Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2014, Article ID 418750, 17 pages
http://dx.doi.org/10.1155/2014/418750

Research Article

IP-Enabled C/C++ Based High Level Synthesis: A Step towards Better Designer Productivity and Design Performance

Sharad Sinha and Thambipillai Srikanthan

CHiPES, School of Computer Engineering, Nanyang Technological University, Nanyang Avenue, Singapore 639798

Correspondence should be addressed to Sharad Sinha; sharad_sinha@pmail.ntu.edu.sg

Received 16 August 2013; Accepted 5 November 2013; Published 8 January 2014

Academic Editor: Nadia Nedjah

Intellectual property (IP) core based design is an emerging design methodology to deal with increasing chip design complexity. C/C++ based high level synthesis (HLS) is also gaining traction as a design methodology to deal with increasing design complexity. In the work presented here, we present a design methodology that combines these two individual methodologies and is therefore more powerful. We discuss our proposed methodology in the context of supporting efficient hardware synthesis of a class of mathematical functions without altering original C/C++ source code. Additionally, we also discuss and propose methods to integrate legacy IP cores in existing HLS flows. Relying on concepts from the domains of program recognition and optimized low level implementations of such arithmetic functions, the described design methodology is a step towards intelligent synthesis where application characteristics are matched with specific architectural resources and relevant IP cores in a transparent manner for improved area-delay results. The combined methodology is more aware of the target hardware architecture than the conventional HLS flow. Implementation results of certain compute kernels from a commercial tool Vivado-HLS as well as proposed flow are also compared to show that proposed flow gives better results.

1. Introduction

C/C++ based high level synthesis has been gaining momentum to deal with the increasing design complexity. Various academic [1, 2] and commercial tools [3, 4] have been introduced. Similarly, intellectual property (IP) core based design has also been proposed to deal with increasing design complexity. Since IP cores are preverified and optimized for a specific task and in some cases can also be configured to support different compute modes; they ease the task of a designer. IP reuse is another dominant factor in evolving design methodologies to deal with design complexity as well as smaller time to market (TTM) windows. We propose a design methodology in this paper that combines IP based design with HLS. We focus on smaller IP cores which typically implement some arithmetic function. A representative list of standard arithmetic in-built functions available in C/C++ is shown in Table 1. The complete list can be found in [5, 6]. It is the IP cores at this level of granularity that we use in the current work. In Section 5, we discuss extending our approach to larger IP cores.

Traditional HLS flow involves the processes of resource allocation, scheduling, and hardware binding. Every operation in the C/C++ description is treated as an atomic level operation and the bigger operations, like those involving the in-built functions, are replaced by their atomic level descriptions thus resulting in a flattened description of complete design. By atomic level operation, we mean addition, subtraction, multiplication, comparison, and so forth. This methodology completely ignores the fact that hardware design is different from software design. Specifically, it ignores the fact that there can be optimized hardware implementations (IP cores) of those bigger operations. Therefore, it is unable to generate RTL microarchitecture that is closer in spirit to the target hardware; instead, the generated RTL microarchitecture is closer in spirit to the software description. As a specific example, HLS flows assume that transcendental functions like sine(), cosine(), and so forth will be replaced by

Table 1: Representative list of standard arithmetic functions in C/C++.

C/C++ in-built functions type	Available in ⟨math · h⟩ > and ⟨cmath · h⟩
General	div(), ldiv(), lldiv(), abs(), labs(), llabs(), and so forth.
Exponential functions	exp(), exp2(), log(), log10(), log2(), and so forth.
Power functions	sqrt(), cbrt(), hypot(), pow()
Trigonometric functions	sin(), cos(), tan(), acos(), asin(), and so forth.
Hyperbolic functions	sinh(), cosh(), tanh(), asinh(), acosh(), and so forth.
Error and gamma functions	erf(), erfc(), lgamma(), tgamma()

their equivalent polynomial expansions or replaced by some other methods like the *Higher Order Table Based Method* (HOTBM) [7]. In either case, these replacements are still described in C/C++. Equations (1) show the polynomial expansion of sine() and cosine() operators, respectively, for an angle *x radians*. Consider

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \quad \forall x,$$
$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots \quad \forall x. \tag{1}$$

Synthesizing such polynomials can not only be expensive in terms of area but also result in poor timing. The number of terms in these expansions affects the accuracy of the calculations and, with bit growth allowed, such implementations can consume a lot of resources. Also, even when other methods like HOTBM are used, since it is their description in C/C++ that will be synthesized using atomic operations, they are not as efficient as predefined and verified hardware specific IP cores for the same purpose. It would be ideal if such operations are mapped to corresponding hardware IP cores which are typically optimized implementations with considerations of parallelism, throughput, and so forth. Also, the designer can be free from the concern related to the number of terms in the polynomial. Thus, such an IP core can facilitate calculation over a wide range of input arguments without the designer having to worry about the number of terms. For other cases, like linear algebra operations involving matrix computations, FPGA device family specific IP cores are available which have been optimized for these device families [8]. We show in this paper that mapping matrix operations to these IP cores during the HLS phase give better performance results than the traditional HLS flow.

The *main contributions* of the work presented in this paper can be summarized as follows.

(i) To the best of our knowledge, this is the first work that presents a C/C++ based HLS design methodology using IP cores.

(ii) It presents a methodology that relies on existing compilation techniques to quickly identify in-built functions in C/C++ for mapping them to their efficient hardware implementations thus facilitating a more *hardware aware* mapping.

(iii) It presents an automatic graph matching based approach to identify matrix operations in C/C++ design descriptions which can then be mapped to relevant target architecture specific IP cores. It also proposed a new compiler optimization called *Pointer Condense* to deal with program variation in matrix operation code.

(iv) It proposes and discusses an *adaptive compilation strategy* embedded in HLS flow to deal with legacy IP cores as well as proprietary IP cores that companies use in real-world applications.

(v) All of the above is achieved without altering original C/C++ source code thus increasing designer productivity and facilitating an easy C/C++ hardware translation.

2. Background and Related Work

2.1. High Level Synthesis. High level synthesis (HLS) has been a widely researched area. The most useful surveys and tutorials can be found in [9–11]. We cover in brief research in HLS for the sake of completeness. High level synthesis includes three main steps: resource allocation, scheduling, and hardware binding. Resource allocation involves allocating functional units like adder, subtractor, multiplier and so forth and communication resources like buses and multiplexers and storage resources like registers and on-chip memory. Allocation of resources is not an algorithmic step but it refers to the use of the aforementioned resources to perform the operations present in the application as well as to make connections between those resources. It should be noted here that so far the functional units have been atomic level performing some basic operation. Scheduling step involves ordering in time the operations present in an application. The aim of scheduling is to reduce the number of control steps required to compute the output of the application. More than one operation can execute in each control-step. A number of scheduling approaches have been proposed in the research literature with notable ones being As Soon As Possible (ASAP) As Late As Possible (ALAP), and force directed scheduling [12, 13]. Another prominent scheduling algorithm based on system of difference constraints was proposed in [14]. It can be found implemented in Leg-Up [15] and it also forms the basis of the scheduling algorithm implemented in [3]. Hardware binding involves binding the allocated resources to operations present in the application. Binding one resource to each operation is a trivial task. Therefore, techniques have been investigated to bind the resources

to operations with the objective of reducing the number resources used. This would lead to a reduction in hardware area which is favorable in many cases. Therefore, hardware binding algorithms have looked at simultaneous binding of functional units and registers to reduce the area and consequently reduce the number of interconnects. Hardware binding based on weighted bipartite matching (WBM) [16], network flow method [17], compatibility path based (CPB) [18, 19], and so forth, has been proposed. Note that in all these algorithms, the operations in an application are bound to atomic level functional units. Once hardware binding is complete, then register transfer level (RTL) description of the design is automatically generated.

