numbers of rules generated for each application. Column "*#Used Rules*" lists the number of selected rules from each grammar to be implemented as MFUs in the final representation of the DFG. Column

Table 5.5: Grammar generation results on the kernels DFGs.

App.	#Rules	#Used Rules	Rule Size	#Insts.	#Insts(g)	Reduction	Coverage
CMC	6	3	[2-9]	136	86	-37%	53%
LMC	18	11	[2-4]	299	219	-27%	50%
DCT	10	8	[2-3]	307	197	-36%	52%
SEAL	8	5	[2-3]	143	107	-25%	45%
CN	18	7	[2-5]	3962	2500	-37%	40%
BN	8	5	[2-7]	2917	1677	-43%	41%
Deblocking	9	5	[2-4]	176	150	-15%	32%
LUD	1	1	[2]	20	18	-10%	10%

"*Rule Size*" shows the range of number of instructions per rule for the selected rules subset. Columns "*#Insts*" and "*#Insts(g)*" list the DFG size before and after grammar-based representation, respectively. Column "*Reduction*" shows the percentage of reduction in the number of primitive instructions. Finally, column "*Coverage*" shows the percentage of the DFG covered by the generated grammar.

Several conclusions can be drawn from table 5.5. Unlike pattern recognition and enumeration approaches, the generated set of subgraphs (i.e. rules) is much smaller in both the total number of subgraphs and subgraph size, yet it covers 40% – 53% of the program DFG.

Figure 5.7 shows the area and synthesis time results (for datapaths and AGUs only) for the benchmarks for the original and the optimized cases. A noticeable result appears in Figure 5.7e and 5.7f for CN and BN kernels, respectively. The two DFGs have very large sizes (approximately 4000 & 3000 nodes, respectively) which lead to routing congestion. Without the grammar-driven synthesis approach the ISE synthesis tool failed to successfully finish placement & routing. On the other hand, after the grammar-driven synthesis optimizations the tool took less than three hours to generate a fully placed and routed design. The reduced DFG size with grammar-based compression required around 20% less time on average to schedule and synthesize, which correlates with the reduction in DFG size.

Grammar-based designs typically involve more FU types than original designs in their datapath, due to the introduction of MFUs. The additional MFU types impose an area overhead. The issue manifests itself more clearly in the C_C configurations, where few FU instances (normally one or two) are allocated for each FU type. In Fig. 11.b and 11.c, we can notice that our algorithm achieves 30% and 17 % reduction in area for the C_B configuration in the DCT and LMC kernels respectively. For the C_C configuration the area gains are limited to 20% and 13% for DCT and LMC. The two kernels use 8 and 11 MFU types respectively in their datapath. While using MFUs reduces multiplexers' area in the design, the area overhead from the large number of used rules limits the overall area reduction for configuration C_C. On the other hand, CMC and SEAL kernels use only 3 and 5 rules respectively, with limited area overhead, hence configuration C_C outperforms configuration C_B. Note also that MFU area overhead can be reduced whenever the pipeline algorithm (Algorithm 3) identifies opportunities to produce compact and lightweight MFUs, which is the case for CMC and

123

SEAL. On the contrary, MFUs in DCT and LMC datapaths consist of heavyweight primitive FUs, that could not be effectively fused.
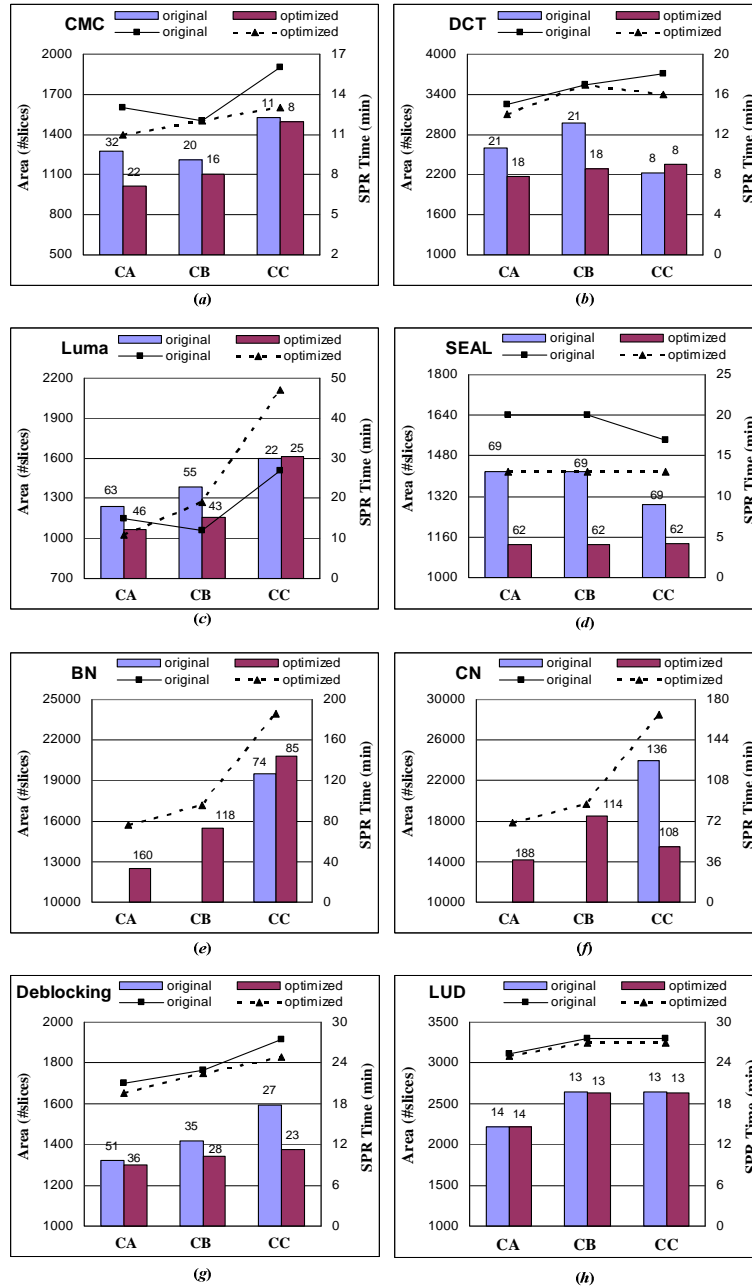
Figure 5.7: Area (slices) and Synthesis, Placement & Routing time (SPR Time in minutes). Results for original configurations, and optimized configurations (with grammar-driven datapath synthesis). In (e) and (f) the missing configurations for the original case are due to the fact that the Xilinx ISE tool chain failed to fully place & route the generated circuit. unless we apply our compression.. The numbers above the bars represent the schedule latency (in clock cycles) of a single loop iteration in each configuration.

