

Chapter 2

Accelerating Data Analytics Kernels with Heterogeneous Computing

Guanwen Zhong, Alok Prakash*, and Tulika Mitra

2.1 Introduction

The past decade has witnessed an unprecedented and exponential growth in the amount of data being produced, stored, transported, processed, and displayed. The journey of zettabyte of data from the myriad of end-user devices in the form of PCs, tablets, smart phones through the ubiquitous wired/wireless communication infrastructure to the enormous data centers forms the backbone of computing today. Efficient processing of this huge amount of data is of paramount importance. The underlying computing platform architecture plays a critical role in enabling efficient data analytics solutions.

Computing systems made an irreversible transition towards multi-core architectures in early 2000. As of now, homogeneous multi-cores are prevalent in all computing systems starting from smart phones to PCs to enterprise servers. Unfortunately, homogeneous multi-cores cannot provide the desired performance and energy-efficiency for diverse application domains. A promising alternative design is heterogeneous multi-core architecture where cores with different functional characteristics (CPU, GPU, FPGA, etc.) and/or performance-energy characteristics (simple versus complex micro-architecture) co-exist on the same die or in the same

* Alok completed this project while working at SoC, NUS

G. Zhong • T. Mitra (✉)

School of Computing, National University of Singapore, Singapore, Singapore

e-mail: guanwen@comp.nus.edu.sg; tulika@comp.nus.edu.sg

A. Prakash

School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore

e-mail: alok@ntu.edu.sg

system. Given an application, only the cores that best fit the application can be exploited leading to faster and power-efficient computing.

Another reason behind the emergence of the heterogeneous computing is the thermal design power constraint [14, 24, 25, 28, 31–33]. While the number of cores on die continues to increase due to Moore’s Law [23], the failure of Dennard scaling [11] has led to rising power density that forces a significant fraction of the cores to be kept powered down at any point in time. This phenomenon, known as the “Dark Silicon” [12], provides opportunities for heterogeneous computing as only the appropriate cores need to switch on for efficient processing under thermal constraints.

Heterogeneous computing architectures can be broadly classified into two categories: *performance heterogeneity* and *functional heterogeneity*. Performance heterogeneous multi-core architectures consist of cores with different power-performance characteristics but all sharing the same instruction-set architecture. The difference stems from distinct micro-architectural features such as in-order core versus out-of-order core. The complex cores can provide better performance at the cost of higher power consumption, while the simpler cores exhibit low-power behavior alongside lower performance. This is also known as single-ISA heterogeneous multi-core architecture [18] or asymmetric multi-core architecture. The advantage of this approach is that the same binary executable can run on all different core types depending on the context and no additional programming effort is required. Examples of commercial performance heterogeneous multi-cores include ARM big.LITTLE [13] integrating high-performance out-of-order cores with low-power in-order cores, nVidia Kal-El (brand name Tegra3) [26] consisting of four high-performance cores with one low-power core, and more recently Wearable Processing Unit (WPU) from Ineda consisting of cores with varying power-performance characteristics [16]. An instance of the ARM big.LITTLE architecture integrating quad-core ARM Cortex-A15 (big core) and quad-core ARM Cortex-A7 (small core) appears in the Samsung Exynos 5 Octa SoC driving high-end Samsung Galaxy S4 and S5 smart phones.

As mentioned earlier, a large class of heterogeneous multi-cores comprise of cores with different functionality. This is fairly common in the embedded space where a multiprocessor system-on-chip (MPSoC) consists of general-purpose processor cores, GPU, DSP, and various hardware accelerators (e.g., video encoder/decoder). The heterogeneity is introduced here to meet the performance demand under stringent power budget. For example, 3G mobile phone receiver requires 35–40 giga operations per second (GOPS) at 1W budget, which is impossible to achieve without custom designed ASIC accelerator [10]. Similarly, embedded GPUs are ubiquitous today in mobile platforms to enable not only mobile 3D gaming but also general-purpose computing on GPU for data-parallel (DLP) compute-intensive tasks such as voice recognition, speech processing, image processing, gesture recognition, and so on.

Heterogeneous computing systems, however, present a number of unique challenges. For heterogeneous multi-cores where the cores have the same instruction-set architecture (ISA) but different micro-architecture [18], the issue is to identify

at runtime the core that best matches the computation in the current context. For heterogeneous multi-cores consisting of cores with different functionality, for example CPU, GPU, and FPGAs, the difficulty lies in porting computational kernels of data analytics applications to the different computing elements. While high-level programming languages such as C, C++, Java are ubiquitous for CPUs, they are not sufficient to expose the large-scale parallelism required for GPUs and FPGAs. However, improving productivity demands fast implementation of computational kernels from high-level programming languages to heterogeneous computing elements. In this chapter, we will focus on acceleration of data analytics kernels on field programmable gate arrays (FPGAs).

With the advantages of reconfigurability, customization, and energy efficiency, FPGAs are widely used in embedded domains such as automotive, wireless communications, etc. that demand high performance with low energy consumption. As the capacity keeps increasing together with better power efficiency (e.g., 16nm UltraScale+ from Xilinx and 14nm Stratix 10 from Altera), FPGAs become an attractive solution to high-performance computing domains such as data-centers [35]. However, complex hardware programming model (Verilog or VHDL) hinders its acceptance to average developers and it makes FPGA development a time-consuming process even as the time-to-market constraints continue to tighten.

To improve FPGA productivity and abstract hardware development using complex programming models, both academia [3, 7] and industry [2, 40, 43] have spent efforts on developing high-level synthesis (HLS) tools that enable automated translation of applications written in high-level specifications (e.g., C/C++, SystemC) to register-transfer level (RTL). Via various optimizations in the form of pragmas/directives (for example, loop unrolling, pipelining, array partitioning), HLS tools have the ability to explore diverse hardware architectures. However, this makes it non-trivial to select appropriate options to generate a high-quality hardware design on an FPGA due to the large optimization design space and non-negligible HLS runtime.

Therefore, several works [1, 22, 29, 34, 37, 39, 45] have been proposed using compiler-assisted static analysis approaches, similar to the HLS tools, to predict accelerator performance and explore the large design space. However, the static analysis approach suffers from its inherently conservative dependence analysis [3, 7, 38]. It might lead to false dependences between operations and limit the exploitable parallelism on accelerators, ultimately introducing inaccuracies in the predicted performance. Moreover, some works rely on HLS tools to improve the prediction accuracy by obtaining performance for a few design points and extrapolating for the rest. The time spent by their methods ranges from minutes to hours and is affected by design space, and number of design points to be synthesized with HLS tools.

In this work, we predict accelerator performance by leveraging a dynamic analysis approach and exploit run-time information to detect true dependences between operations. As our approach obviates the invocation of HLS tools, it enables rapid design space exploration (DSE). In particular, our contributions are two-fold:

- We propose Lin-Analyzer, a high-level analysis tool, to predict FPGA performance accurately according to different optimizations (loop unrolling, loop pipelining, and array partitioning) and perform rapid DSE. As Lin-Analyzer does not generate any RTL implementations, its prediction and DSE are fast.
- Lin-Analyzer has the potential to identify bottlenecks of hardware architectures with different optimizations enabled. It can facilitate hardware development with HLS tools and designers can better understand where the performance impact comes from when applying diverse optimizations.

The goal of Lin-Analyzer is to explore a large design space at an early stage and suggest the best suited optimization pragma combination for an application mapping on FPGAs. With the recommended pragma combination, a HLS tool should be invoked to generate the final synthesized accelerator. Experimental evaluation with different computational kernels from the data analytics applications confirms that Lin-Analyzer returns the optimal recommendation and its runtime varies from seconds to minutes with complex design spaces. This provides an easy translation path towards acceleration of data analytics kernels on heterogeneous computing systems featuring FPGAs.

2.2 Motivation

As the complexity of accelerator designs continues to rise, the traditional time-consuming manual RTL design flow is unable to satisfy the increasingly strict time-to-market constraints. Hence, design flows based on HLS tools such as Xilinx Vivado HLS [43] that start from high-level specifications (e.g., C/C++/SystemC) and automatically convert them to RTL implementations become an attractive solution to designers.

The HLS tools typically provide optimization options in the form of pragmas/directives to generate hardware architectures with different performance/area trade-offs. Pragma options like loop unrolling, loop pipelining, and array partitioning have the most significant impact on hardware performance and area [8, 21, 44]. Loop unrolling is a technique to exploit instruction-level parallelism inside loop iterations, while loop pipelining enables different loop iterations to run in parallel. Array partitioning is used to alleviate memory bandwidth constraints by allowing multiple data reads or writes to be completed in one cycle.