2.2. Intellectual Property (IP) Core Based Design. Intellectual property core based design and design reuse is a widely debated topic in the semiconductor industry though there is not much research on a proper design methodology. While almost everyone agrees that it has the potential to reduce efforts to deal with design complexity and is the way forward, research work published so far is relatively new and has instead largely focused on IP protection techniques using watermarking [20, 21], fingerprinting [22] and on verification techniques for such designs [23]. There has also been work done on creating a repository of IP cores like UMIPS [24]. There have been concerns in the industry regarding different IP description methods adopted by different IP vendors as this inhibits rapid use and a standardized flow for IP core based designs. Therefore, the Institute of Electrical and Electronic Engineers (IEEE) introduced the IP-XACT standard [25] to enable a standard way of describing IP cores thus facilitating their integration with design tools. While the work presented here is focused more on identification of code segments which can be mapped to IP cores during HLS, we do discuss the impact of a standard way of describing IP with respect to selecting one IP core from among many others during the mapping phase after identification.

Previous work on IP based hardware design has also focused on evaluating and selecting an IP for a given requirement. It has also been referred to as component based design. In [26], the authors have investigated system level design space exploration to select different components for different blocks of a system. They modeled digital systems as communication task graphs [27] and formulated the task of IP selection as a bicriteria convex cost flow problem in an incremental design space exploration methodology with online refinement. However, their methodology for IP selection relies on a convex piecewise linear curve for each component. This neglects the fact that an IP core or a component will have features and options which cannot be represented by a number, for instance, the maximum and the minimum bit-width, the internal serial or parallel configuration, rounding mode, and so forth. Simulation based IP core selection has been studied in [28]. This also has the same drawback as in [27] besides being computationally intensive for system with large number of components. Another work [29] looks at augmenting the IP-XACT description of IP cores and then searching the database for SoC design based on requirements similarity. It should be noted here that in all these previous works, it is assumed that the component for which an IP core needs to be selected is known *a priori*. A major contribution of the work presented in this paper is that it does not assume any prior knowledge of components required in a design.

2.3. Program Recognition. Program recognition is a branch of study in computer science that seeks to understand what a program does by analyzing and studying the source code. It relies on *lexical and semantic analysis* of the source code as well as *algorithm recognition.* An excellent reference in this area is the work on automatic algorithm recognition and replacement [30] which focuses on such efforts directed at software optimization. In the work proposed here, we have tried to investigate the same principle but applied to hardware optimization in a high level synthesis flow. Lexical and semantic analysis of the source code can tell us which in-built functions available in a language have been used. Since in-built functions are standard, their interpretation is also constant and does not lead to any *program variation* with respect to their functionality. For instance, the function cos() in C would always return the cosine value of a parameter of data type *double* and the function cosf() in C would always return the cosine value of a parameter of data type *float*. Since the in-built functions implement some algorithm to complete some computational process, for instance calculating the cosine value of some parameter x, the identification of in-built functions can be equated with *algorithm recognition*. Once these functions are identified and the algorithms thus recognized, they can be matched to optimized hardware implementations which achieve the same purpose. It should be noted here that the *actual algorithm* used (in a source language like C) to complete a computational process is not important because the optimized hardware implementation may or may not have used the same algorithm.

In [31], the authors have investigated a new approach to DSP intrinsics for DSP processors. Here, also they have tried to identify specific instructions from a compiler's IR and promote them to C-keywords. Intrinsics were developed because often an engineer would like to use a target specific instruction in code, but there is no way to describe that instruction in C/C++. In its most basic form, an intrinsic is an *asm* statement as found in gcc [32]. This concept is similar to the fact that a HLS user may not be aware of the different optimized IP cores available for different sets of operations in C and that there is no way to describe them in a language like C/C+. In [33], a very recent work, the authors have looked at program recognition in order to parallelize sequential programs. They use a precompiler phase which processes specific keywords in the main source code and replaces chosen sequential algorithms with their parallel implementations selected from a database. They have assumed that most programmers are either unable or unwilling to write parallel implementations of their sequential code. This assumption is valid even for HLS use, the case which the current work addresses. Most HLS users might be comfortable in describing an algorithm in C/C++ but they would not be

aware of corresponding hardware implementations because they are outside their domain expertise. Even for hardware designers, who adopt HLS, the barrier that existing design flows have to exploiting optimized IP cores gives rise to issues with performance optimization. It is only sufficient to mention that the workload of these two classes of designers can be significantly reduced if knowledge about IP cores is somehow embedded in the compilation phase instead of them having to adopt some ad hoc way of identifying candidates for IP core implementation in the source code.

It has also been mentioned in [34] that program and algorithm recognition can play an important role in hardware-software codesign.

2.4. Compiler-Specific Pattern Based Synthesis. Pattern based high level synthesis has been investigated in [35] where the authors propose an algorithm to find recurring patterns in an application through subgraph enumeration, pruning and matching techniques in compute intensive applications. These patterns are then mapped to FPGAs. In [36], the authors have proposed a scheduling methodology based on behavioral templates which are essentially subgraphs. Here also, the templates are created by enumerating the CDFG of an application. Research in [37–41] and so forth is all focused on different methods to identify potential custom instructions based on graph processing techniques applied to the CDFG of an application. Thus, this class of work is focused on finding *often repeated* patterns in application where the patterns are either generated by enumeration or taken from a library of available patterns and finally subgraph matching is performed. However, the patterns used are small with maximum size being 40 nodes in [35]. Subgraph matching has its own drawback as it requires the patterns in the library to be very precise. Also, the compilation step may generate a pattern which would be *functionally identical* to a pattern in a library but *structurally dissimilar* thus making it impossible to match. Also, since different compilers may generate the intermediate representation (IR) in different ways, patterns in a predefined and *compiler-unaware* library may have a reduced significance; as such patterns may not appear in the IR. The problem with enumeration based pattern finding and matching is that even in this case the design flow has to rely on a predefined set of rules to enumerate patterns during the enumeration phase. This can lead to a large number of patterns which may not necessarily yield any advantage. Also, it is extremely difficult to construct a set of rules to describe a pattern that matches any IP core as the number of rules is likely to be large given the different options to configure and describe an IP core. The work closest to using the concept of *compiler-aware* library is [31] where the authors have looked at writing DSP intrinsics to the compiler's IR and promoting them as C-keywords for code targeted to a DSP processor.

Since in any HLS flow, there will be a compilation step involved, in the current work, we propose a *compiler-specific* pattern finding and matching approach. Thus, in the real world, any HLS vendor will only need to define a set of patterns which is specific to the underlying compilation process. We demonstrate this approach for linear algebra operations involving matrix operations under the LLVM compiler [42] framework. This approach distinguishes the current work from others.

3. Proposed IP-Enabled C/C++ HLS Design Flow

We propose a design flow where candidate code segments for mapping to IP cores are selected at two different levels of analysis. The first level involves *algorithm recognition* and here lexical and semantic analysis of the source code and the symbol table [43] used by the compiler come into action. The second level involves *pattern-matching* at the IR level. The complete design flow is shown in Figure 1.

The proposed design flow is based on LLVM [42] compiler framework. Control and dataflow graph (CDFG) forms the IR of an application.

3.1. Lexical and Semantic Analysis for In-Built Functions. In this step, the symbol table generated as a result of lexical and semantic analyses phases of the compilation process is analyzed to locate calls to those in-built functions in C/C++ which are listed in Table 1. These function calls also show up in the IR. The lexical analysis phase parses the source code to generate identifiers like variable names, operation symbols, reserved words (like in-built functions), and so forth. The semantic analysis phase analyses the context in which an identifier is used, for instance, as a variable or a procedure, and so forth. These help in generating the symbol table where each identifier is stored as a symbol along with its usage as a variable, procedure and so forth, its data type, number of and list of parameters if used as a procedure, return data type, and so forth. Thus, at this step, there is complete information about the different in-built functions that are used in a given C/C++ application.

3.2. Identification of Linear Algebra Operations. We identify the linear algebra operations involving matrix operations by processing the different basic blocks (BB) in the IR for nested loop identification and thereafter subgraph matching as shown in the design flow. Matrix operations are chosen because they are found in a number of applications. Each basic block is a sequence of operations and has only one entry and exit point. The algorithm to identify the matrix operations is given in Section 3.3.

3.2.1. Structure of Linear Algebra Operations in LLVM IR. We solve the matrix operation identification problem for matrix multiplication of different kinds as well as matrix addition and subtraction. Matrix multiplication can be classified into (a) matrix-matrix multiplication, (b) matrix-vector multiplication, and (c) matrix scaling where a matrix is multiplied by a constant. Vector-vector multiplication is a special case of matrix-matrix multiplication.