124

DGFs characterized by patterns with a very low number of occurrences and low DFG coverage are also potentially susceptible to area overheads from the introduction of MFUs. In this case, the combination of MFUs overhead with the limited multiplexers area reduction might produce designs with very little or no area reduction, which is the case for LUD and Deblocking kernels. However, during our experimental evaluation with a variety of kernels we observed that, even for DFGs with a small number of pattern repetitions (see Table V, #Instances per Rule), area reductions are achieved because these repetitions cover 45% to 53% of the DFG. Therefore, instruction clustering led to a significant reduction in the area spent for multiplexers, overweighing the MFUs area overhead.

It appears from the experimental evaluation that the grammar-based approach sometimes performs poorly at II = 1. This is expected because in this case there are no multiplexers to optimize out. For some benchmarks (DCT and Luma) the consumed area is slightly more than that of the original configurations. For these benchmarks, the pipeline algorithm (Algorithm 3) produced fully pipelined MFUs, because they contained heavyweight primitive FUs that could not be fused with others.

Moreover, using macro-instructions in those benchmarks increased variable lifetimes, which led to allocating more registers. This is, for example, the case for the BN kernel(configuration C_A). The version produced after instruction clustering requires more area than the original one, despite the fact that the pipelining algorithm efficiently produced more compact MFUs. Most of the generated MFUs in BN kernel are not flat. They have latencies between 3 and 4 cycles (after being optimized down from 7 cycles by the pipelining algorithm). The large amount of MFUs with such latencies imposed an overhead on the scheduler, leading to increased variable lifetimes and registers requirements.

On the other hand, the proposed approach worked well even at II = 1 for other benchmarks (CMC, Deblocking and CN), in which the logic gain for generated macro-instructions was significant. The MFUs produced were compact and lightweight, which subsequently led to the area reduction. Compact MFUs generated using Algorithm 3 have a positive impact on variables lifetime at II = 1 – if the MFUs latency is not larger than 2 cycles – leading to reductions in registers requirements. Therefore, at II = 1, area reductions are obtained mainly by compressing and optimizing the generated MFUs using Algorithm 3. Otherwise designs incorporating MFUs would be expected to pose an area overhead compared with the original designs.

The schedule latency tends to be smaller for optimized configurations, except when the pipelining algorithm inserts a pipeline register after each primitive

instruction, as in DCT and Luma benchmarks. However, the schedule latency effect on the datapath throughput is very small, since we pipeline the loop iterations that execute on the datapath, and for large number of loop iterations the II value is the main parameter that determines the datapath throughput.

Figure 5.8 depicts the synthesis, placement & routing (SPR) speedup achieved on the standard Xilinx toolset for the optimized versus the original DFGs. Synthesis, placement & routing for grammar-based designs is on average faster than for the original designs achieving an average speedup 1.2x. In CN and BN kernels original designs (without MFUs) in C_B and C_C configurations were processed for over 12 hours before eventually failing to produce fully placed and routed designs because of routing congestion. The DFGs produced for the same benchmarks and configurations by the grammar-based approach succeeded in less than 3 hours. CN kernel achieves the highest speedup (2.2x) among the other benchmarks, mainly because of the significant area reduction attained by the optimized design.

SPR runtime is affected by a wide range of factors. The synthesis phase is affected by the total number of allocated resources and potential logic cells optimizations. The placement & routing runtime is even more sensitive on the size of the generated netlist, routing complexity and user constraints. In Fig. 12, LMC kernel optimized configurations C_A and C_B are slower to SPR than the original configurations. Analysis of the SPR time for LMC showed that both original and optimized designs took the same time for synthesis and routing steps for configurations C_A and C_B. However, the "Global Placement" step, during which the design netlist is placed on the FPGA fabric, the optimized design took more time to finish leading to slower SPR runtime.
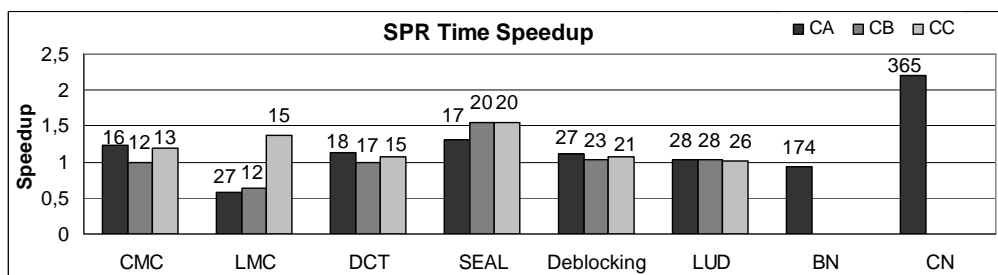

Figure 5.8: Synthesis, Placement & Routing (SPR) Speedup. C_B and C_C bars in BN and CN kernels are missing because the original designs failed to finish placement and routing successfully after 12 hours of runtime, while optimized designs succeeded within 3 hours. The numbers above the bars are the SPR time (in minutes) required for the original, unoptimized DFGs.

126

We did not manage to identify any correlation with any of the design parameters. Moreover, the lack of information on the algorithmic and implementation details of this step in Xilinx tools does not allow us to further reason on the problematic increase in runtime.

Fig. 13 demonstrates the correlation between area reduction and either the number of macro instructions per rule or the DFG coverage for the three configurations, C_A, C_B, and C_C. From Fig. 13 we can conclude that for configurations with large II (and thus complex, large multiplexers as in C_C), the reduction in area is highly correlated with the number of macro-instructions per rule and DFG coverage (correlation equals 0.95 for both cases). As II becomes smaller (and so does the multiplexers overhead), so does the correlation. For configuration C_B where II = 8, the correlation equals 0.8 for both cases. For configuration C_A where II = 1, the correlation of area reduction with the number of macro-instructions per rule and DFG coverage is 0.15 and 0.12 respectively. As explained earlier, in this case the area reduction is expected to come mainly from the pipelining algorithm and not from instruction clustering.