However, this diverse set of pragma options necessitate designers to explore a large design space to select the appropriate set of pragma settings that meets performance and area constraints in the system. The large design space created by the multitude of available pragma settings makes the design space exploration a significantly time-consuming work, especially due to the non-negligible runtime of HLS tools using the DSE step. We highlight the time complexity of this step by using the example of Convolution3D kernel, typically used in big data domain.

Listing 2.1 Convolution3D kernel

```

...
/* Constant values of a window filter: {c11,...,c21,...,c33} */
loop_1: for (i = 1; i < N-1; i++) {
  loop_2: for (j = 1; j < M-1; j++) {
    loop_3: for (k = 1; k < K-1; k++) {
      b[i][j][k] = c11*a[i-1][j-1][k-1] +
                  c13*a[i+1][j-1][k-1] + c21*a[i-1][j-1][k-1] +
                  c23*a[i+1][j-1][k-1] + c31*a[i-1][j-1][k-1] +
                  c33*a[i+1][j-1][k-1] + c12*a[i][j-1][k] +
                  c22*a[i][j][k] + c32*a[i][j+1][k] +
                  c11*a[i-1][j-1][k+1] + c13*a[i+1][j-1][k+1] +
                  c21*a[i-1][j][k+1] + c23*a[i+1][j][k+1] +
                  c31*a[i-1][j+1][k+1] + c33*a[i+1][j+1][k+1];
    }
  }
}

```

Table 2.1 HLS runtime of Convolution3D

Input size	Loop pipelining	Loop unrolling	Array partitioning	HLS runtime
32*32*32	Disabled	loop_3 factor:30	a, cyclic, 2 b, cyclic, 2	44.25 s
	loop_3, yes	loop_3 factor:15	a, cyclic, 16 b, cyclic, 16	1.78 h
	loop_3, yes	loop_3 factor:16	a, cyclic, 16 b, cyclic, 16	3.25 h

Table 2.2 Exploration time of convolution 3D: exhausted vs. Lin-Analyzer

Input size	Design space	Exploration time	
		Exhaustive HLS-based DSE	Lin-Analyzer
32*32*32	120	10 days ^a	29.30 s

^aFor few design points with complex pragmas, the HLS tool takes a long time and thus we stop the program after 10 days

Listing 2.1 shows the Convolution3D kernel. We use a commercial HLS tool, Xilinx Vivado HLS [43], to generate an FPGA-based accelerator for this kernel with different pragma combinations and observe the runtime for this step, as shown in Table 2.1. It is noteworthy that the runtime varies from seconds to hours for different choices of pragmas. As the internal workings of the Vivado HLS tool is not available publicly, we do not know the exact reasons behind this highly variable synthesis time. Other techniques proposed in the existing literature, such as [29], that depend on automatic HLS-based design space exploration are also limited by this long HLS runtime.

Next, we perform an extensive design space exploration for this kernel using the Vivado HLS tool by trying the exhaustive combination of pragma settings. Table 2.2 shows the runtime for this step. It can be observed that even for a relatively smaller input size of (32 * 32 * 32), HLS-based DSE takes more than 10 days.

However, in order to find good-quality hardware accelerator designs, it is imperative to perform the DSE step rapidly and reliably. This provides designers with important information about the accelerators, such as FPGA performance/area/power at an early design stage. For these reasons, we develop Lin-Analyzer, a pre-RTL, high-level analysis tool for FPGA-based accelerators. The proposed tool can rapidly and reliably predict the effect of various pragma settings and combinations on the resulting accelerator’s performance and area. As shown in the last column of Table 2.2, Lin-Analyzer can perform the same DSE as the HLS-based DSE, but in the order of seconds versus days. In the next section, we describe the framework of our proposed tool.

2.3 Automated Design Space Exploration Flow

The automated design space exploration flow leverages the high-level FPGA-based performance analysis tool, Lin-Analyzer [46], to correlate FPGA performance with given optimization pragmas for a target kernel in the form of nested loops. With the chosen pragma that leads to the best predicted FPGA performance within resource constraints returned by Lin-Analyzer, the automated process invokes HLS tools to generate an FPGA implementation with good quality. The overall framework is shown in Fig. 2.1. The following subsections describe more details in Lin-Analyzer.

2.3.1 The Lin-Analyzer Framework

Lin-Analyzer is a high-level performance analysis tool for FPGA-based accelerators without register-transfer-level (RTL) implementations. It leverages dynamic analysis method and performs prediction on dynamic data dependence graphs (DDDGs) generated from program traces. The definition of DDDG is given below.

Definition 1 A DDDG is a directed, acyclic graph $G(V_G, E_G)$, where $V_G = V_{op}$ and $E_G = E_r \cup E_m$. V_{op} is the set containing all operation nodes in G . Edges in E_r represent data dependences between register nodes, while edges in E_m denote data dependences between memory load/store nodes.

As the DDDG is generated from a trace, basic blocks of the trace have been merged. If we apply any scheduling algorithms on DDDG, operations can be scheduled across basic blocks. The inherent feature of using dynamic execution

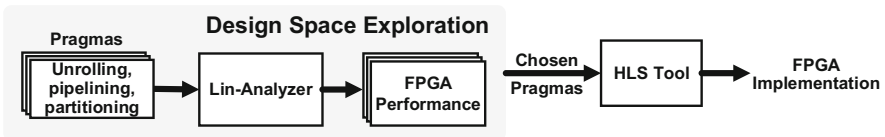


Fig. 2.1 The proposed automated design space exploration flow

trace is that it automatically enables global code motion optimization. In contrast, almost all of the current state-of-the-art HLS tools use static analysis and therefore, need to leverage advanced scheduling algorithms such as System of Difference Constraints (SDC) scheduling [4, 7, 43, 44] to perform global code optimization. However, the inherent feature of the dynamic trace coupled with the dataflow nature of accelerators makes DDDG a good candidate for modeling hardware behavior [38].

With the DDDG, Lin-Analyzer mimics HLS tools and estimate performance of FPGA-based accelerators directly from algorithms in high-level specifications such as C/C++ without generating RTL implementations.

2.3.2 Framework Overview

Figure 2.2 shows the Lin-Analyzer framework. As we can see, Lin-Analyzer consists of three stages: *Instrumentation*, *DDDG Generation & Pre-optimization* and *DDDG Scheduling*. It starts from high-level specifications (C/C++) of an

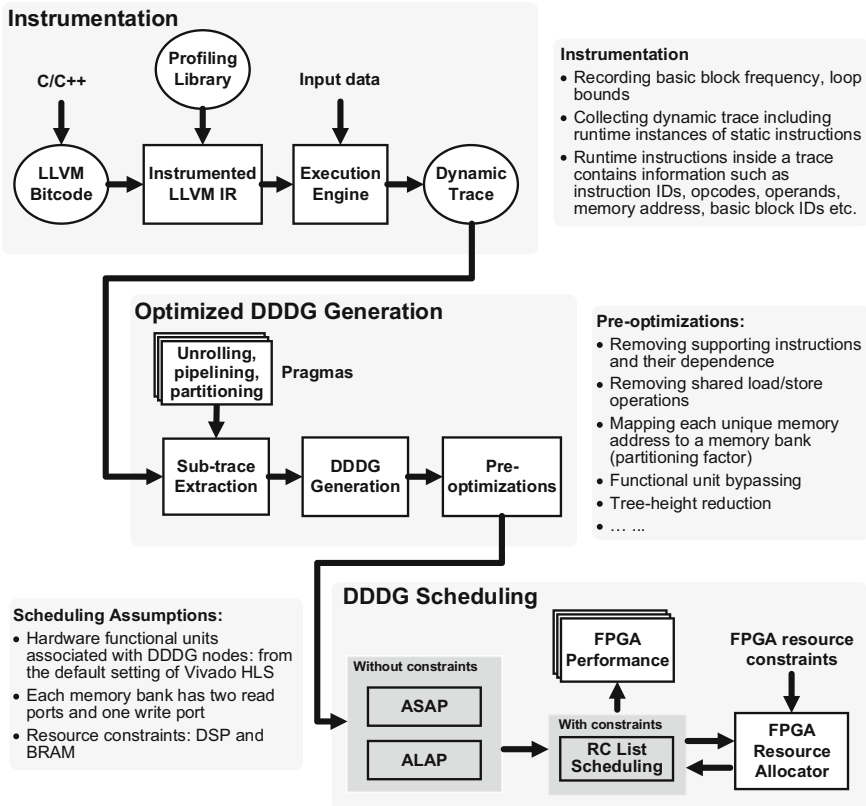


Fig. 2.2 The Lin-Analyzer framework

algorithm with changes and optimizations. By inserting profiling functions into the original codes, Lin-Analyzer collects dynamic trace in *Instrumentation* stage. According to pragmas provided by users, Lin-Analyzer extracts a *sub-trace* from the dynamic trace and builds a dynamic data dependence graph (DDDG) to represent hardware accelerators. As initial DDDG usually contains unnecessary information and needs to be optimized, Lin-Analyzer performs pre-optimizations on it and creates a new DDDG. With the optimized DDDG, it schedules the nodes with resource constraints and estimates performance of the FPGA-based accelerator for the given algorithm. As the analysis is based on DDDG of the relevant *sub-trace* and utilizes fast scheduling algorithm, runtime of Lin-Analyzer is small even for kernels with relative large data size and complex pragma combination such as complete loop unrolling, large array partitioning factors, and loop pipelining.