Algorithm 1 shows the *for-loop* code for matrix-matrix multiplication in C for 2×2 matrices. Algorithm 2 shows the code for matrix-vector multiplication. Algorithm 3 shows the code for matrix scaling. For a 2×2 matrix-matrix addition,

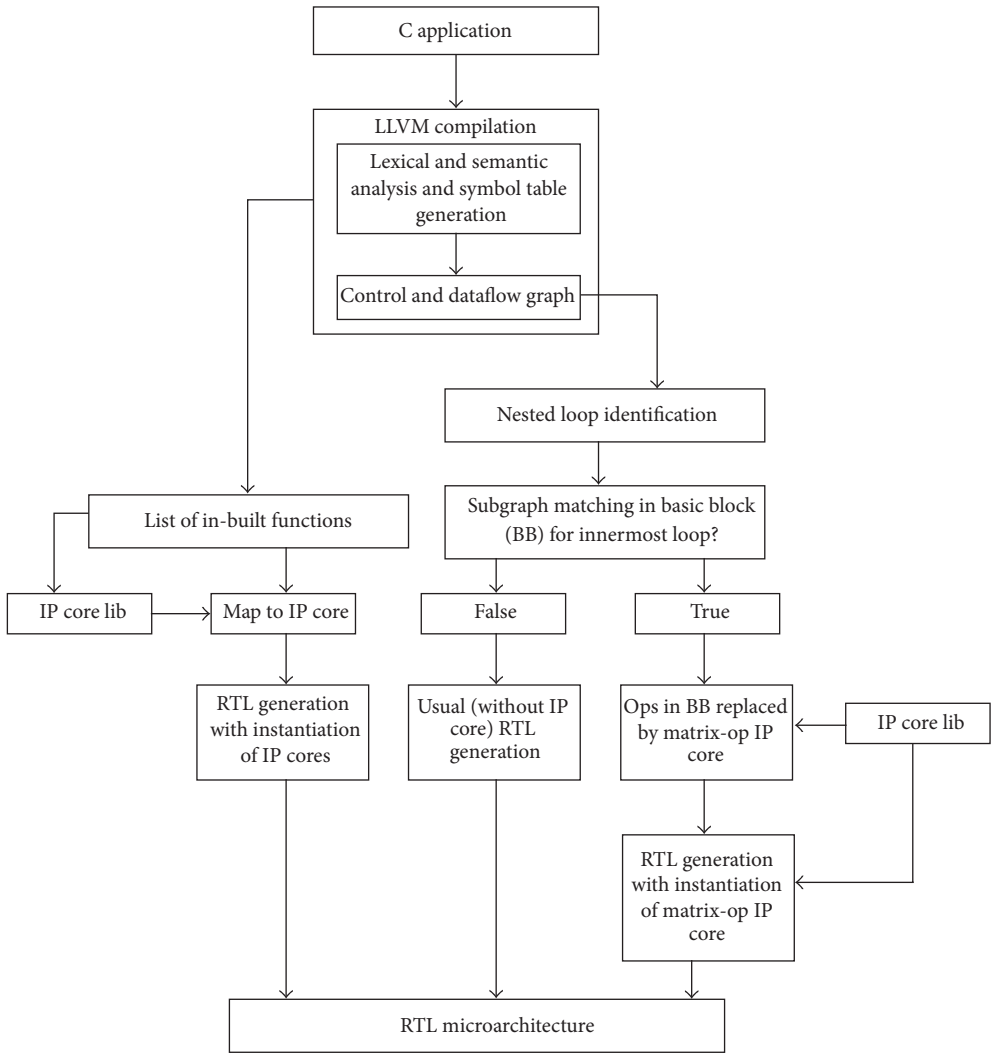

Figure 1: Proposed IP-enabled C/C++ HLS flow.

```
for (i = 0; i < 2; i + +) {
   for (j = 0; j < 2; j + +) {
     for (k = 0; k < 2; k + +) {
           prod[i][j] = prod[i][j] + mat1[i][k] ∗ mat2[k][j]; }
         }
}
```

Algorithm 1: For-loop code for 2 × 2 matrix-matrix multiplication in C.

```
for (i = 0; i < 2; i + +) {
   for (k = 0; k < 2; j + +) {
      prod[i][0] = prod[i][0] + A[i][k] ∗B[k][0];
   }   }
```

Algorithm 2: For-loop code for a matrix-vector multiplication.

```
for (i = 0; i < 2; i + +) {
    for (k = 0; k < 2; j + +) {
        A[i][k] = A[i][k] ∗ scalefactor;
    }   }
```

Algorithm 3: For-loop code for scaling a 2 × 2 matrix.

```
for (i = 0; i < 2; i + +) {
    for (k = 0; k < 2; j + +) {
        sum[i][j] = A[i][j] + B[i][j];
    }
}
```

Algorithm 4: For-loop code for 2 × 2 matrix-matrix addition in C.

the *for-loop* is shown in Algorithm 4. It is the same for matrix-matrix subtraction with addition operator replaced by subtraction operator. Figures 2, 3, 4, and 5 show the loop body IR as a directed acyclic graph (DAG) for the code segments in Algorithms 1, 2, 3, and 4, respectively. It can be seen that the actual multiplication or addition or subtraction code block is always in the innermost *for-loop* of a nested block of *for*-loops.

Loop Identification. The first step in identifying matrix multiplication operations in the IR is to identify the loops. Loop identification is a well-established technique in compiler design [43]. The dominator tree analysis is used for this purpose. For the different kinds of matrix multiplication (a), (b), or (c) mentioned in the beginning of this section, there are different levels of dominating loops with the inner most loop being most dominated. We use "*Lx*" where *x* is a number to indicate the level of domination of a loop with a higher value of *x* meaning a higher domination of the said loop by other loops. Thus, *L*3 (*loop k*) is the most dominated loop while *L*1 (*loop i*) is the least and *L*1 dominates both *L*2 (*loop j*) and *L*3 as in Algorithm 1.

Matrix Size Calculation. The size of the matrices being multiplied and of the vector in case of matrix-vector multiplication is determined from the loop iteration limits. Therefore, if the iteration limit for loops *L1*, *L2* and *L3,* is *I1*, *I2,* and *I3,* respectively, then the matrix sizes, that is, total number of elements and the number of rows and the number of columns, are given in Table 2.

In the case of matrix multiplied by a vector or a vector multiplied by a matrix, there are only two loops *L1* and *L2* with iteration limits *I1* and *I2,* respectively. *L2* is considered the innermost loop. In case of a matrix multiplied by a vector, the sizes are given in Table 3. In case of a vector multiplied by a matrix, the sizes are given in Tables 4 and 5 which show the sizes for matrix scaling.

In case of matrix-matrix addition or subtraction, there are only two loops *L1* and *L2* with iteration limits *I1* and *I2,*

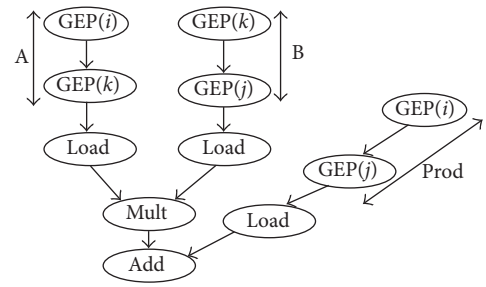

Figure 2: Directed acyclic graph (DAG) of IR for matrix-matrix multiplication.

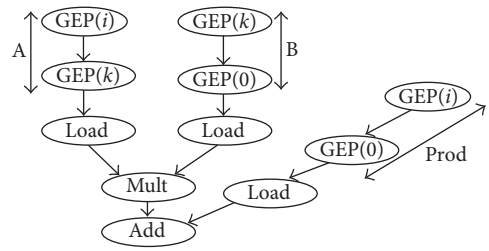

Figure 3: Directed acyclic graph (DAG) of IR for matrix-vector multiplication.

respectively. *L2* is considered the innermost loop. The sizes are determined as follows:

$$\mathrm{S}(\text{matrix1}) = \mathrm{S}(\text{matrix2}) = \mathrm{S}(\text{sum matrix}) = I1 \times I2. \quad (2)$$

GEP() stands for the "getelementpointer inbounds" instruction in LLVM. It is a memory instruction that fetches the address of the array locations which store the matrix elements.