Area and synthesis results demonstrate the effectiveness of the grammar-driven
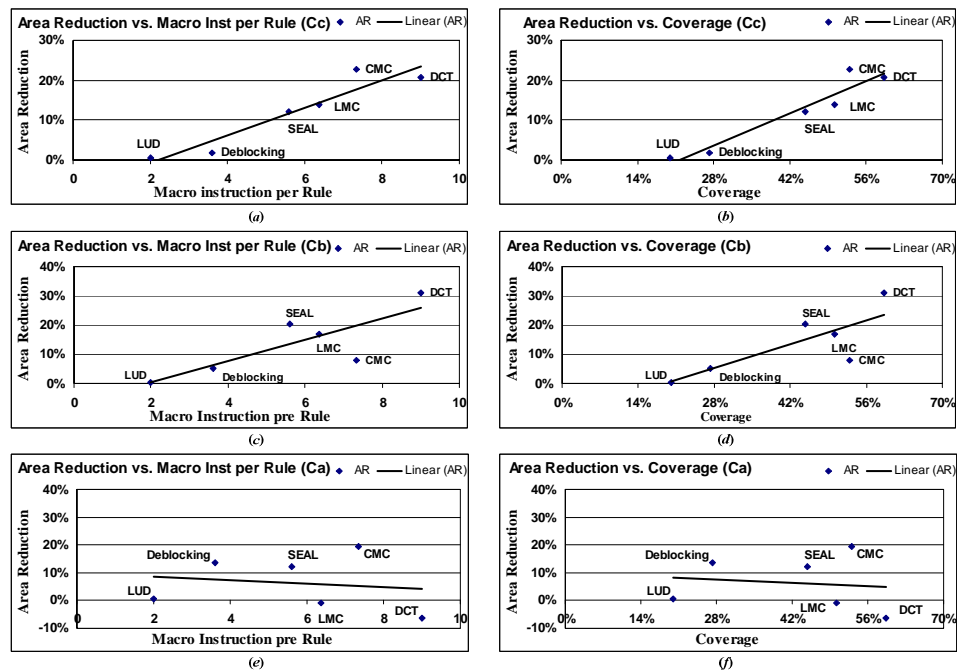

Figure 5.9: Area Reduction (AR) correlation with the number of macro-instructions per grammar rule (a, c, e) and the DFG coverage (b, d, f).

127

approach to reduce the amount of multiplexers and their routing overhead. The generation of MFUs and the selective pipelining algorithm (algorithm 4.5) produced compact macro FUs that performed computations with fewer logic cells and latency. The achieved clock frequency (Table 5.6) for optimized configurations has a deviation between +8% to -1.2% from the original configurations.

In general, the proposed approach achieves higher gain with increasing value of II, in cases where the multiplexer tree has a significant area overhead. Also, for II = 1, significant gain can be achieved if the primitive FUs in each MFU can be packed tighter (high logic gain). If this is not the case, the use of macro-instructions tends to put more constraints on scheduling, increasing the lifetime of variables. The proposed grammar-based algorithm proved to be very fast; in all cases the grammar generation and rules selection took less than a second to finish and to produce a new DFG.

5.6   Cache Allocation Evaluation

The cache unit is useful in holding data across outer loop iterations, especially when the computation of a single data element requires a block of data which will be reused for the computation of following elements. SOpenCL determines allocating a cache if it detects continuous blocks of data reused across loop iterations.

The SEAL kernel has runtime dependent addresses, hence no memory access pattern can be detected and no cache is allocated. The Deblocking kernel has a RAW dependency across outer loops iterations which limit cache utilization. In addition, no data reuse was detected across inner loop iterations. In LUD, CN, and BN kernels, also no data reuse has been detected across loop iterations since data is accessed

Table 5.6: Instruction Clustering Frequency (MHz) results for the test kernels on Xilinx Virtex-6 LX760.

App.	Original			Optimized		
	C_A	C_B	C_C	C_A	C_B	C_C
CMC	165	184	160	166	186	163
LMC	154	161	161	158	164	164
DCT	160	161	163	161	161	163
SEAL	184	184	185	201	201	201
CN	-	-	97	101	85	100
BN	-	-	84	111	100	85
Deblocking	162	162	159	162	161	163
LUD	159	160	163	159	161	163

Table 5.7: FPGA Slices for CMC and LMC kernels with and without cache. N/C refers to configurations with No cache allocated.

	C_A			C_B			C_C		
	Cache	N/C	Overhead	Cache	N/C	Overhead	Cache	N/C	Overhead
CMC	2051	1984	+3.4%	2074	2009	+3.3%	3421	3041	+12.5%
LMC	2989	2487	+20.2	3540	2630	+34.6%	5395	4290	+20.5%

column wise. In DCT data reuse is within each single loop iteration, but non across loop iterations. The tool does not generate a cache to serve only data reuse within single loop iteration, because these are already served by the requests generation unit (RGU).

Only two CMC and LMC kernels have forced SOpenCL to allocate a data cache. Figure 5.8 depicts the data reuse pattern for each kernel. Shaded area represents pixels shared between successive outer loop iterations. Here a row of pixels represents a continuous block of data. Based on the reuse pattern, SOpenCL allocates the following cache blocks for each kernel: 6 blocks of size 32 bytes for luma kernel, and 2 blocks of size 16 bytes for chroma kernel.

Table 5.7 depicts area results for both kernels with and without cache. For both kernels, configurations with cache allocated consume one 36k-bit Block RAM (not shown in the table). Column "Cache" represents configurations with cache enabled. Column "N/C" refers to configurations without cache allocation, and column "Overhead" refers to the area overhead computed as follows:

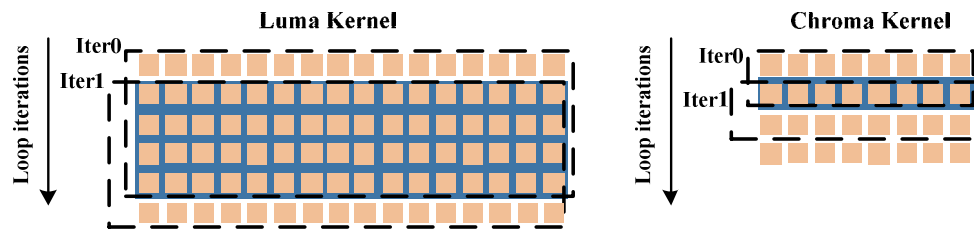$$Overhead = \frac{Slices_{cache} - Slices_{N/C}}{Slices_{N/C}}$$

Figure 5.10: Luma (LMC) and Chroma (CMC) kernels data reuse pattern. The shaded area represents the data (pixels) reused in later outer loop iterations. The pixels surrounded with the dashed rectangle represent the data loaded in a single outer loop iteration.