2.3.3 Instrumentation

A program trace of the kernel containing dynamic instance of static instructions is required for DDDG generation. In this work, we utilize the Low-Level Virtual Machine (LLVM) [19] to instrument programs and collect traces. LLVM leverages passes to perform code analysis, optimization, and modification based on a machine-independent intermediate representation (IR), which is a Static Single Assignment (SSA) based representation.

Lin-Analyzer first converts an application in C/C++ into LLVM IR and instrument the IR by inserting profiling functions. The profiling functions are implemented in the *Profiling Library* and used to record basic block frequency and trace information. With the instrumented LLVM IR, Lin-Analyzer invokes the embedded *Execution Engine*, an LLVM Just-in-Time (JIT) compiler, to run the IR with input data if available. After execution, a run-time trace is dumped into the disk. The dynamic trace includes runtime instances of static instructions and detailed information can be found in Fig. 2.2.

2.3.4 Optimized DDDG Generation

To perform analysis on whole dynamic trace is inefficient and slow, as trace typically contains million or even billion of instruction instances. Therefore, Lin-Analyzer only focuses on a subset of the trace and creates a DDDG for the *sub-trace*. Size of the *sub-trace* is based on pragmas given by users. The initial generated DDDG usually includes unnecessary operations and dependences, and is not good enough to represent hardware accelerators. Thus, Lin-Analyzer performs pre-optimizations on DDDGs before scheduling.

2.3.4.1 Sub-trace Extraction

Size of a *sub-trace* is related to loop unrolling and loop pipelining pragmas. A kernel in the form of a nested (perfectly or non-perfectly) loop can be represented by $L = \{L_1, \dots, L_i, \dots, L_K\}$ with K loop levels and the innermost loop level is L_K . Users can apply loop unrolling pragma at any loop levels in L . Assume a given unrolling factor tuple is $\{U_1, \dots, U_i, \dots, U_K\}$, where U_i is the factor of i -th loop level. Lin-Analyzer extracts U_i iterations of Loop L_i as the *sub-trace* if its inner loops ($\{L_{i+1}, L_{i+2}, \dots, L_K\}$) are completely unrolled; otherwise, the *sub-trace* only includes U_K iterations of Loop L_K .

According to Vivado HLS [43], the HLS tool only considers loop pipelining when the pipelining pragma is applied at one loop level L_i in L and all its inner loops ($L' = \{L_{i+1}, L_{i+2}, \dots, L_K\}$) are forced to be completely unrolled irrespective of their unrolling factors. In this case, the *sub-trace* contains all instruction instances of the inner loops L' . If L_i is the innermost loop level ($i = K$), Lin-Analyzer extracts U_K iterations of L_K as the *sub-trace*.

2.3.4.2 DDDG Generation & Pre-optimizations

Once the *sub-trace* is ready, Lin-Analyzer generates a dynamic data dependence graph (DDDG) to represent the hardware accelerator. In our implementation, a node in the DDDG represents a dynamic instance of an LLVM IR instruction, while an edge represents register- or memory-dependence between nodes. We only consider true dependences. Anti- or output-dependences are not included, as they could be potentially eliminated by optimizations. As we work with dynamic traces, control dependences are not considered.

The initial generated DDDG normally contains supporting instructions and dependences between loop index variables, which cannot model hardware accelerators properly [38]. Therefore, we perform several optimizations before scheduling.

- *Removing supporting instructions and their dependences:* Some of the instructions in a nested loop are related to computation directly, while others are supporting instructions that are used to keep computation in the correct sequence such as instructions related to loop indices, instructions used to obtain memory address of a pointer or based address of an array, etc. Those instructions might potentially introduce true dependences that are not relevant to actual computation, for example, dependence between loop index variables. To remove those information in DDDG, Lin-Analyzer assigns zero latency to those nodes.
- *Removing redundant load/store operations:* A program might potentially contain redundant memory accesses (load or store). This redundancy increases memory (BRAM) bandwidth requirement of a hardware accelerator. To save memory bandwidth, Lin-Analyzer removes redundant memory access operations.

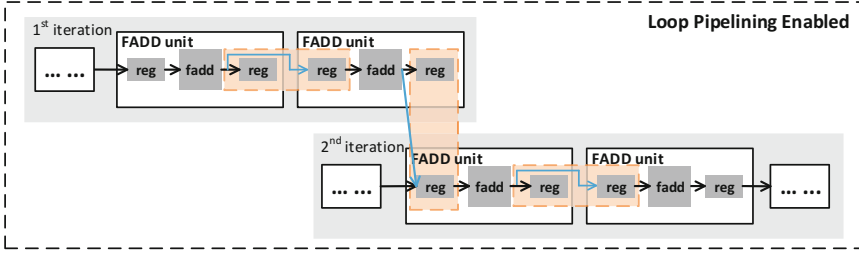


Fig. 2.3 Functional unit bypassing used in Vivado HLS when loop pipelining enabled

- *Associating memory banks with memory addresses:* In our implementation, we assume a memory bank only allows one write and two reads at the same cycle. To restrict the DDDG with the memory constraint, Lin-Analyzer maps each unique address of load/store instructions to a memory bank index. The number of memory banks supported is related to array partitioning factors provided by users.
- *Tree height reduction:* An application might sometimes contain long expression chains. To expose potential parallelism and reduce height of the chains, we employ tree height reduction similar to Shao's work [38].
- *Functional unit bypassing:* When applying loop pipelining and unrolling pragmas at the innermost loop level, we observe (through RTL simulation) that HLS tool (Vivado HLS [43]) enables functional unit bypassing optimization, which is shown in Fig. 2.3. The optimization bypasses output registers of pipelined functional units and directly sends results to the next units connected. Lin-Analyzer also enables similar optimization when users apply loop pipelining and unrolling pragmas at the innermost loop level.

2.3.5 DDDG Scheduling

Lin-Analyzer leverages Resource-Constrained List Scheduling (RCLS) algorithm to schedule nodes on a DDDG. The algorithm takes the optimized DDDG generated by the previous stage and a priority list as inputs. The priority list is obtained from As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) scheduling policies. The RCLS algorithm works with the following assumptions:

1. Nodes in the DDDG are associated with hardware functional units. Configurations of those units follow the default setting of Vivado HLS such as functional types, latencies, and resource consumption;
2. Data is stored into memory banks (FPGA BRAM) and each bank supports one write and two reads at the same cycle;

3. Nodes that are supporting instructions are removed by assigning zero latency;
4. As most of the accelerator designs are restricted by BRAM and DSP resource, in current implementations, we only consider these two resource constraints.

Based on the above assumptions, Lin-Analyzer finds the minimum latency of the DDDG utilizing ASAP policy, which only schedules a node with the condition that all predecessors of the node are completed. ALAP policy schedules a node as late as possible when its successors are all finished. RCLS scheduling takes timestamps (a priority list) of nodes returned by ALAP scheduling as an input. Both ASAP and ALAP have no resource limitation, which is infeasible. Therefore, Lin-Analyzer leverages resource-constrained list scheduling policy to obtain a feasible schedule of minimum latency within FPGA resource constraints.

The RCLS policy schedules a DDDG node with the following conditions:

- All predecessors of the node have been scheduled and completed;
- Among the unscheduled ready nodes, the node has the highest priority;
- There are sufficient FPGA resources for allocating the node.