3.3. Algorithm for Identification of Matrix Operations. Since the size of matrices can be determined as shown in Tables 2, 3, 4, and 5, the relevant IP cores can be configured automatically in the design flow. We discuss this in more detail in Section 3.4 and Algorithm 5.

Table 2: Matrix size determination for matrix-matrix multiplication.

	Left matrix	Left matrix, rows	Left matrix, cols	Right matrix	Right matrix, rows	Right matrix, cols	Product matrix	Product matrix, rows	Product matrix, cols
Size	I1 ∗ I3	I1	I3	I3 ∗ I2	I3	I2	I1 ∗ I2	I1	I2

Table 3: Matrix size determination for matrix-vector multiplication.

	Matrix	Matrix, rows	Matrix, cols	Vector	Vector, rows	Product matrix	Product matrix, rows
Size	I1 ∗ I2	I1	I2	I2 ∗ 1	I2	I1 ∗ 1	I1

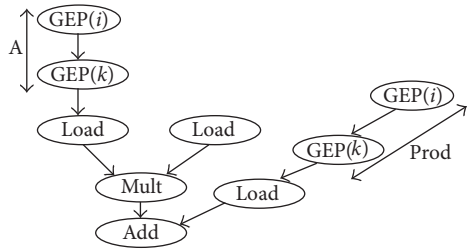

Figure 4: Directed acyclic graph (DAG) of IR for matrix scaling.

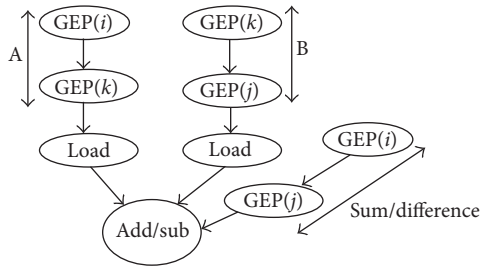

Figure 5: Directed acyclic graph (DAG) of IR for matrix-matrix addition/subtraction.

3.3.1. Dealing with Program Variation. The directed acyclic graph representation (DAG) of IR shown in Figures 2, 3, 4, and 5 represents the C/C++ code as shown in Algorithms 1, 2, 3, and 4, respectively. A key issue to address here is the structure of the DAGs when code is not written as demonstrated in these figures or when additional statements are inserted in the loop body. These additional statements could be unrelated to matrix operations or the matrix operation could be executed on more than one matrix or they could be some form of manual optimization, for instance, more than one matrix multiplication in the same loop body. The code segments shown in Algorithms 1, 2, 3, and 4 are the easiest way of coding the relevant matrix operation from a software perspective. Thus, when one is building an executable specification of a design in C/C++ and using HLS to map it to hardware (FPGA or ASIC), it is not necessary to write an optimized C/C++ implementation because the target of such a design description is not a processor but mapping to hardware. The executable nature of the specification is needed only to verify the hardware mapping against execution results from the C/C++ description to ensure functional correctness. The matrix operation DAGs as shown earlier will always be present in the loop body DAG irrespective of any other code in the loop body. Thus, the matrix operation DAGs can always be located in the innermost loop body IR.

Loop Fission. Traditionally loop fission optimization can be applied to cases of more than one matrix multiplication (or any other matrix operation) being done in the same loop body, assuming that the relevant matrices are all of similar sizes. This would replicate the loop constructs while putting each matrix operation in a different loop construct. However, we do not adopt this approach because in our graph matching based approach more than one instance of matrix multiplication (or any other matrix operation) DAG will be easily found. Similarly, if the loop body code has a mix of different matrix operations, instances of each DAG type can be found.

Different Loop Constructs. Loops can be constructed using *do-while* and *while* constructs also in C/C++. For each of these constructs, the loop body IR is different from the loop body IR for *for*-loops. However, the matrix operation DAGs are still present in the innermost loop body IRs in the same form as shown in Figures 2, 3, 4, and 5. Thus, it does not matter which loop construct is used to form the loops.

Loop Rerolling. If the loop body is partially unrolled in the code, we apply the *loop-rerolling* [44] optimization. This is because loop unrolling obscures the underlying computation algorithm (for example matrix multiplication) and as such DAG matching will fail. Loop rerolling cannot be applied to completely unrolled loops because there is no loop in the code in such cases. The unrolling is visible to and understood by only the designer.

Data Structure Variation. The C codes given for different matrix operations make use of arrays whose elements are accessed in the usual way of accessing array elements. However, there are two main variations in data structure: (1) the array elements can be accessed using pointers even when declared as arrays and (2) the array elements are accessed using pointers when the two-dimensional matrices' arrays are declared using pointers. Typically, pointer based operations and matrices will be required only for code targeting a processor and involving large matrices. For HLS, one can safely stick to the usual access methods irrespective of the size of matrices. Nevertheless, we have addressed this issue for the sake of accommodating different programming styles by

Table 4: Matrix size determination for vector-matrix multiplication.

	Vector	Vector, cols	Matrix	Matrix, rows	Matrix, cols	Product matrix	Product matrix, cols
Size	$1 * I2$	I2	$I2 * I1$	I2	I1	$1 * I1$	I1

Table 5: Matrix size determination for matrix scaling.

	Matrix	Matrix, rows	Matrix, cols	Product matrix	Product matrix, rows	Product matrix, cols
Size	$I2 * I1$	I2	I1	$I2 * I1$	I2	I1

introducing a new compiler optimization pass specifically for HLS. Equation (3) shows three equivalent ways of accessing a memory element $A[i][j]$:

$$A[i][j] \equiv *(A[i] + j) \equiv *(*(A + i) + j). \quad (3)$$

These three different ways of accessing an array element lead to different numbers of GEP() instruction in LLVM IR. $A[i][j]$ involves two GEP() while $*(A[i] + j)$ requires three GEP() and $*(*(A+i)+j)$ needs four GEP() instructions. The easiest way to accommodate these program variations would be to define three different DAGs corresponding to each type of memory access. However, that would add to the DAG database as well as lead to additional time in the execution of DAG matching. The additional increase in the size of the DAGs would also add to the execution time. Therefore, to deal with this program variation, we have developed a compiler optimization pass called *Pointer Condense*. This pass converts both $*(A[i]+j)$ and $*(*(A+i)+j)$ to $A[i][j]$ during semantic analysis phase applied to the syntax tree using the symbol table when A is an array.

3.4. Intellectual Property (IP) Core View. The IP core view is a view of the IP that provides information on its configurable as well as static attributes. The attributes are separated into *computation* and *noncomputation* attributes. The computation attributes are those which are essential and necessary for computation operations within the IP core to take place. They include available input-output port bit-widths, supported rounding modes, and any internal precision attribute. The noncomputation attributes are those which are not essential and necessary for computation operations within the IP core to take place. They include area (in terms of LUT when FPGA is the target architecture), maximum clock frequency, serial and parallel interfaces, latency, and throughput. The list of computation and noncomputation attributes is determined by the commonly used parameters for IP cores targeted in the work presented here. Attributes like input-output port bit-widths, supported rounding modes, serial and parallel interfaces, use of LUT or DSP blocks for multiplication, and so forth, can be configurable depending upon the nature of an IP core. We assume that these attributes have already been extracted from the IP-XACT [25] description of the IP cores. We use these attributes to arrange the IP cores in an IP core library. Since timing is critical and is the most important in any design, the IP cores are arranged based on their maximum clock frequencies. IP cores which have no configurable attributes will have only one maximum frequency because such cores cannot be configured to yield a different design internally. Only those IP cores which have at least one configurable attribute may have more than one maximum clock frequency. We use the maximum clock frequency as the head of a linked-list that stores the IP attributes. For any function f which is a candidate for IP core implementation, the different IP cores can be described as shown in Table 6. Note that the number of available attributes for different IP cores needs not be same. Some IP cores may have architecture attributes like serial or parallel. These are different from interface attributes. We do not show all the attributes in Table 6 for brevity. Also, the IP core selection algorithm, as discussed in Section 3.5, does not use any attribute information besides maximum clock frequency, rounding mode, and input and output bit-widths.