129

Configurations with cache tend to consume more resources for managing cache data blocks dynamic allocation and incoming read requests. One can see that LMC configurations have higher area overhead than CMC configurations because LMC configurations have more cache blocks allocated.

Figure 5.9 depicts execution time for LMC and CMC cache configurations. The negative percentage value represents the decrease in execution time in cache configurations compared to configurations without cache. Interestingly, one can notice that the execution time reduction percentage correlates with percentage of reused pixels: 50% for CMC kernel, and 80% for LMC kernel.

Cache allocation successfully achieves its goal, reducing memory traffic and increasing performance. For these two benchmarks, performance gain achieved with cache allocation surpasses area overhead.

5.7   Overall Performance Analysis and Comparisons

Figure 5.10 depicts, for each benchmark, the optimal execution time when all optimizations are enabled, for two cases: full accelerator execution (memory transfers + computations) and datapath computations only (i.e. assuming zero cycle memory accesses). The latter case assumes input data always available when needed. The system architecture is a PLB bus based system with peak memory bandwidth equal to 64-bits per clock cycle.
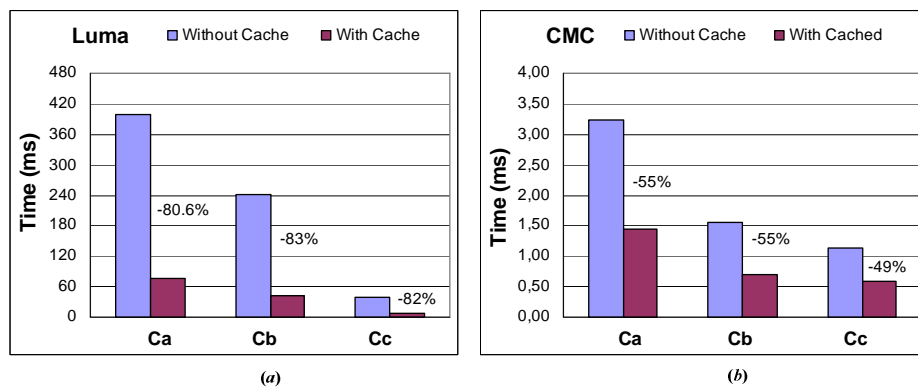

Figure 5.11: Execution time for LMC and CMC configurations with and without cache. The negative percentage value represents the decrease in execution time in cache configurations compared to configurations without cache.

130

Figure 5.10 shows that most kernels are I/O bounded. BN and CN datapaths require 304 bytes/cycle and 256 bytes/cycle respectively, to keep datapath 100% utilized and the memory system provides only 64 bytes/cycle in the best case. While I/O requirements of the deblocking filter are within bus bandwidth limits, execution time spikes when memory transfers are considered. Irregular access patterns push the effective memory bandwidth away from its theoretical peak value. Half of the loop iterations require 10 continuous pixel data per cycle, i.e. pixels are accessed row-wise, and can be served with two read/write requests on the PLB bus. In the second half of loop iterations, each of the 10 bytes requested is in a different frame row, i.e. pixels are accessed column-wise, hence the read/write requests spike to 10 requests.

To better assess the efficacy of our tool flow and methodology to provide high quality designs, we have compared the accelerators generated using SOpenCL with manual, fully optimized designs. Table 5.8 compares Deblocking filter accelerator generated our tool (SOpenCL) with the manual design described in [93]. The throughput numbers are for 1280×720 HD video format (720p). SOpenCL synthesis tool area and clock frequency results are very close to the manual design results. Even with the large gap in throughput SOpenCL produced an accelerator that fullfils real-time requirements (30 frames per second).

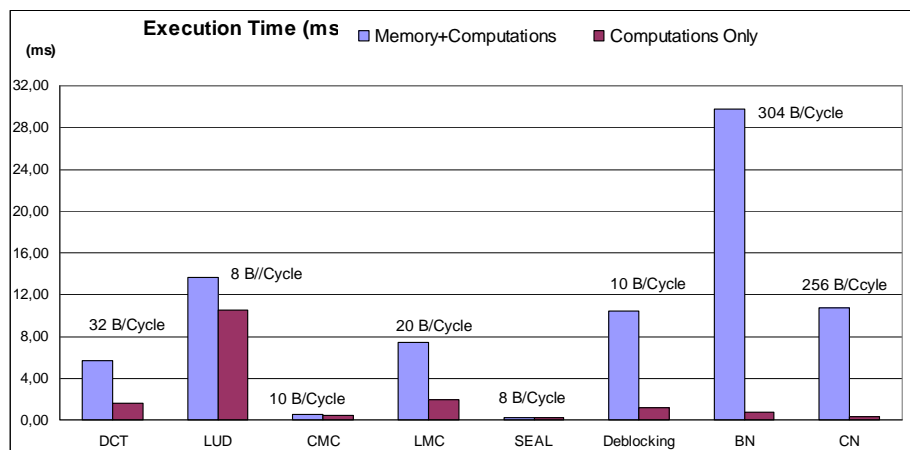The Deblocking filter processes vertical and horizontal edges in every 16x16

Figure 5.12: Comparison of execution time for Memory transfers plus computations and computations only. The numbers above the bars indicate the I/O rate required by each kernel. The Cc configuration with all optimizations have been enabled is used in this figure.

Table 5.9: SOpenCL based design of SEAL kernel compared to manual design. The throughput numbers are for 1 Gbit plaintext messages.

Application	SOpenCL based design Complete Accelerator	Manual design
Slices	2112	1450
Execution Time (second)	8.35	9.3
Frequency (MHz)	201	158

macroblock in a specific sequence: first vertical edges then horizontal edges. As a result, computed pixels only at the corners of horizontal edges are used in later computations of pixels at the corners of vertical edges. This irregular dependency pattern significantly limited the efficiency of the streaming unit in the SOpenCL-based deblocking accelerator. The C code consists of a single nested loop that process both horizontal and vertical edges sequentially which hid potential parallelism between horizontal and vertical edges.