The resource management (allocation and release) is provided by FPGA Resource Allocator (FRA). To schedule a type T node, FRA checks if there exists an allocated T functional unit available. If all T units are occupied and there are still sufficient resources, FRA allocates a new T functional unit for the node and records its occupied status; otherwise, the node is assigned with an available allocated T functional unit. Functional units consist of pipelined and non-pipelined designs. In this work, we utilize pipelined functional units for floating-point operations and the rest uses non-pipelined units. For pipelined units, if a node using this kind of unit is scheduled, the occupied pipelined unit will be released in the next cycle by FRA. For non-pipelined unit, an occupied functional unit will be released only if the associated node finishes.

When RCLS policy finishes scheduling all nodes in the DDDG, Lin-Analyzer obtains the final schedule and execution latency of the DDDG. With the loop bounds and latency of *sub-trace*, Lin-Analyzer predicts execution cycles of an FPGA-based design for the kernel.

2.3.6 Enabling Design Space Exploration

Designers can use HLS tools to develop diverse hardware implementations by inserting various optimization pragmas. The three prominent pragmas, loop unrolling, loop pipelining, and array partitioning, have significant impact on FPGA performance and resource consumption [9, 29]. Therefore, the three pragmas are supported in this work and Lin-Analyzer enables rapid design space exploration with this feature.

Loop Unrolling With this optimization, HLS tools can schedule instructions of multiple loop iterations and exploit more instruction-level parallelism. To mimic

the optimization on HLS tools, Lin-Analyzer properly selects size of *sub-trace* according to unrolling factors as explained in Sect. 2.3.4.1 and predicts performance on the optimized DDDG generated. Assume that designers provide loop unrolling optimization with factor u , Lin-Analyzer extracts a *sub-trace* containing dynamic instruction instances of u loop iterations and generates an initial un-optimized DDDG. After pre-optimization in Sect. 2.3.4.2, Lin-Analyzer schedules nodes in the new generated optimized DDDG with RCLS policy and obtains latency IL of the *sub-trace* according to loop unrolling configuration. With loop bounds, unrolling factor u and latency IL of the *sub-trace*, Lin-Analyzer predicts performance of the FPGA-based accelerator.

Loop Pipelining Operations in a loop iteration i are executed in sequence. The next iteration $i + 1$ of the loop can only start execution when all operations inside the current loop iteration i are complete. Loop pipelining optimization enables operations in the next loop iteration $i + 1$ begin execution without waiting for the current loop iteration i to be finished. This concurrent execution manner significantly improves performance of hardware accelerators. With pipelining optimization enabled, performance of an accelerator is determined by an initiation interval (II) of the loop. II is a constant clock cycle period required between the start of two consecutive loop iterations. To predict performance of accelerators with loop pipelining enabled, Lin-Analyzer does not perform scheduling and calculates the minimum initiation interval (MII) to approximate the II instead. This can reduce the size of *sub-trace* and help to reduce Lin-Analyzer's runtime. The calculation of MII is done by the following Eqs. (2.1)–(2.4),

$$MII = \max(RecMII, ResMII) \quad (2.1)$$

$$ResMII = \max(ResMII_{mem}, ResMII_{op}) \quad (2.2)$$

$$ResMII_{mem} = \max_m \left(\left\lceil \frac{R_m}{RPorts_m} \right\rceil, \left\lceil \frac{W_m}{WPorts_m} \right\rceil \right) \quad (2.3)$$

$$ResMII_{op} = \max_n \left(\left\lceil \frac{Fop_Par_n}{Fop_used_n} \right\rceil \right) \quad (2.4)$$

where $RecMII$ is the recurrence-constrained MII and $ResMII$ is the resource-constrained MII . $ResMII_{mem}$ is used to analyze MII that is restricted by memory bandwidth, while $ResMII_{op}$ is limited by number of floating-point hardware units. The number of memory read and write operations of array m within a pipelined stage are represented by R_m and W_m , respectively. The number of read and write ports of array m depends on number of memory banks associated, which is related with array partitioning factors. The available number of read and write ports of array m are denoted by $RPorts_m$ and $WPorts_m$, respectively. Fop_Par_n and Fop_used_n are the number of floating-point functional unit of type n returned by ALAP scheduling and RCLS policy, respectively. Fop_Par_n denotes the maximum number of functional units that can run simultaneously without resource constraints.

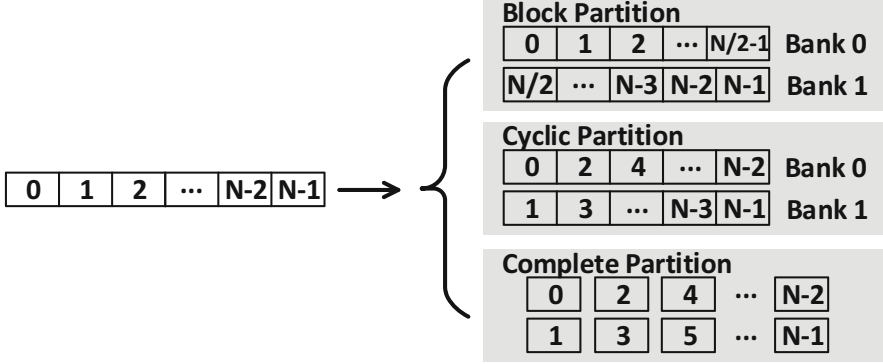


Fig. 2.4 Array partitioning example for three strategies with factor 2 [43]. For simplicity, the partitioning factor used here is two

In this work, we use latency IL of the selected *sub-trace* as its pipeline depth. With loop bounds, pipeline depth IL , and the estimated MII , Lin-Analyzer predicts performance of the FPGA-based accelerator using the equation in [20].

Array Partitioning Data in FPGA-based accelerator is stored into one or multiple memory banks which are composed of FPGA BRAM resource. As memory ports per bank are limited, the number of read/write through the same bank at the same cycle is restricted (we assume two-read and one-write ports per memory bank). Accelerators might suffer from this memory bandwidth bottleneck. In Vivado HLS [43], it supports array partitioning pragma to split data into multiple memory banks to improve the bandwidth. The partitioning strategies include three types, *block*, *cyclic*, and *complete* as shown in Fig. 2.4. To simulate array partitioning optimization, Lin-Analyzer first maps addresses of load and store operations in the DDDG to memory banks, and leverages FRA to keep track of read/write ports used each bank and prevent RCLS scheduling from violating memory port constraints. Memory bank number $Bank_N_m$ related to array partitioning factor is calculated as below,

$$Bank_N_m = \begin{cases} (addr_m) / (\lceil size_m / pf \rceil) & \text{if block} \\ (addr_m) \text{ modulo } (pf) & \text{if cyclic} \end{cases} \quad (2.5)$$

where $addr_m$ represents a memory address of array m , $size_m$ denotes array size of m , and pf describes the partition factor. Memory-port constraint is released for *complete* array partitioning, as the whole array is implemented with registers.

An Example Figures 2.5 and 2.6 show two examples to describe how Lin-Analyzer estimates FPGA performance when given different pragmas. In the examples, the *fadd* functional unit has 5-cycle latency and it is a pipelined design. Memory operations (load and store) have 1-cycle latency. These FPGA node latencies follow the default setting of Vivado HLS.