As can be seen in Table 6, there are a total of 4 different IP cores that can perform function *f*. IP Core2 has 2 configurations and each can support two different rounding modes RND(M1) and RND(M2) for a total of four configurations. The input and the output port bit-widths are different for these two configurations and so is the interface (serial versus parallel). The only design constraints that are required to walk on the linked-lists are the required clock frequency, T_{req}, and the required rounding mode, RND (req).

Description like in Table 6 is maintained for each FPGA device architecture in the IP library and therefore the flow takes into account whether a particular function f has support for IP core in a target FPGA device or not.

3.5. IP Core Selection. The IP core selection space is pruned in a hierarchical way by walking over their linked list descriptions. First stage of pruning involves checking for the truth of the inequality given in (4) below, where T_i is the maximum clock frequency of an IP core configuration

$$T_i \geq T_{\text{req}}. \quad (4)$$

The second stage of pruning involves checking for the truth of the equality given in

$$\text{RND}(M_i) = \text{RND}(\text{req}). \quad (5)$$

These are applied successively on each linked-list. Those configurations that satisfy (4) and (5) are further examined for the cost metric called *Difference of Attributes* (DoA) as given in

$$\text{DoA}(n_i, k) = \sum_{j=1}^{t} \frac{1}{j} \left[AV_k(j, 0) - AV_{n_i}(j, 0) \right]. \quad (6)$$

Table 6: A function f and corresponding IP cores.

Function	IP cores
f	$T_1 \rightarrow \text{RND}(M_1) \rightarrow$ InputBW1 $\rightarrow$ OutputBW1 $\rightarrow$ None $\rightarrow$ Serial Arch $\rightarrow$ Area1 $\rightarrow$ IPCore1
f	$T_2 \rightarrow \text{RND}(M_1) \rightarrow$ InputBW2 $\rightarrow$ OutputBW2 $\rightarrow$ Serial *I/F* $\rightarrow$ Area2 $\rightarrow$ IPCore2
f	$T_3 \rightarrow \text{RND}(M_2) \rightarrow$ InputBW2 $\rightarrow$ OutputBW2 $\rightarrow$ Serial *I/F* $\rightarrow$ Area2 $\rightarrow$ IPCore2
f	$T_4 \rightarrow \text{RND}(M_2) \rightarrow$ InputBW3 $\rightarrow$ OutputBW3 $\rightarrow$ Parallel *I/F* $\rightarrow$ Area3 $\rightarrow$ IPCore2
f	$T_5 \rightarrow \text{RND}(M_1) \rightarrow$ InputBW3 $\rightarrow$ OutputBW3 $\rightarrow$ Parallel *I/F* $\rightarrow$ Area3 $\rightarrow$ IPCore2
f	$T_6 \rightarrow \text{RND}(M_1) \rightarrow$ InputBW4 $\rightarrow$ OutputBW4 $\rightarrow$ None $\rightarrow$ Area4 $\rightarrow$ IPCore3
f	$T_7 \rightarrow \text{RND}(M_1) \rightarrow$ InputBW5 $\rightarrow$ OutputBW5 $\rightarrow$ None $\rightarrow$ Parallel Arch $\rightarrow$ Area5 $\rightarrow$ IPCore4

Here, n_i refers to the node (or set of nodes) which is being mapped to an IP core and k is the kth IP core that can implement the function in that node (or set of nodes). AV_k and AV_{n_i} are the attribute vectors that store the input port bit-widths and the output port bit-widths. The scaling factor of $1/j$ gives more weight to input port bit-widths because if input port sizes do not match then the IP configuration is unusable. While output bit-width is also important, we give it less importance because a full precision calculation may not always be required. Hence, even if output bit-width is not to full precision, the input bit-width is still important so as to not lose precision on the input. The upper summation limit t is the sum of the numbers of input and output ports. We restrict the results in this paper to two input and one output ports. The IP core configuration that yields the minimum DoA value is selected for mapping. This would lead to one design point corresponding to the selected IP core configuration. However, we would like to point out here that this very approach can be used for a localized, IP core level design space exploration. As long as DoA > 0, the corresponding IP core configuration can be used in the design.

Area Reduction by IP Core Resolution. Let there be operations *f1* and *f2* to be computed in the same control step in the schedule. Let these operations belong to Table 1 so that they are candidates for mapping to IP cores. It is possible that some IP cores can provide more than one output at the same time on the same set of inputs. For instance, there could be an IP core that can compute both sine and cosine simultaneously for a given input. An example of such an IP core is the CORDIC IP core provided by Xilinx [45]. In an application, if *f1* and *f2* are two such operations which are scheduled in the same control step and they share the same input, then they could be mapped to an IP core which computes both simultaneously. Therefore, once the code candidates for mapping to IP cores have been identified (Sections 3.2, 3.3), we implement *IP Core Dependency Resolution Algorithm* (Algorithm 6).

The reason for doing this is that it can lead to reduced area. Instead of using two different IP cores or two instances of the same IP core (because it can implement both functions *f1* and *f2*), only one IP core can be used since the operations *f1* and *f2* are scheduled in the same control step and have the same inputs.

3.6. Interface Synthesis. Since we slice away a part of the IR for implementation using IP cores from a library while the remaining IR is implemented using the traditional atomic level hardware binding method, it is important to account for interface synthesis for the point where the IR was sliced away and the point where the IP core output joins the IR. The interface is characterized by *data interface* and *control interface*. Data interface describes the input-output port sizes, data types (signed, unsigned, single precision, double precision, etc.) and whether the input-output ports are serial, parallel, or some other configuration. Control interface describes the control signals necessary for input and output of data to and from the IP core.

Once an IP core is selected as described in Section 3.5, interface synthesis takes place. If the operation identified for mapping to IP core is the *only* operation in the design, then different microarchitectures corresponding to the different interfaces supported by the selected IP core can be generated leading to an area-delay tradeoff curve. For instance, if a *matrix operation* is the only operation in the C/C++ description of the application and if the target device is a Xilinx Virtex-6, Virtex-7, or Kintex-7 FPGA, then the Xilinx linear algebra IP core supports both serial and parallel input-output for data. Thus, the design flow returns *different microarchitectures* corresponding to each interface provided the relations in (4) and (5) (Section 3.5) are satisfied. Additionally, if there exist additional IP core configuration attributes like serial or parallel architecture, the design flow returns even more microarchitectures provided again (4) and (5) are satisfied. As can be seen in Table 6, each possible configuration of an IP core that can implement a function f is stored as a linked-list in the IP library. Thus, the design flow returns as many microarchitectures as successful (satisfying (4) and (5)) configurations of one IP core or more.

On the other hand, if the application has other operations besides those listed in Table 1, then the interface synthesis involves understanding the control and data flow at the points where the CDFG was sliced for the sliced portion to be implemented in an IP core. Therefore, in this case, a further pruning of IP cores selected in Section 3.5 is done. The input data to the IP core can come from the previous computation nodes in the CDFG or they can come from some memory block. In either case, the latency and the throughput attributes of the IP core are taken into account. Note that latency and throughput are functions of the IP core's internal

Input: CDFG of Application in C/C++
Output: CDFG with cluster of operations (sub-graph) identified as matrix operations
(1) **For** each Basic Block
(2) Perform ***Dominator Analysis*** to identify nested loops
(3) **For** each set of nested loops
(4) Label loops using "*Lx*" as defined in Section 3.2.1.
(5) Select nodes in CDFG bound by loop with highest *x* value
(6) Perform *subgraph-matching* for subgraphs shown in Figures 2, 3, 4 and 5
(7) **If *true*** for Figure 2
(8) Mark nested loop set as matrix-matrix multiplication
(9) **If *true*** for Figure 3
(10) Mark nested loop set as matrix-vector multiplication
(11) **If *true*** for Figure 4
(12) Mark nested loop set as matrix scaling
(13) **If *true*** for Figure 5
(14) Mark nested loop set as matrix add/subtract
(15) **EndFor**
(16) **EndFor**

Algorithm 5: Identification of matrix operations in LLVM IR.