Contrary to SOpenCL generated accelerator, the manual design includes separate datapaths for processing horizontal and vertical edges. Moreover, a specific mechanism has been designed to handle the data dependency that only occurs at the horizontal and vertical edges corners. Extra registers allocated specifically to hold only required pixels for later computations. This special mechanism, allowed more efficient pipelining of successive macroblocks processing.

The manual design only builds the datapath assuming input frame pixels available in On-chip Block RAMs and output pixels are written to another bank on-chip Block RAM. On the other hand, SOpenCL based design requires over 1400 slices for

Table 5.8: SOpenCL based design of Deblocking filter compared to manual design. The throughput numbers are for 1280×720 HD video format (720p). MB latency refers to the number of clock cycles required to complete the processing of a single Macroblock.

Application	SOpenCL based design		Manual design (Datapath Only)
	Complete Accelerator	Datapath Only	
Slices	2714	1295	1430
Throughput (frames/Second)	31	260	379
Frequency (MHz)	161	161	160.5
MB Latency (Clock Cycles)	172	172	118

132

read/write requests for data alignment and synchronization.

Finally, table 5.9 compares SEAL kernel accelerator with the manual design of [8]. The manual design consists of three components: tables generation, initialization and the main body of SEAL encoder. SOpenCL accelerators implement only the last two components, i.e. initialization and the main body. For a 1 Gbit plaintext session, Tables generation components executes only for the first 32 Kbit plaintext message, hence, its execution time overhead can be ignored compared to the main processing operations in the other components.

Our design achieved slightly smaller execution time compared to the manual design with acceptable area overhead 44% FPGA slices. The improvement on execution time was mainly caused by the lower clock frequency achieved via SOpenCL. For a clock frequency similar to the manual design our design would require higher execution time (10.62 ms). The additional area cost in our implementation is due to the input and output streaming units and bus arbitration. The datapath only consumes only 54% of the accelerator area (i.e. 1135 slices).

133

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this dissertation we have investigated and described a methodology to generate hardware accelerators from complex, unmodified OpenCL kernels and C functions. One of the main tasks of this work was the evaluation of the presented methodology which consists of two parts: architectural template design and hardware-driven transformations and optimizations.

The architectural template design and transformation addresses the following issues:

- Generating hardware for imperfect loop nests and data- and control-flow DAGs. The template distinguishes inner most loops code from outer loops code and loop invariant code and maps them on different resources. This mapping paradigm allows arbitrary shapes of loops to be supported for hardware generation.

- Hiding memory latency and overhead through the disassociation of computational operations and data-transfers, effectively facilitating the overlap of computation and communication. Moreover, the template allocates resources and mechanisms to exploit data reuse and reduce memory traffic and bandwidth requirements.

- Exploiting inherent parallelism in OpenCL kernels (and generated C functions) as in task- and pipeline parallelism. The template allows concurrent execution of multiple loop iterations, and pipelines multiple loop nests.

- Customized and application specific datapath design through bitwidth optimization, and instruction clustering. Instruction clustering allows designing optimized application specific functional units which provide improved performance reduced area.

134

All the aforementioned capabilities are based on compiler analysis of memory access patterns, control- and data-dependencies and require no programmer intervention. Equally important, the hardware generator can be tuned to match the available FPGA resources and respect target performance requirement.

We introduced instruction clustering a grammar-based instruction clustering algorithm. Our approach targets the reduction of the routing complexity and overhead in FPGA designs, allowing FPGA implementation of kernels that could not be routed otherwise, such as the DVB-S2 kernels. The core of the methodology is the production of a hierarchical grammar representation of a DFG. The rules of the grammar correspond to subgraphs of the DFG which can be considered as candidate macro-instructions. The proposed algorithm performs the tasks of grammar generation, rule selection and implementation with negligible computation complexity. Furthermore, we presented a simple yet systematic area estimation technique, which can be applied to characterize each target FPGA architecture and toolchain. The results of the area estimation are used to both guide the rules selection phase, and drive the insertion of pipeline registers in the produced macro FUs.

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 model, instruction clustering and data streams caching.

The concurrent execution model proved its efficiency achieving up to 56% increase in performance as in the DCT kernel case. Our analysis showed that applications written in OpenCL kernels with multi-dimensional computations grid will achieve significant performance gain using concurrent execution model.

Decoupled computations (on datapath) and address generation (on AGU), combined with concurrent execution model, efficiently reduced the effect of memory latency on the overall performance. Data prefetching reduced the idle state time gaps of the memory system over the course of a kernel invocation.

Experimental evaluation of data caching proved the effectiveness of the caching mechanism. While the cache utilization is limited to regular data streams, the cache

135

allocation methods consumed small amount of memory (96 byte for CMC and 256 bytes for LMC) to achieve over 50% increase in performance.

Experiments showed the efficiency of the proposed instruction clustering approach in reducing routing complexity and hence reducing area. Moreover, the pipelining algorithm used to design macro functional units, typically produced schedules with smaller latency and no penalty on clock frequency. Most importantly the grammar-driven optimization allowed successful placement and routing on complex designs that were not deemed implementable before.

Instruction clustering and the corresponding algorithms and tool prototypes are another necessary step in the direction of producing efficient FPGA designs from algorithmic descriptions expressed in high level parallel programming languages. This process moves FPGA development closer to the realm of software engineers, thus facilitating the wider adoption and exploitation of FPGAs in everyday, embedded and high-performance computing.

Concluding, the proposed methodology and techniques compared well with manually optimized designs. The generated designs achieved comparable performance with little area overhead.

Hardware generation from high level programming language is a promising technology and the key for promoting FPGA integration in heterogeneous systems. Our research showed that developing a fully automatic architectural synthesis tool that enables software engineers to target FPGA based platforms is not an easy undertaking since it requires extensive analysis of the input programs and sophisticated compiler transformations.

Our future work includes 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 multiple kernels (or multiple instantiations of the same kernel) with multiple accelerators interconnected through customized memory hierarchies. Last but not least, area and performance estimation algorithms are necessary to guide hardware/software partitioning in the high level compiler.

BIBLIOGRAPHY

[1]   Nvidia Inc. "nVIDIA GeForce 256 User Guide", March, 2000. www.nvidia.com

[2]   S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The Reconfigurable Streaming Vector Processor (RSVP^{TM})", *In Proceedings of the International Symposium on Microarchitecture (MICRO)*, December, 2003, San Diego, CA, U.S.A.