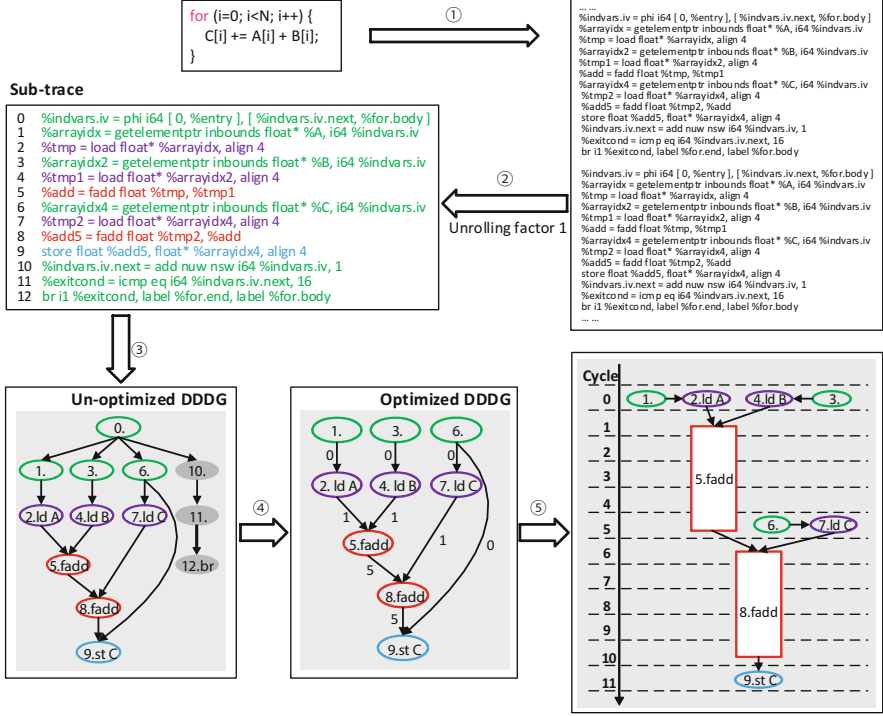


Fig. 2.5 An example without optimization pragma: ① Lin-Analyzer instruments the source code and generates dynamic trace; ② Given loop unrolling factor uf ($uf = 1$, which means no optimization), Lin-Analyzer extracts dynamic instruction instances of one loop iteration as a *sub-trace*; ③ With the *sub-trace*, our tool generates an un-optimized DDDG to represent the hardware accelerator; ④ Lin-Analyzer performs pre-optimizations on the un-optimized DDDG; ⑤ Lin-Analyzer performs RCLS scheduling on the optimized DDDG. Latency IL of the *sub-trace* returned from the scheduling graph is 12 cycles and the total FPGA execution cycle of the loop is $(12 * N)$ cycles, where N is the loop bound

Figure 2.5 shows the example without optimization, which means that loop unrolling uf and array partitioning factors pf are equal to 1 and no loop pipelining is enabled. In Fig. 2.5, the instructions in the *sub-trace* highlighted in green are *supporting* instructions, which are used to keep computation being carried out in the correct manner. Lin-Analyzer removes the *supporting* instructions by assigning zero-latency as their edge weights. As we can see that, there is a true dependence between Instruction 0 and 10, which are related to loop indices. This kind of dependence is removed after performing optimization on an initial DDDG. With the optimizations mentioned in Sect. 2.3.4.2, Lin-Analyzer schedules the DDDG leveraging RCLS policy. The final scheduling graph is shown in Fig. 2.5. Array A , B , and C consume only one memory bank (BRAM consumption depends on their size) because of $pf = 1$ and can support two-read and one-write operations simultaneously. Based on the scheduling, we know the latency of uf

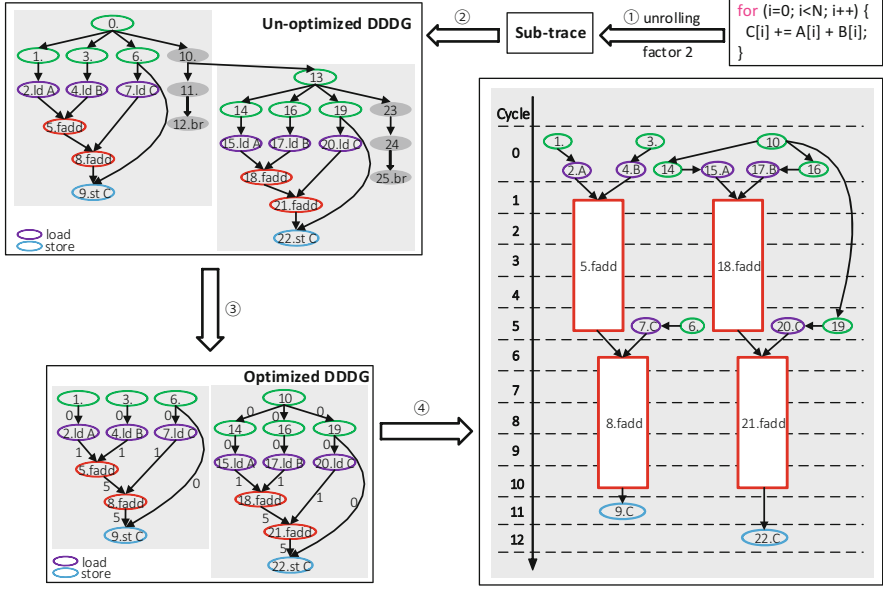


Fig. 2.6 An example with loop unrolling pragma ($uf = 2$): With a new optimization pragma ①, Lin-Analyzer extracts a new *sub-trace*, creates an initial DDDG accordingly ②, performs pre-optimization to generate the optimized DDDG ③, and schedules nodes on the DDDG. Latency IL of the *sub-trace* returned from the scheduling graph is 13 cycles in this case and the total FPGA execution cycle of the loop with unrolling factor $uf = 2$ is $(13 * N/2)$ cycles

loop iterations and Lin-Analyzer predicts its FPGA performance for this kernel without optimization. In this example, the hardware accelerator uses one 32-bit *fadd* functional unit, which consumes 2 DSPs.

Figure 2.6 shows the same example with loop unrolling enabled ($uf = 2$). As the time spent on collecting the whole trace is a one-time cost, Lin-Analyzer reuses the whole trace from the previous example and extracts a new *sub-trace* according to loop unrolling pragma provided. With the new *sub-trace*, Lin-Analyzer follows similar steps in the previous example and predicts performance of the hardware accelerator with loop unrolling enabled without generating any RTL implementations. In Fig. 2.6, Instruction 9 and 22 in blue are used to store data in Array C. As we do not enable array partitioning pragma ($pf = 1$), Array C only allows one write operation per cycle due to memory bank constraint and thus Lin-Analyzer spends two cycles on Instruction 9 and 22. In this example, the hardware accelerator shares one 32-bit *fadd* functional unit, which consumes 2 DSPs.

When we apply loop pipelining pragma on the example in Fig. 2.5, we follow the same steps in the figure and calculate the initiation interval II with Eqs. (2.1)–(2.4). In the example, there is no recurrence loop dependence and thus $RecMII$ is 0. The number of memory read and write operations (R_A and W_A) of Array A within a pipelined stage from the figure is 1 for both. The available number of read and write ports of Array A are 2 and 1, respectively. With Eq. (2.3) for Array A, we have

$\lceil \frac{R_A}{R_{Ports_A}} \rceil = \lceil \frac{1}{2} \rceil = 1$, while $\lceil \frac{W_A}{W_{Ports_A}} \rceil = \lceil \frac{1}{1} \rceil = 1$. Data of Array B is the same with that of A . For Array C , R_C is 0, as there is no memory read. According to Eq. (2.4), $ResMII_{mem}$ is 1, which means that the hardware accelerator with the given configuration is not constrained by memory ports. The maximum number of *fadd* functional unit Fop_Par_{fadd} returned by ALAP scheduling is 1, and the number of *fadd* functional unit Fop_used_{fadd} used in RCLS scheduling is 1. Based on Eq. (2.4), we get $ResMII_{op} = 1$. We calculate $MII = 1$ with Eq. (2.1) and use its value to approximate the II . The pipeline depth of the accelerator leverages the latency of the *sub-trace*, which is $IL = 12$. Therefore, the total FPGA execution cycle of the hardware accelerator with loop pipelining enabled is $(II * (N - 1) + IL = N - 1 + 12 = N + 11)$ cycles. As we can see in Fig. 2.5, the two *fadd* instructions, *5.fadd* and *8.fadd*, can start execution at every $6 * i$ ($i \in [1, 2, \dots]$) cycles simultaneously and thus with loop pipelining enabled, the hardware accelerator consumes two 32-bit *fadd* functional units, which uses 4 DSPs.

From the above examples, Lin-Analyzer can explore different hardware architectures of a kernel rapidly by changing combinations of pragmas without any RTL implementations. This ability makes Lin-Analyzer can explore and evaluate a large design space of hardware implementations in the order of seconds to minutes. However, similar to other works using dynamic analysis [15, 38, 41], if different program inputs have significant impacts on behaviors of an application, Lin-Analyzer might also suffer from inaccuracy when predicting performance. In this case, selecting a representative input for generating trace is necessary and crucial. Moreover, in current implementation, as Lin-Analyzer only optimizes for FPGA performance, it tries to use available resources as much as possible if necessary. Area-performance tradeoff in accelerator design will be included inside our framework in future.