Input: List of Candidate functions for mapping to IP cores, their C-step number, their operands
Output: Candidate functions to be mapped to same IP core in the same C-step
(1) **For** each C-step number in the list
(2) **Check** for candidate functions with same input operands
(3) **If***same input* operands
(4) **Check** for IP core selected
(5) **If** same IP core selected
(6) **Check** if IP core is concurrent multi-output
(7) **If** *true*, use only one instance of the IP core
(8) **Else** use more than one instance of the IP core
(9) **EndIf**
(10) **EndIf**
(11) **EndIf**
(12) **EndFor**

Algorithm 6: IP core dependency resolution algorithm.

architecture which may be a configurable attribute (serial or parallel architecture). Typically, all IP cores will have an elementary set of control signals to signal to the sender and the receiver modules the readiness to receive new set of input data from the sender and the readiness to send a new set of valid output data to the receiver. This information related to the flow control mechanism of the selected IP core is used in the design flow to design the state machine for the datapath. A part of the state machine relevant to the IP core section of the design is shown in Figure 6.

In Figure 6, S(p) refers to the state machine state prior to IP core computation phase and S(l) refers to the state later than IP core computation phase. The format *a/b* stands for *input signal a to state* and *state output signal b* when *input signal a* is present. Ready for data (RFD) is a control signal in which the IP core asserts when it is ready to receive a new set of input data. ND stands for new data which the previous state asserts while sending *data in* to the IP core.

As can be seen that there is only output control signal RDY (stands for ready with valid output) from the state in which IP core computation is performed. This signal is automatically asserted by the IP core after it has calculated the valid output data. The state machine in Figure 6 transitions from one state to another, that is, state S(p) to state S(c) to state S(l) based on the different control flow signals. Any additional flow control signals that the IP core has can be easily integrated in the three states.

If the data to the IP core comes from memory blocks, then choosing between (or if possible both) serial and parallel data interface is a matter of how the memory blocks are arranged. The sequence of instructions GEP > GEP > LOAD as present in the DAGs shown in Figures 2, 3, 4, and 5 are actually memory read operations. In a typical HLS flow, the matrices will be stored in arrays and mapped to a memory, for example, block RAM (BRAM) on Xilinx FPGAs. Thus, this sequence of instructions would translate into read from these

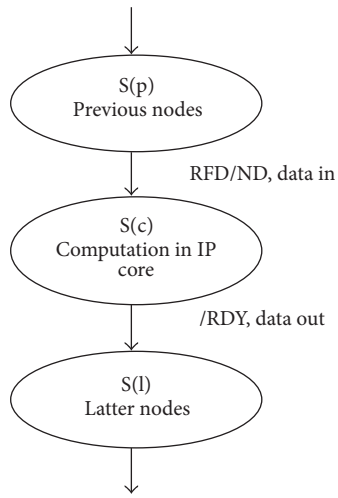

Figure 6: A part of the state machine for datapath in the presence of an IP core.

BRAMs. For the case of different kinds of matrix operations, the depth of the BRAM will be equal to the size of the matrices as given in Tables 2, 3, 4, and 5 and (2). Depending on the type of memory (single port, simple dual port, true dual port, and RAM/ROM), the interface for the IP core is selected. We do not discuss mapping of arrays to memory blocks and selection of appropriate memory blocks as it is outside the scope of the current work.

Since the actual mapping to an IP core is done during the hardware binding phase in HLS flow, the information on the type of memory used for matrices is already available. Hence, the interface in the selected IP core is appropriately chosen. If the memory is a single port RAM or a simple dual port RAM or a true dual port RAM, *serial interface* on the IP core is automatically selected. Absolute addressing is used to generate addresses that are needed to read from the input matrices' BRAM and to write to the output matrix BRAM. Address generation for reading from input matrices and writing to output matrices is part of state S(c) in the state machine in Figure 6. The number of addresses to be generated for input and output matrices is given by the size of left, right, and product matrices as given in Tables 2, 3, 4, and 5 and (2).

3.7. Scheduling in the Presence of IP Cores. Once the matrix operation patterns and the keywords have been identified and extracted from the CDFG IR, the CDFG needs to be scheduled. Every IP core will have a latency and a throughput attribute associated with it. These attributes can take up different values depending on whether the architecture of the core is configurable or not. Since the library of IP cores has information on IP cores in a very structured manner, the scheduling step can easily query the library. It is evident from an examination of commonly available IP cores provided by FPGA vendors not only for the class of operations discussed in this paper but also for other functions that the IP cores can be sequential multicycle in nature or combinational multicycle. We use the SDC based scheduling [14] because it can easily take care of IP cores. We do not discuss research in scheduling further as it is outside the scope of the current work.

4. Experimental Setup, Results, and Their Discussion

A set of compute kernels in different applications involving different in-built C/C++ functions and matrix operations were selected. We show the area and clock frequency results for the compute kernels when mapped using a commercial HLS tool (Vivado-HLS version 2012.4) and when mapped under our proposed flow using IP cores to Xilinx FPGA devices. Table 7 shows the synthesis and implementation results for sqrt(), abs(), and integer divide that is div() in-built functions in C which were implemented as top modules so that design space exploration could also be performed.

Table 8 shows the results for matrix multiplication using the traditional flow (v-HLS) and the proposed flow. Table 9 shows the results for *exponential decay*(exp-decay)benchmark using single precision floating point (SPFP) data. Exponential decay is given by (7) and is widely used in studying radioactive decay, RC-circuit analysis, and so forth. Consider

$$N_t = N_0 e^{-\alpha t}. \tag{7}$$

In (7), N_t is some quantity measured at time t, N_0 is the same quantity measured at time $t = 0$, α is the decay constant and t is time.

Table 10 shows the synthesis and implementation results for a formula used to calculate *great circle distance* (GC) [46]. This formula returns the central angle between two geographical locations and is widely used in global positioning systems (GPS) and is given in (8). Only the parameter of $\cos^{-1}$ has been synthesized to maintain fairness with the v-HLS tool which does not support synthesis of $\cos^{-1}$. Consider

$$\begin{aligned} \mathrm{GC} = \cos^{-1}(&\sin(\mathrm{lat1}) \times \sin(\mathrm{lat2}) + \cos(\mathrm{lat1}) \\ &\times \cos(\mathrm{lat2}) \times \cos(\mathrm{lon1} - \mathrm{lon2})). \end{aligned} \tag{8}$$

In (8), lat1, and lat2 stand for latitude1 and latitude2, respectively, while lon1 and lon2 stand for longitude1 and longitude2, respectively, and they correspond to two different geographical locations on the surface of the earth.

4.1. Discussion of Results. In Table 7, we see that when bound to the IP core, the synthesis and implementation results are better not only for area (LUT, FF, etc.) and maximum clock frequency but also for latency. The square root function was mapped to a hardware IP core for Xilinx FPGAs. One main reason why the v-HLS flow resulted in high area values is because it converted 32-bit integer to 64-bit double while using the in-built C implementation of sqrt() function. This clearly demonstrates that such an approach which ignores hardware level IP and instead relies on software level algorithmic description for in-built functions may not result in efficient synthesis and implementation. For the

Table 7: Mapping of some representative in-built *C* functions.

Design	XC4VFX12FF668-10								
	I/P Data	LUT	FF	DSP	SRL	BRAM	CLK (ns)	Latency/throughput	IP core
sqrt (v-HLS)	32 bit integer	3105	4308	0	135	0	5.419[1]	75[2]/75	No
sqrt (proposed)	32 bit integer	358	363	0	0	0	5.323[1]	17[3]/1	CORDIC
div (v-HLS)	32 bit integer	4342	4105	0	136	0	7.572[4]	34[2]/34	No
div (proposed)-D1	32 bit integer	1230	1095	13	262/1230	0	5.674[4]	36/5	Divider[5]
div (proposed)-D2	32 bit integer	1224	1011	13	262/1230	0	6.769[4]	34/5	Divider[5]
div (proposed)-D3	32 bit integer	1150	937	13	234/1150	0	6.336[4]	30/5	Divider[5]
div (proposed)-D4	32 bit integer	2279	6720	0	72/2279	0	6.820[4]	68/1	Divider[5]
div (proposed)-D5	32 bit integer	1284	3244	0	68/1284	0	5.607[4]	36/1	Divider[6]
div (proposed)-D6	32 bit integer	1253	1971	0	4/1253	0	7.885	37/2	Divider[6]
abs (v-HLS)	16 bit integer	51	19	0	0	0	2.462	1/1	No
abs (proposed)	16 bit integer	32	32	0	0	0	3.121	1/1	Yes

[1]Clock period constraint of 4.5 ns with input jitter of 0.56 ns (clock uncertainty).
[2]Worst case and best case latencies.
[3]Maximum latency.
[4]Clock period constraint of 8 ns with input jitter of 1 ns (clock uncertainty).
[5]CORDIC configuration of integer quotient and fractional remainder with maximum pipelining.
[6]CORDIC configuration of integer quotient and integer remainder with maximum pipelining.