[3]   J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das, "Evaluating the Imagine Stream Architecture", *In Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2004, Munich, Germany.

[4]   M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams", *In Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2004, Munich, Germany.

[5]   W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, "Merrimac: Supercomputing with Streams", *In Proceedings of ACM/IEEE International Conference on Supercomputing (SC)*, November 2003, Phoenix, Arizona, U.S.A.

[6]   M. Duranton, D. Black-Schaffer, S. Yehia, and K. De Bosschere, "Computing Systems: Research Challenges Ahead, The HiPEAC Vision 2011/2012", October 2011.

[7]   H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling", *In Proceedings of the International Symposium on Computer Architecture (ISCA)*, June, 2011, San Jose, CA, U.S.A.

[8]   K. Theoharoulis, C. Antoniadis, N. Bellas, and C. D. Antonopoulos, "Implementation and Performance Analysis of SEAL Encryption on FPGA, GPU and Multi-Core Processors", *In Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2011, Salt Lake City, UT, U.S.A.

[9]   Z.H. Chen, W.Y. Su, M.T. Sun and S. Hauck, "Accelerating Statistical LOR Estimation for a High-Resolution PET Scanner using FPGA Devices and a High Level Synthesis

Tool", *In Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2011, Salt Lake City, UT, U.S.A.

[10] S. Muhlbach, and A. Koch, "A Scalable Multi-FPGA Platform for Complex Networking Applications", *In Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2011, Salt Lake City, UT, U.S.A.

[11] B. Sukhwani, B. Abali, B. Brezzo, and S. Asaad, "High-Throughput, Lossless Data Compression on FPGAs", *In Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2011, Salt Lake City, UT, U.S.A.

[12] R. Pottathuparambil, J. Coyne, J. Allred, W. Lynch and V. Natoli, "Low-latency FPGA Based Financial Data Feed Handler", *In Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2011, Salt Lake City, UT, U.S.A.

[13] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman, "PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators", Journal of VLSI Signal Processing Systems, Vol. 31(1), pp 127-142, June 2002.

[14] S. McCloud. "Catapult C Synthesis-based Design Flow: Speeding Implementation and Increasing Flexibility", Mentor Graphics Inc., October 2003.

[15] 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.

[16] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, "A MATLAB Compiler For Distributed, Heterogeneous, Reconfigurable Computing Systems", *In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2000, Napa Valley, CA, U.S.A.

138

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

[18] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of Platform Architectures from OpenCL Programs", *In Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May, 2011, Salt Lake City, UT, U.S.A.

[19] M. Owaida, N. Bellas, C. D. Antonopoulos, K. Daloukas, C. Antoniadis, "Massively Parallel Programming Models Used as Hardware Description Languages: The OpenCL Case", *In Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, November, 2011, San Jose, CA, U.S.A.

[20] G. Falcao, M. Owaida, D. Novo, M. Purnaprajna, N. Bellas, C. D. Antonopoulos, G. Karakonstantis, A. Burg and P. Ienne, "Shortening design time through multiplatform simulations with a portable OpenCL golden-model: the LDPC decoder case", *In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2012, Torronto, Canada.

[21] K. Daloukas, C. D. Antonopoulos, and N. Bellas. "GLOpenCL: OpenCL Support on Hardware- and Software-Managed Cache Multicores". *In Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, January, 2011, Heraklion, Greece.

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

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

[24] L. Chris, and A. Vikram, "LLVM: A Compilation Framework for Lifelong Program Analysis Transformation", *In Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, March, 2004, Palo Alto, CA, U.S.A.

[25] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations", *In Proceedings of the International Conference on VLSI Design (VLS)*, January, 2003, New Delhi, India.

[26] J. L. Tripp, M. B. Gokhale, and K. D. Peterson, "Trident: From High-Level Language to Hardware Circuitry", *IEEE Computer Journal*, Vol. 40(3), pp.28-37, March 2007.

[27] M. Kudlur, K. Fan, and S. Mahlke, "Streamroller: Automatic Synthesis of Prescribed Throughput Accelerator Pipelines", *In Proceedings of International conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS),* October, 2006, Seoul, Korea.

[28] M. Bowen, "Handel-C Language Reference Manual", Embedded Solutions Ltd.

[29] S. Möhl, "The Mitrion-C Programming Language", Mitrionics AB. 2005.

[30] J. Gabriel, F. Coutinho, and W. Luk, "Source-Directed Transformations for Hardware Compilation", *In Proceedings of IEEE International Conference on Field-Programmable Technology (FPT)*, December, 2003, London, UK.

[31] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross, "High-Level Language Abstraction for Reconfigurable Computing", *IEEE Computer Journal*, Vol. 36(8), pp. 63-69, August 2003.

[32] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, and J. R. Beveridge, "Cameron: High Level language Compilation for Reconfigurable Systems", *In Proceedings of International conference on Parallel Architectures and Compilation Techniques (PACT)*, October, 1999, Newport Beach, CA, U.S.A.

[33] M. Gokhale, J. Stone, J. Arnold, "Stream-Oriented FPGA Computing in the Streams-C High Level Language", *In Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2000, Napa, CA, U.S.A.

[34] Impulse Accelerated Technologies Inc, "Impulse Tutorial: Using C-Language Simulation for Algorithm Verification", 2003.

[35] N. Bellas, S. Chai, M. Dwyer, and D. Linzmeier, "FPGA implementation of a license plate recognition SoC using automatically generated streaming accelerators", *Reconfigurable Architectures Workshop (RAW)*, April, 2006, Napa, CA, U.S.A.

[36]   A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: Efficient Realization of Streaming Applications on FPGAs", *In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES),* October, 2008, Atlanta, Georgia, U.S.A.

[37]   Cadence Design Systems, Inc. "Cadence C-To-Silicon Compiler High Level Synthesis", 2008, www.cadence.com.

[38]   Forte Design Systems, Inc. "Cynthesizer: The most productive path to silicon", 2008, www.forteds.com.

[39]   A. Papakonstantinou *et al*. "FCUDA*:* Enabling efficient compilation of CUDA kernels onto FPGAs"*, In Proceedings of the Symposium on Application Specific Processors (SASP)*, July, 2009, Boston, MA, U.S.A

[40]   E. Jääskeläinen, C. S. de La Lama, P. Huerta, and J. Takala, "OpenCL-based Design Methodology for Application-Specific Processors". *In Proceedings of SAMOS X: Embedded Computer Systems: Architectures, MOdeling, and Simulation*, July, 2010, Samos, Greece.