2.4 Acceleration of Data Analytics Kernels

The experiment is set up on a computer with an Intel Xeon CPU E5-2620 running at 2.10 GHz with 64 GB RAM and the OS used is Ubuntu 14.04. We leverage Xilinx Vivado HLS version 2014.4 as the HLS tool and frequency of accelerators is set to 100MHz. Our target FPGA device is Xilinx ZC702 Evaluation Kit [43]. We select four kernels related to big-data applications for evaluation.

- **GEMM:** This kernel is a generic matrix–matrix multiplication application from Polybench Benchmark Suite [30]. It is widely used in machine learning applications such as Convolutional Neural Network [36].
- **KMeans:** This kernel is a clustering algorithm, which is used extensively in data-mining. It is modified from Rodinia Benchmark Suite [6].
- **CONV2D & CONV3D:** Convolution 2D/3D can be used to implement edge detection and smoothing as a filter. It is an important computation in signal/image processing, machine learning, and elsewhere [17, 42]. The two kernels are adapted from Polybench Benchmarks Suite GPU version [30].

2.4.1 Estimation Accuracy

Given loop unrolling, loop pipelining, and array partitioning pragmas, Lin-Analyzer predicts FPGA performance for kernels in C/C++. As Vivado HLS is based on static analysis, it might conservatively add false loop-carried dependences and limit the exploitable parallelism of accelerators. To analyze estimation quality of Lin-Analyzer, we describe prediction accuracy separately for different pragma combinations.

2.4.1.1 Loop Unrolling and Loop Pipelining

Considering loop unrolling and loop pipelining pragmas, Fig. 2.7 shows the performance (execution cycle counts) comparison of Lin-Analyzer and Vivado HLS for *GEMM*, *KMeans*, *CONV2D*, and *CONV3D* kernels. The Y-axis denotes the execution cycle counts of different configurations, while the X-axis describes various configuration combinations consisting of loop unrolling and loop pipelining. As we can see from the figure, the predicted performance from Lin-Analyzer (the yellow dashed lines with triangles) matches the ones from Vivado HLS (the green solid lines with stars) very closely for all four kernels. The average difference between the execution cycle counts returned from Vivado HLS and the

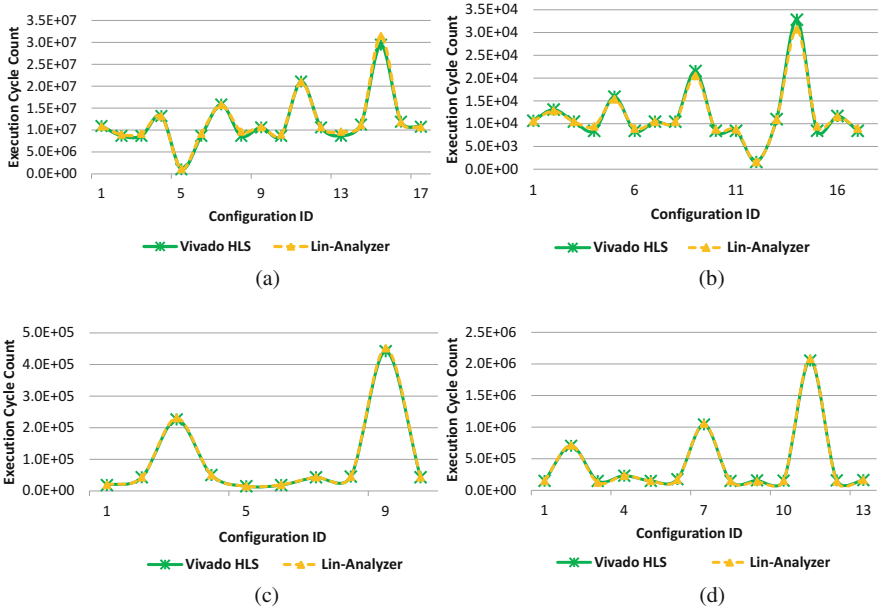


Fig. 2.7 Accuracy of Lin-Analyzer compared to Vivado HLS considering loop unrolling and loop pipelining pragmas. (a) GEMM. (b) KMeans. (c) CONV2D. (d) CONV3D

Table 2.3 Performance comparison with loop unrolling and loop pipelining enabled: Lin-Analyzer vs. Vivado HLS

Benchmark	GEMM	KMeans	CONV2D	CONV3D
Difference (%)	3.25	3.78	1.63	3.75

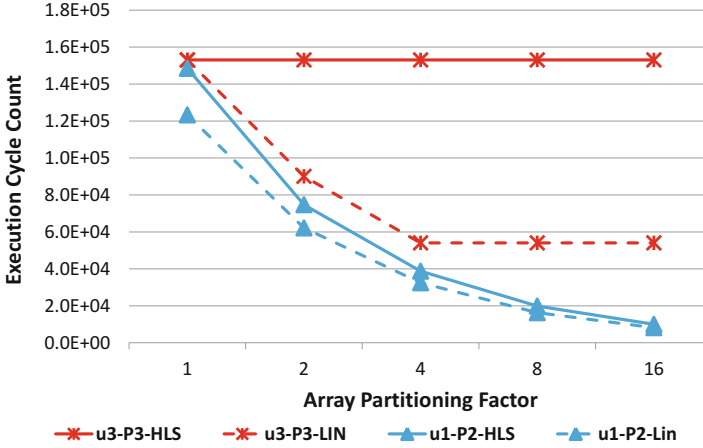


Fig. 2.8 Performance comparison with loop unrolling, loop pipelining, and array partitioning: Lin-Analyzer vs. Vivado HLS for CONV3D. Due to false loop-carried dependences, Vivado HLS generates inefficient designs, which leads to difference with predictions from Lin-Analyzer

ones estimated by Lin-Analyzer is calculated for the same configuration across all combinations and shown in Table 2.3. From the table, Lin-Analyzer can predict performance of FPGA-based accelerators with loop unrolling and loop pipelining enabled within 4.0% difference across all four kernels, which is quite accurate.

2.4.1.2 Array Partitioning

Figure 2.8 demonstrates the result comparison between Lin-Analyzer and Vivado HLS for *CONV3D* kernel with loop unrolling, loop pipelining, and array partitioning enabled. In Fig. 2.8, we fix loop unrolling and loop pipelining configuration and analyze performance when varying array partitioning factors. The Y-axis denotes the execution cycle counts of different configurations, while the X-axis describes different array partitioning factors applied varying from 1 to 16 in step of 2. (*ui-Pj*) represents a configuration combination consisting of loop unrolling factor *i* applied at the innermost loop level and loop pipelining applied at loop level *j*. Solid and dashed lines represent results from Vivado HLS and Lin-Analyzer, respectively.

As a memory bank on FPGAs has limited ports, which potentially hinders HLS tools to exploit more parallelism, array partitioning pragma is designed to split data into multiple memory banks and increase memory bandwidth. This pragma

usually works with loop unrolling or loop pipelining. In Fig. 2.8, performance (the red solid line with stars, *u3-P3-HLS*) from Vivado HLS with configuration *u3-P3* remains constant when applying different array partitioning factors, which means that increasing memory bandwidth has no impact on hardware performance. The results from Lin-Analyzer (the red dashed line with stars, *u3-P3-LIN*) show different behavior compared to Line *u3-P3-HLS*. It demonstrates that increasing memory bandwidth can actually improve performance. The reason that leads to the performance discrepancy of Vivado HLS and Lin-Analyzer can be explained as follows. HLS tools rely on static analysis and perform conservative dependence analysis. It might potentially add *false loop-carried dependences*. In the example above, the *false loop-carried dependences* introduced by Vivado HLS leads to a high recurrence Π (*RecII*) values and the *MII* in Eq. (2.1) is dominated by *RecII*. Therefore, increasing memory bandwidth in this case cannot help to exploit more parallelism. As Lin-Analyzer relies on dynamic trace and all dependences are known, Line *u3-P3-LIN* shows that increasing memory bandwidth can help to reduce execution cycles of accelerators. A hand-written RTL code or enabling dependence pragma to disable specific loop-carried dependence in HLS tools can effectively improve hardware performance as predicted by Lin-Analyzer in Line *u3-P3-LIN*. In addition, by simulating RTL codes generated by Vivado HLS, we find that for some configurations with array partitioning enabled, there exist redundant memory loads. This further deteriorates the hardware performance due to the memory inefficiency in Vivado HLS designs when compared to optimized hand-written RTL implementations.

Although results from the two might be different, Lin-Analyzer can accurately predict the hardware performance trends with array partitioning enabled. Moreover, Lin-Analyzer also can help designers to better understand design bottlenecks and generate high-quality FPGA-based accelerators with HLS tools.