Table 8: Mapping of Matrix Multiplication Operation.

Design (2×2 matrix multiplication; 16 bit integer elements)	XC4VFX12FF668-10								
	LUT	FF	DSP	BRAM	CLK (ns)[7]	Latency/ throughput[8]	Wall clock time (ns)	IP core	Comments
D1 (v-HLS)	31	62	1	0	2.856	70/70	199.9	No	External memory (EM)
D2 (v-HLS)	39	138	8	0	2.856	9/9	25.7	No	EM
D3 (v-HLS)	135	86	1	3	3.247	N.A	N.A	No	—
D4 (v-HLS)	129	198	8	3	3.112	N.A	N.A	No	—
D5 (v-HLS)	124	103	1	3	3.215	117/117	376.1	No	—
D6 (v-HLS)	155	214	8	3	2.988	59/59	176.2	No	—
D7 (proposed)	269	622	2	4	2.956	22/4	11.8	Yes	—
D8 (proposed)	514	1688	8	0	3.126	17/1	3.1	Yes	—
D9 (proposed)	183	576	4	7	3.005	17/2	3.0	Yes	—

[7]Clock period constraint of 4 ns with 0.5 ns clock jitter.
[8]The best case and the worst case latencies were the same for v-HLS flows.

Table 9: Mapping of exponential decay formula.

Design	XC7VX330T-1FFG1157								
	I/P data	LUT	FF	DSP	SRL	BRAM	CLK (ns)[9]	Latency/throughput	IP core
exp-decay (v-HLS)	SP FP	2656	2147	40	237	0	4.486	56/56	No
exp-decay (proposed)-D1	SP FP	4539	4679	8	39/4539	0	2.649	61/30	Yes
exp-decay (proposed)-D2	SP FP	4633	4807	6	65/4633	0	2.618	63/32	Yes
exp-decay (proposed)-D3	SP FP	4561	4566	8	17/4561	0	4.568	57/26	Yes
exp-decay (proposed)-D4	SP FP	4571	4675	6	45/4675	0	2.725	61/30	Yes
exp-decay (proposed)-D5	SP FP	4539	4679	8	39/4539	0	2.649	26/20	Yes

[9]Clock period constraint of 5 ns with zero input jitter.

integer divide function div(), we again see that the proposed IP based flow results in better area, clock, latency, and throughput measures. It can also be seen that 6 different designs (D1 to D6) could be generated because at least 6 different configuration of the Xilinx divider IP core satisfied the conditions during IP selection phase as described in Section 3.5.

Results in Table 8, which shows the synthesis and implementation results for matrix multiplication operations, show that it is possible to generate multiple designs even when using Vivado-HLS flow, but these were obtained by *manually altering* the source code. Nevertheless, their latency and throughput results are very poor and some of them require memory blocks external to the compute block for the entire matrix multiplication exercise to yield results. On the other hand, graph based pattern matching resulted in three different designs corresponding to three different configurations of the Xilinx linear algebra toolkit IP core [8] *without altering* the source code. Also, their latency and throughput metrics were far better than the ones from traditional flow for the same clock period constraint though at the expense of some increase in area.

Results in Table 9 show that at least 5 different designs could be achieved using the different configuration of the IP core for exp() calculation and all satisfied the same clock period specification of 5 ns with no clock jitter (clock uncertainty). The Vivado-HLS flow resulted in around 40% less LUTs but at least 400% more DSP blocks. On the other hand, the designs using proposed flow used more LUTs but a lot less DSP blocks. This demonstrates the fact the tradeoff between DSP block and LUT could be the result of variation in the internal design of the IP core and the type of C-implementation of exp() function. The IP core used for exp() calculation made use of CORDIC IP core for Xilinx FPGAs. Nevertheless, it is evident from the results that the designs with hardware IP core achieved better clock frequency with around 40% faster clock and around 50% higher throughput. Xilinx Virtex-7 was used for this experiment because Vivado-HLS (version 2012.4) does not support the synthesis of exp() function for Virtex-4.

In Table 10, the Vivado-HLS result shows the estimated (Est) clock period for synthesis and implementation of the *greater circle distance* benchmark. This is so because the LUT utilization of 22522 is so high that this design could not fit into the target device. Again, these numbers are high because the v-HLS flow used single precision floating point for implementation because that is how the C based implementations of sin() and cosine() were available. Three designs were generated as per our proposed flow and they used only 4 DSP blocks while the LUT utilization varied from 3113 to 8567 depending on the IP core configuration. The IP core used was the Xilinx CORDIC IP core. All the designs could meet the clock specification of 5.5 ns period with zero clock jitter. When the polynomial expansions of sin() and cos() were used, the DSP utilization was 16 and 21 for 3 terms and 4 terms in the expansion, respectively. When DSP blocks were not used, their LUT utilization was higher than two of the IP core based designs. However, the two polynomial expansion based designs did not meet the timing specification. The latency figures for the IP core based design are also much better ranging from 18 to 50 for the three design versus 109 for Vivado-HLS and 41 and 51 for polynomial based synthesis. It should be noted here that in this particular case, algorithm for *area reduction by IP core resolution* (Section 3.5, Algorithm 6) in the flow resulted in lesser area because sin() and cos() are calculated for both lat*1* and lat*2* in the benchmark. Hence, instead of using separate CORDIC IP core for each, which would have resulted in a total of 4 instances of the core, only 2 instances were used thus leading to area savings.

4.1.1. IP Core Characterization. We would like to point out here that rigorous characterization of IP cores is very important especially with respect to their timing characterization for various configurations. It is this information that is crucial in the design flow and it affects all stages, from IP core selection to the synthesis of interfaces to calculation of latency and throughput. As an example, Table 11 shows the different configurable parameters of the Xilinx CORDIC IP core [45].

Not all configurable parameters will be available for all functions supported by the Xilinx CORDIC IP core. At the same time, some parameters cannot be configured when a particular parameter is selected. For instance, when configured for square root operation with 32 bit unsigned integer input, the output width is automatically set to 17 in the core and the architectural configuration is set to parallel. On the other hand, both input and output widths are configurable for sin() calculation. This nature of the IP core is typical of the way IP cores are developed and can be trivially observed in many available IP cores. Table 12 shows the configurable parameters and possible values when Xilinx CORDIC IP core is used to map the sqrt() function.

For experimental results reported in this paper, we performed the characterization of this core for the different functions that were looked at namely, sqrt(), sin(), and cos(). It should be noted that the resource utilization for a given design does not vary across different members of a device family. For instance, resource utilization for a given design does not vary across different devices in Xilinx Virtex-4 family. This is because the basic architecture comprising LUT, FF, and so forth remains the same in a device family and hence logic synthesis and technology mapping yield similar results during characterization. For example, all Xilinx Virtex-4 device family members will use the same amount of resources for a particular design. It is only the timing performance which can vary from one member to another. Table 13 shows a representative characterization data for sqrt() mapping.

5. On Integration of Proprietary IP Cores

The proposed IP based high level synthesis can easily deal with third party IP cores as well as those computations which can be identified based on graph based pattern matching like matrix operations. However, an additional *programmer's interface* is provided for using proprietary IP cores. This would enable *adaptive* compilation where the adaptation comes from the fact that the compilation of

Table 10: Mapping of greater circle distance formula.