[41]   Altera, Inc. "Implementing FPGA Design with the OpenCL Standard", 2011, www.altera.com.

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

[43]   M. Owaida, N.Bellas, C. D. Antonopoulos, K. Daloukas, Ch. Antoniadis, K. Krommydas and G. Tsoumblekas, "Implementation and Performance Comparison of the Motion Compensation Kernel of the AVS Video Decoder on FPGA, GPU and Multicore Processors", *In Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2011, Salt Lake City, UT, U.S.A.

[44]   C. Zissulescu, T. Stefanov, B. Kienhuis, and Ed Deprettere, ''LAURA: Leiden Architecture Research and Exploration Tool'', *In Proceedings of the International conference on Field Programmable Logic and Applications (FPL)*, September, 2003, Lisbon, Portugal.

[45]   G. Kahn, "The semantics of a simple language for parallel programming", *In Proceedings of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

[46]   B. Kienhuis, E. Rypkema, and E. Deprettere, "Compaan: Deriving process networks from Matlab for embedded signal processing architectures", *In Proceedings of the International Workshop on Hardware/Software Codesign (CODES)*, May, 2000, San Diego, USA.

[47]   S. van Haastregt, and B. Kienhuis, "Automated Synthesis of Streaming C Applications to Process Networks in Hardware", *In Proceedings of Design, Automation & Test in Europe conference (DATE)*, April, 2009, Nice, France.

[48]   J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C With ROCCC 2.0", *In Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, May, 2010, Charlotte, NC, U.S.A.

[49]   I. Lebedev, S. Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, M. Lin, and J. Wawrzynek, "MARC: A Many-Core Approach to Reconfigurable Computing", *In Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2011, Salt Lake City, UT, U.S.A.

[50]   E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow" *IEEE Proceedings*, September, 1987.

[51]   D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, W. W. Hwu, "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture", *In Proceedings of the International Symposium on Computer Architecture (ISCA)*, June, 1998, Barcelona, Spain.

[52]   N. Snavely, S. Debray, G. Andrews, "Predicate Analysis and If Conversion in an Itanium Link Time Optimizer", *In Proceedings of the Workshop on Explicitly Parallel Instruction Set (EPIC) Architectures and Compilation Techniques*, March, 2002, Seattle, Washington, U.S.A.

[53]   W. Chuang, B. Calder, J. Ferrante, "Phi-Predication for Light-Weight If-Conversion", *In Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, March, 2003, San Francisco, CA, U.S.A.

[54]   P. Y. T. Hsu, and E. S. Davidson, "Highly Concurrent Scalar Processing", *In Proceedings of the international symposium on Computer architecture (ISCA)*, June, 1986, Tokyo, Japan.

[55]   Y. Choi, A. Knies, L. Gerke, and T. Ngai, "The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel Itanium Processor", *In Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, November, 2002, Istanbul, Turkey.

[56]   B. R. Rau, D. W.L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 Departmental Supercomputer Design Philosophies, Decisions, and Trade-offs", *IEEE Computer Journal*, Vol. 22(1), pp. 12-26, January 1989.

[57]   R.E. Kessler, "The ALPHA 21264 Microprocessor", *IEEE Micro Journal*, Vol. 19(2), pp. 24-36, March, 1999.

[58]   F. de Ferriere, "Improvements to the Psi-SSA Representation", *In Proceedings of the International Workshop on Software & Compilers for embedded systems (SCOPES)*, April, 2007, Nice, France.

[59]   C. Bruel, "If-Conversion SSA Framework and Transformations", *In Proceedings of the SSA annual meeting*, April, 2009, Monterey, CA, U.S.A.

[60]   M. Weiser, "Program Slicing", *In Proceedings of the International Conference on Software Engineering (ICSE)*, March, 1981, San Diego, CA, U.S.A.

[61]   P. Brisk, P. Kaplan, A. Kastner, and M. Sarrafzadeh, "Instruction Generation and Regularity Extraction for Reconfigurable processors", *In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, October, 2002, Grenoble, France.

[62]   K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under micro-architectural constraints", *In Proceedings of the International Design Automation Conference (DAC),* June, 2003, Anaheim, CA, U.S.A.

[63]   P. Yu, and T. Mittra, "Scalable Custom Instructions Identification for Instruction-Set Extensible Processors", *In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, September, 2004, Washington, DC, U.S.A.

[64]   N. Clark, H. Zhong, and S. Mahlke, "Automatic custom instruction generation for domain-specific processor acceleration", *IEEE Transactions On Computers*, Vol. 54(10) pp. 1258 – 1279, October, 2005.

[65]   D. S. Rao, and F. J. Kurdahi, "On Clustering for Maximal Regularity Extraction", *IEEE Transactions On Computer Aided Design of Integrated Circuits and Systems*, Vol.12(8), pp. 1198-1208, August, 1993.

[66]   J. Cong, and W. Jiang, "Pattern-based behavior synthesis for FPGA resources reduction", *In Proceedings of the international ACM/SIGDA symposium on Field programmable gate arrays (FPGA)*, February, 2008, Monterey, CA, U.S.A.

[67]   N. Manning, H. Witten, and L. Maulsby, "Compression by Induction of Hierarchical Grammars" In *Proceedings of Data Compression Conference (DCC)*, March, 1994, Snowbird, UT, U.S.A.

[68]   M. Lohrey, S. Maneth, and R. Mennike, "Tree Structure Compression with RePair", *In Proceedings of Data Compression Conference (DCC)*, March, 2011, Snowbird, UT, U.S.A.

[69]   J. Cheriyan, and K. Mehlhorn, "Algorithms for Dense Graphs and Networks on the Random Access Computer", Algorithmica, Vol. 15(6) 521-549, June, 1996.

[70]   G. A. Constantinides, "Perturbation Analysis for Word-length Optimization", *In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April, 2003, Napa, CA, U.S.A.

[71]   B. Le Gal, C. Andriamisaina, and E. Casseau, "Bit-Width Aware High Level Synthesis for Digital Signal Processing Systems", *In Proceedings of IEEE International System-on-Chip Conference (SoCC)*, September, 2006, Austin, Texas, U.S.A.