2.4.2 Rapid Design Space Exploration

As mentioned in Sect. 2.3, given various pragma combinations consisting of loop unrolling, loop pipelining, and array partitioning, Lin-Analyzer can rapidly evaluate hardware performance accordingly and enable design space exploration to find the high-quality design point without generating RTL implementations. The design space we consider is shown below,

- *Loop unrolling factor*: Its range includes divisors of loop bound N .
- *Loop pipelining*: Its range includes *True* and *False*.
- *Array partitioning*: The factor can vary from 1 to 16 in steps of 2. The partitioning types are *cyclic*, *block*, and *complete*.

Table 2.4 demonstrates the design space exploration results with exhaustive HLS-based method and Lin-Analyzer. Kernels considered in this work are listed in Column 1. Number of loop levels and design space of each kernel are shown

Table 2.4 Design space exploration results

Benchmark	Loop levels	Design space	Configuration		Total DSE Time (s)			
			Exhaustive	Lin-Analyzer	Exhaustive	Lin-Analyzer		
						Profiling	DSE	Total
GEMM	3	85	1,1,2	1,1,2	36579.48	176.38	8.99	185.37
KMeans	2	136	(8,12,16),1,1	(8,12,16),1,1	3922.33	1.26	45.47	46.73
CONV2D	2	62	16,1,1	16,1,1	26573.13	5.48	17.28	22.76
CONV3D	3	65	16,1,2	16,1,2	21586.68	12.85	9.62	22.47

Configuration format is (*array partitioning factor, loop unrolling factor, pipeline level*)

in Column 2 and 3, respectively. To evaluate accuracy of DSE with Lin-Analyzer, we leverage Vivado HLS to perform exhaustive DSE with the same design space (shown as *Exhaustive* in Table 2.4) and record HLS exploration time and execution cycles of the generated accelerator implementations.

The optimal design points of each kernel given by *Exhaustive* DSE and Lin-Analyzer are shown in Column 4 and 5 in Table 2.4. The configuration format used here is (array partitioning factor i , loop unrolling factor j , pipeline level k). The pipeline level k means that a pipelining pragma is applied at loop level L_k . Column 4 and 5 in Table 2.4 demonstrate that the configurations, which achieves the best performance for each kernel within the design space given, recommended by *Exhaustive* method and Lin-Analyzer are exactly the same.

The exploration time of *Exhaustive* method is shown in Column 6 in Table 2.4. As Lin-Analyzer relies on dynamic trace, its exploration time consists of two parts: *Profiling* and *DSE*. The *Profiling* part is the time spent on collecting dynamic trace, which is a one-time overhead and can be amortized. The total exploration time of DSE with Lin-Analyzer is shown in Column 9 in Table 2.4. Comparing Column 9 with 6, we can see that the exploration time needed by Lin-Analyzer is only a fraction of the time using *Exhaustive* method while recommending the correct configuration combinations. Exploration time speedup of each kernel normalized to *Exhaustive* method is shown in Fig. 2.9. The results in Fig. 2.9 confirm that Lin-Analyzer is capable to perform rapid architectural exploration and the average speedup achieves 617X for the four kernels.

To evaluate the quality of the best design points (within the design space considered) given by our automated DSE flow, we compare their execution time of FPGA implementations with CPU-based performance for all the kernels. CPU-based performance is obtained by running single-thread C implementations of the same kernels on one Intel Xeon CPU E5-2620 at 2.1 GHz and one ARM Cortex-A15 core (from Odroid-XU3 [27]) at 2.0 GHz. We utilize ‘-O3’ as the GCC optimization option. Besides, we also run OpenCL implementations for the four kernels using 4 Cortex-A15 from Odroid-XU3 [27]. The OpenCL implementations are obtained from Polybench Benchmark Suite GPU version [30] and Rodinia Benchmark Suite [6].

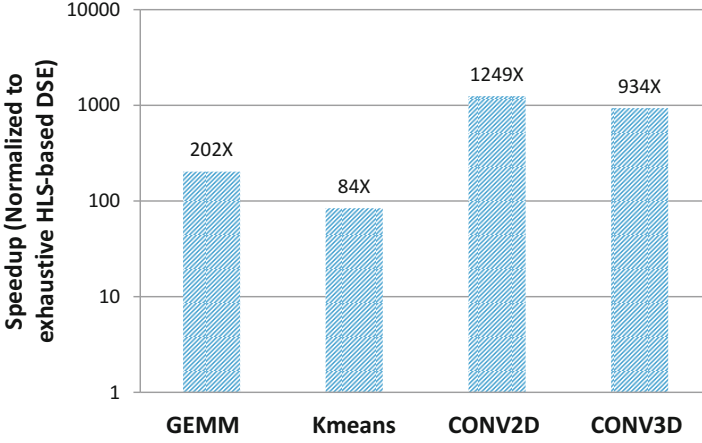


Fig. 2.9 Exploration time speedup compared to exhaustive HLS-based DSE

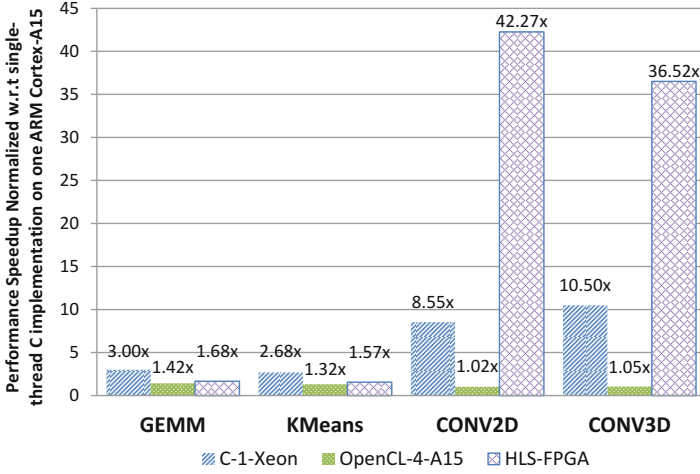


Fig. 2.10 Speedup of different implementations normalized to single-thread C design on one ARM Cortex-A15 core. FPGA implementations, *HLS-FPGA*, leverage the best design points returned by the proposed automated DSE framework. *C-1-Xeon* denotes single-thread C implementation on one Intel Xeon core, while *OpenCL-4-A15* represents OpenCL implementations using four ARM Cortex-A15 cores

The results are shown in Fig. 2.10. In the figure, we use *C-1-Xeon*, *C-1-A15*, *OpenCL-4-A15*, and *HLS-FPGA* to denote the corresponding implementations. For *GEMM* and *KMeans*, performance of *HLS-FPGA* can achieve around 1.6x speedup compared to that of *C-1-A15* and better than *OpenCL-4-A15*. Compared to implementations on the high-end CPU, *C-1-Xeon*, performance of *HLS-FPGA* is slightly slower. The reason for *GEMM* is that its *II* is dominated by memory bandwidth. Due to the limited FPGA BRAM resource, we cannot leverage large array

partitioning factors to increase memory bandwidth. For *KMeans*, as it can be easily vectorized and has less computation operations compared to other kernels, CPU and OpenCL implementations can be well optimized to match FPGA performance. *CONV2D* and *CONV3D* have extensive memory reads and computations and there is no dependence among different output data. However, they are memory-bound kernels on the CPU, as the ratio between their arithmetic operations and memory accesses is low [5]. In Fig. 2.10, due to the memory-bound problem, *OpenCL-4-A15* can not achieve good speedup compared to *C-1-A15*. However, *HLS-FPGA* achieves around 40x speedup compared to *C-1-A15* implementations and roughly 5x speedup compared with *C-1-Xeon*. The reason is that with array partitioning enabled, Vivado HLS can exploit more instruction-level parallelism and utilize deep pipelining. Thus, their FPGA implementations can instantiate lots of functional units for computation and occupy up to 92% DSP resource. This demonstrates that the design points returned by our automated DSE framework have high quality.

As Lin-Analyzer does not rely on HLS tools or generate any RTL implementations, its runtime scales linearly with more complex configuration combinations (larger unrolling and partitioning factors, pipelining at higher loop levels, etc.). This makes Lin-Analyzer be an attractive complementary tool for HLS to perform design space exploration.

2.5 Conclusion

In this chapter, we focus on accelerating data analytics kernels on heterogeneous computing systems featuring FPGAs. In particular, we present a toolchain, called Lin-Analyzer, that allows easy but performance-efficient implementation of data analytics kernels on FPGA-based accelerators. Lin-Analyzer relies on the dynamic data dependence graph (DDDG) to avoid the false data dependences created by the static analysis techniques used in most existing techniques including commercial HLS tools. This results in an accurate performance estimation of FPGA-based accelerators without resorting to time-consuming HLS runs. The tool also helps in identifying design bottlenecks while exploring various pragmas such as loop unrolling, pipelining, and array partitioning. Lastly, Lin-Analyzer can assist HLS developers in identifying potential limitations of the HLS tool. Our experimental evaluation with a number of data analytics kernels confirms the effectiveness of Lin-Analyzer.

Acknowledgements This work was partially supported by the Singapore Ministry of Education Academic Research Fund Tier 2 MOE2015-T2-2-088.

References

1. S. Bilavarn, G. Gogniat, J.L. Philippe, L. Bossuet, Design space pruning through early estimations of area/delay tradeoffs for FPGA implementations. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **25**. Doi:10.1109/TCAD.2005.862742
2. Cadence Inc. C-to-Silicon Compiler (2015)
3. A. Canis, J. Choi, M. Aldham et al., LegUp: high-level synthesis for FPGA-based processor/accelerator systems, in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'2011)*, Monterey (2011)
4. A. Canis, D. Brown, J.H., Anderson, Modulo SDC scheduling with recurrence minimization in high-level synthesis, in *The 24th International Conference on Field Programmable Logic and Applications (FPL)*, Munich (2014)
5. S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.* **68**(10), 1370–1380 (2008)
6. S. Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, L. Wang, K. Skadron, A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads, in *2010 IEEE International Symposium on in Workload Characterization (IISWC)* (2010), pp. 1–11
7. J. Cong, Z. Zhang, An efficient and versatile scheduling algorithm based on SDC formulation, in *The 43rd ACM/IEEE Design Automation Conference (DAC'2006)*, San Francisco (2006)
8. J. Cong, W. Jiang, B. Liu, Y. Zou, Automatic memory partitioning and scheduling for throughput and power optimization, in *IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, San Jose, CA (2009)
9. J. Cong, M. Huang, P. Pan, Y. Wang, P. Zhang, *Source-to-Source Optimization for HLS, FPGAs for Software Programmers*, chap. 8 (Springer International Publishing, Cham, 2016), pp. 137–163. Doi:<http://dx.doi.org/10.1145/2209291.2209302>. ISBN 978-3-319-26408-0
10. W.J. Dally, J.D. Balfour, D. Black-Schaffer, J. Chen, R.C. Harting, V. Parikh, J. Park, D. Sheffield, Efficient embedded computing. *IEEE Comput.* **41**(7), 27–32 (2008)
11. R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, A.R. LeBlanc, Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE J. Solid State Circuits* **9**(5), 256–268 (1974)
12. H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling, in *2011 38th Annual International Symposium on Computer Architecture (ISCA)* (IEEE, New York, 2011), pp. 365–376
13. A.P. Greenhalgh, Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7 (2011)
14. M. Guevara, B. Lubin, B.C. Lee, Navigating heterogeneous processors with market mechanisms, in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)* (IEEE, New York, 2013), pp. 95–106
15. J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.N. Pouchet, A. Rountev, P. Sadayappan, Dynamic trace-based analysis of vectorization potential of applications, in *The 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Beijing (2012)
16. Ineda Systems, Hierarchical computing (2014). [Online]
17. Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: convolutional architecture for fast feature embedding. Preprint (2014). arXiv:1408.5093
18. R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, D.M. Tullsen, Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction, in *MICRO* (2003), pp. 81–92
19. C. Lattner, V. Adve, LLVM: a compilation framework for lifelong program analysis & transformation, in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*, Palo Alto, CA (2004)
20. P. Li, P. Zhang, L.N. Pouchet, J. Cong, Resource-Aware Throughput Optimization for High-Level Synthesis, in *The 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA (2015)

21. Y. Liang, K. Rupnow, Y. Li, D. Min, M.N. Do, D. Chen, High-level synthesis: productivity, performance, and software constraints. *J. Electr. Comput. Eng.* **2012** (2012). Doi:10.1155/2012/649057
22. H. Liu, L.P. Carloni, On learning-based methods for design-space exploration with high-level synthesis, in *The 50th Annual Design Automation Conference (DAC)*, Austin (2013)
23. G.E. Moore, Cramming more components onto integrated circuits. *Proc. IEEE* **86**(1), 82–85 (1998)
24. T.S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, S. Vishin, Hierarchical power management for asymmetric multi-core in dark silicon era, in *Proceedings of the 50th Annual Design Automation Conference* (ACM, New York, 2013), p. 174
25. T.S. Muthukaruppan, A. Pathania, T. Mitra, Price theory based power management for heterogeneous multi-cores, in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and operating systems* (ACM, New York, 2014), pp. 161–176
26. nVidia, Variable SMP—a multi-core CPU architecture for low power and high performance (2011)
27. Odroid-XU3. <http://goo.gl/Nn6z3O>
28. A. Pathania, Q. Jiao, A. Prakash, T. Mitra, Integrated CPU-GPU power management for 3D mobile games,” in *Proceedings of the the 51st Annual Design Automation Conference on Design Automation Conference* (ACM, New York, 2014), pp. 1–6
29. N. Pham, A.K. Singh, A. Kumar, M.M.A. Khin, Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis, in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, San Jose, CA (2015)
30. L. Pouchet, PolyBench/C3.2 (2012)
31. M. Pricopi, T. Mitra, Bahurupi: a polymorphic heterogeneous multi-core architecture. *ACM Trans. Archit. Code Optim.* **8**(4), 22 (2012)
32. M. Pricopi, T. Mitra, Task scheduling on adaptive multi-core. *IEEE Trans. Comput.* **63**(10), 2590–2603 (2014)
33. M. Pricopi, T.S. Muthukaruppan, V. Venkataramani, T. Mitra, S. Vishin, Power-performance modeling on asymmetric multi-cores, in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)* (2013), pp. 1–10
34. A. Prost-Boucle, O. Muller, F. Rousseau, A fast and autonomous HLS methodology for hardware accelerator generation under resource constraints, in *Euromicro Conference on Digital System Design (DSD)*, Los Alamitos, CA (2013)
35. A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides et al., A reconfigurable fabric for accelerating large-scale datacenter services, in *Proceeding of the 41st Annual International Symposium on Computer Architecture* (IEEE, New York, 2014), pp. 13–24
36. J. Redmon, S. Divvala, R. Girshick, A. Farhadi, You only look once: unified, real-time object detection. Preprint (2015). arXiv:1506.02640
37. B.C. Schafer, K. Wakabayashi, Divide and conquer high-level synthesis design space exploration. *ACM Trans. Des. Autom. Electron. Syst.* **17**(3), Article 29 (2012), 19pp. Doi:<http://dx.doi.org/10.1145/2209291.2209302>
38. Y. Shao, B. Reagen, G.Y. Wei, D. Brooks, Aladdin: a pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures, in *The 41st Annual International Symposium on Computer Architecture (ISCA)*, Minneapolis (2014)
39. B. So, M.W. Hall, P.C. Diniz, A compiler approach to fast hardware design space exploration in FPGA-based systems, in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin (2002)
40. Synopsys Inc. (2015)
41. M.A. Todd, S.S. Gurindar, Dynamic dependency analysis of ordinary programs, in *The 19th Annual International Symposium on Computer Architecture*, New York (1992)
42. F.M. Vallina, C. Kohn, P. Joshi, Zynq all programmable SoC Sobel filter implementation using the Vivado HLS tool. Application Note XAPP890, Xilinx (2012)
43. Xilinx Inc. (2015)
44. Z. Zhang, B. Liu, SDC-based modulo scheduling for pipeline synthesis, in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA (2013)

45. G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, S. Niar, Design space exploration of multiple loops on FPGAs using high level synthesis, in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, Seoul (2014)
46. G. Zhong, A. Prakash, Y. Liang, T. Mitra, S. Niar, Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators, in *The 53rd Annual Design Automation Conference (DAC)*, Austin (2016)

Emerging Technology and Architecture for Big-data
Analytics

Chattopadhyay, A.; Chang, C.H.; Yu, H. (Eds.)

2017, XI, 330 p. 162 illus., 98 illus. in color., Hardcover

ISBN: 978-3-319-54839-5