[72]   A. Abdul-Gaffar, O. Mencer, W. Luk, P.Y.K. Cheung, and N. Shirazi, "Floating-point Bitwidth Analysis via Automatic Differentiation", *In Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, December 2002, Hong Kong.

[73]   A. Abdul-Gaffar, O. Mencer, W. Luk, P. Y.K. Cheung, "Unifying Bit-width Optimization for Fixed-point and Floating-point Designs", *In Proceedings of the IEEE*

Symposium on Field-Programmable Custom Computing Machines (FCCM), April, 2004, Napa, CA, U.S.A.

[74]   Y. Pu, and Y. Ha, "An Automated, Efficient and Static Bit-width Optimization Methodology Towards Maximum Bit-width-to-Error Tradeoff With Affine Arithmetic Model", *In Proceedings on the Asia and South Pacific Design Automation Conference (ASP-DAC)*, January, 2006, Yokohama, Japan.

[75]   M. Weinhardt, and W. Luk, "Pipeline Vectorization", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 20(2), February, 2001.

[76]   M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, and M. Rabaey, "Performance Optimization Using Template Mapping for Datapath-Intensive High Level Synthesis", *IEEE Transactions On Computer Aided Design of Integrated Circuits and Systems*, Vol. 15(8) 877-888, November, 2006.

[77]   T. Ly, D. Knapp, R. Miller, and D. Macmillen, "Scheduling using Behavioral Templates", *In Proceedings of the International Design Automation Conference (DAC)*, June, 1995, San Francisco, CA, U.S.A.

[78]   O. Bringmann, and W. Rosenstiel, "Resource Sharing in Hierarchal Synthesis" In *Proceedings of the IEEE/ACM Conference on Computer Aided Design (ICCAD)*, November, 1997, San Jose, CA, U.S.A.

[79]   P. Bonzini, and L. Pozzi, "Polynomial-Time Subgraph Enumeration for Automated Instruction Set Extension", *In Proceedings of the Conference on Design automation and test in Europe (DATE)*, April, 2007, Nice, France.

[80]   D. Goodwin, and D. Petkov, "Automatic Generation of Application Specific Processors", *In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, October, 2003, San Jose, CA, U.S.A.

[81]   J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures", *In Proceedings of the ACM/SIGDA International Symposium on Fieald Programmable Gate Arrays (FPGA)*, February, 2004, Monterey, CA, U.S.A.

145

[82] R. Kastner, S. Memik, E. Bozorgzadeh, and M. Sarrafzadeh, "Instruction Generation for Hybrid Reconfigurable Systems", *ACM Transactions On Design Automation of electronic Systems*, Vol. 7(4) 605-627, October, 2002.

[83] J. Bennett, "A Methodology for automated design of computer instruction sets", PhD. Thesis, University of Cambridge, 1988.

[84] J. V. Praet, G. Goossens, D. Lanneer, and H. Man, "Instruction Set Definition and Instruction Selection for ASIPs", *In Proceedings of the international symposium on High-level synthesis (ISSS)*, May, 1994, Ontario, Canada.

[85] A. Peymandoust, L. Pozzi, P. Ienne, and G. Micheli, "Automatic Instruction Set Extension and Utilization for Embedded Processors", *In Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, June, 2003, Hague, Netherlands.

[86] N. Pothineni, P. Brisk, P. Ienne, A. Kumar, and K. Paul, "A High-Level Synthesis Flow for Custom Instruction Set Extensions for Application-Specific Processors", *In Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, January, 2010, Taipei, Taiwan.

[87] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero, "Swing Modulo Scheduling: A Lifetime-Sensitive Approach", *In Proceedings of International conference on Parallel Architectures and Compilation Techniques (PACT)*, October, 1996, Boston, MA, U.S.A.

[88] B. Ramakrishna Rau, "Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops", 1994, *In Proceedings of the International Symposium on Microarchitecture (MICRO)*, November, 1994, San Jose, CA, U.S.A.

[89] P. G. Paulin, and J. P. Knight, "Scheduling and binding algorithms for high-level synthesis", *In Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, June, 1989, Las Vegas, NV, U.S.A.

[90] W. Gao, S. MA, L. SU, and D. Zhao, "AVS Video Coding Standard", *In Studies in Computational Intelligence*, Vol. 280, 125-166, 2010.

[91] J.A. Michell, J. Solana, and G. Ruiz, "A high-throughput ASIC processor for 8x8 transform coding in H.264/AVC", *In ACM Image Communication*, 26(2), pp. 93-104, February, 2011.

[92] S. Muller, M. Schreger, M. Kabutz, M. Alles, F. Kienle, and N. When, "A novel LDPC decoder for DVB-S2 IP", *In Proceedings of the Conference on Design, Automation and Test in Europe* (DATE), April, 2009, Dresden, Germany.

[93] A. Karapatis, "Implementation of AVS video decoder on FPGA", Undergraduate Diploma, University of Thessaly, July 2011, Volos, Greece.

[94] F. de Dinechin, B. Pasca, and E. Normale, "Custom Arithmetic, Datapath Design for FPGAs using the FloPoCo Core Generator", *IEEE Design & Test of Computers*, Vol. 28(4), August, 2011.

[95] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers", *In Proceedings of the ACM/SIGDA International Symposium on Fieald Programmable Gate Arrays (FPGA)*, February, 1998, Monterey, CA, U.S.A.

[96] K. Beyls, and E.H, D`Hollander, "Reuse Distance as a Metric for Cache Behavior", *In Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, August, 2001, Anaheim, U.S.A.

[97] C. Y. Haung, Y. S. Chen, Y. L. Lin, and Y. C. HSU, "Data Path Allocation Based on Bipartite Weighted Matching", In *Proceedings of the 27th annual ACM/IEEE Design Automation Conference (DAC),* June, 1990, San Orlando, FL, U.S.A.

[98] D. Chen, and J. Cong, "Low-Power High-Level Synthesis for FPGA Architectures", In *Proceedings of the International Symposium On Low Power Electronics and Design (ISLPED),* June, 2003, Seoul, Korea.

[99] J. Cong, Y. Fan, and W. Jiang, "Platform-Based Resource Binding Using a Distributed Register-File Microarchitecture", In *Proceedings of the IEEE/ACM Conference on Computer Aided Design (ICCAD)*, November, 2006, San Jose, CA, U.S.A.