Design	XC4VLX15-12SF363								
	I/P data	LUT	FF	DSP	SRL	BRAM	CLK (ns)[11]	Latency/throughput	IP core
GC (v-HLS)	16 bit	22522	11937	68	0	20	5.31 (Est)	109/109	No
GC (proposed)-D1	16 bit	3113	3136	4	15/3113	0	5.47	24/1	CORDIC[10]
GC (proposed)-D2	16 bit	1925	2062	4	15/1925	0	5.38	18/1	CORDIC[12]
GC (proposed)-D3	16 bit	8567	8719	4	33/8567	0	5.456	50/1	CORDIC[13]
GC (3-terms)	16 bit	894	3433	16	0	0	8.61	41/1	No
		3456	3664	0	0	0	9.95	41/1	No
GC (4-terms)	16 bit	2145	6729	21	84/2145	0	8.67	59/1	No
		5375	6959	0	84/5375	0	9.99	59/1	No

[10] Parallel configuration, iteration, and precision: automatic, coarse rotation.
[11] Clock period constraint of 5.5 ns with 0 ns clock jitter.
[12] Parallel architecture, iterations-8, and precision: automatic, coarse rotation.
[13] Parallel architecture, iterations-40, and precision: automatic, coarse rotation.

Table 11: Configurable parameters in Xilinx CORDIC IP core.

Configurable parameter (attribute)	Possible values (attribute values) (to select one)
Functional selection	Rotate, translate, sin and cos, sinh and cosh, arc tan, arc tanh, square root
Architectural configuration	Word serial, parallel
Pipelining mode	No pipelining, optimal, maximum
Data format	Signed fraction, unsigned fraction, unsigned integer
Phase format	Radians, scaled radians
Input width	Range 8 to 48
Output width	Range 5 to 48
Round mode	Truncate, round positive infinity, round positive negative infinity, nearest even
Iterations	Range 0 to 48
Precision	Range 0 to 48
Coarse rotation	Yes, no
Compensation scaling	No scale compensation, Lut based, Bram based, embedded multiplier

C/C++ description adapts to the availability of proprietary or legacy IP cores. By proprietary IP cores, we refer to those IP cores whose equivalent computation nodes in a CDFG are extremely difficult to identify based on pattern matching or standard keyword based lexical and semantic analysis. For instance, it is extremely difficult to identify a code segment that implements Fast Fourier Transform in an application. To the best of our knowledge there is no work on program understanding and algorithm recognition that is able to deterministically and efficiently identify such code segments. There are third party IP cores for such algorithms and there could as well be proprietary IP cores developed in-house.

If a proprietary IP core is given a name *PIPC* and if it implements a function which is given the function name *FUNCT* in the C/C++ description of an application, then a *programmer's interface* to the compiler can facilitate insertion of *FUNCT* to the symbol table. Therefore, the symbol table is composed of a *static part* and a *dynamic part* where static part could be updated by the programmer while the dynamic part is updated by the usual lexical and semantic analysis phase. The information in the *static part* is also used during the lexical and semantic analysis phases. The target backend code generator associates *FUNCT* in the symbol table with the proprietary IP core *PIPC* and any other relevant proprietary IP core.

It should be understood that a *programmer's interface* is most effective only for large IP cores like FFT modules, encoder-decoder blocks, and so forth, because it eases the task of updating the static part of symbol table and updating the target backend code generator. Also, it would not require restructuring the C/C++ source code of an application using such code blocks because typically these would be implemented as function calls. Doing this for operations like matrix operations would require the C/C++ source code of an application to be restructured as such smaller granularity computations are not always implemented as functions in a C/C++ description of an application.

The availability of *programmer's interface* also obviates the need to restructure C/C++ description using directives,

Table 12: Available configurable parameters for Xilinx CORDIC IP core implementing sqrt() function.

Configurable parameter (attribute)	Possible values (attribute values) (to select one)
Pipelining mode	No pipelining, optimal, maximum
Data format	Unsigned fraction, unsigned integer
Input width	Range 8 to 48
Round mode	Truncate, round positive infinity (RPI), round positive negative infinity (RPNI), nearest even (RNE)
Optional pins	Handshake signals (chip enable, CE; synchronous clear, SCLR; new data, ND; ready, rdy)

Table 13: Representative characterization data for CORDIC configured for sqrt().

Configurable parameter (attribute)	Attribute value (for 32 bit unsigned integer) (XC4VFX12FF668-10)			
Pipelining mode	No pipelining			
Round mode	Truncate	RPI	RPNI	RNE
Latency attained	2	2	2	2
Fmax obtained (MHz) (ns)	45.535 (21.961)	37.038 (26.999)	37.038 (26.999)	33.357 (29.979)
LUT/FF	327/27	375/27	375/27	416/27
Pipelining mode	Optimal			
Round mode	Truncate	RPI	RPNI	RNE
Latency attained	10	10	10	11
Fmax obtained (MHz) (ns)	147.558 (6.777)	149.298 (6.698)	149.298 (6.698)	149.365 (6.695)
LUT/FF	351/201	399/227	399/227	440/239
Pipelining mode	Maximum			
Round mode	Truncate	RPI	RPNI	RNE
Latency attained	17	18	18	19
Fmax obtained (MHz) (ns)	280.426 (3.566)	264.131 (3.786)	264.131 (3.786)	263.227 (3.799)
LUT/FF	358/363	403/380	403/380	444/419

pragmas and so forth. This maintains code portability and readability.

6. Conclusion

We have presented in this paper a framework for IP enabled C/C++ based high level synthesis and relevant methods and results to show that IP reuse and high level synthesis can be combined for better area-time results and more designs points *without any code restructuring* by a designer. We believe that the three-pronged approach to deal with IP cores, that is, lexical and semantic analysis, compiler aware pattern matching, and programmer's interface, can support all kinds of IP cores. The IP core view is an efficient abstracted view of the IP cores with only details relevant to high level synthesis. The methodology proposed can be used to implement any function, including user defined functions, using IP cores by following the graph based approach of which matrix multiplication was an example. However, lexical and semantic analysis based approach cannot be applied to user defined functions because users may choose to name such functions in any way they wish without any relation to the computation performed by those functions. Hence, we focus on the graph based approach for user defined functions restricting the lexical and semantic analysis based approach to standard in-built functions. While the results in this paper have been quoted for FPGAs, the framework, as well as the techniques, including IP core characterization and view, is applicable to high level synthesis of ASICs also. The ASIC IP cores would require characterization for supported technology nodes in this case. Also, since no source code restructuring or use of directives embedded in the source code is required (which is the case with all existing HLS flows), the proposed flow and methods are sure to increase designer productivity as well as acceptance of HLS design methodology. In addition to improving design performance, the presence of lower level optimized implementations of in-built functions in the IP library relieves a system designer from the worry of algorithms needed to implement such functions. This is particularly useful to those engineers who are more accustomed to software design and less to hardware design.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] R. Gupta, S. Gupta, N. D. Dutt, and A. Nicolau, *SPARK: A Parallelizing Approach to the High Level Synthesis of Digital Circuits*, Kluwer Academic, New York, NY, USA, 2004.

[2] GAUT, "GAUT: high-level synthesis tool from C to RTL," 2012, http://www-labsticc.univ-ubs.fr/www-gaut.

[3] H. L. S. Vivado, "Vivado high level synthesis," 2012, http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm.

[4] Catapult-C SYNTHESIS, "Catapult-C synthesis," 2012, http://calypto.com/en/products/catapult/overview.

[5] "ISO/IEC C 11 STANDARD," ISO/IEC 9899—Programming languages—C, 2011.

[6] ISO/IEC C++ 11 STANDARD, ISO/IEC 14882:2011 Programming Language C++.

[7] J. Detrey and F. deDinechin, "Table-based polynomials for fast hardware function evaluation," LIP Research Report 2004-52, 2004.

[8] Xilinx LogicCORE IP Linear Algebra Toolkit (LAT) v1. 0. March 1. DS829, 2011.

[9] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design and Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[10] A. Sangiovanni-Vincentelli, "Quo vadis, SLD? Reasoning about the trends and challenges of system level design," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007.

[11] J. M. P. Cardoso, P. Diniz, and M. Weinhardt, "Compiling for reconfigurable computing: a survey," *ACM Computing Surveys*, vol. 42, no. 4, article 13, 2010.

[12] D. D. Gajski, N. D. Dutt, A. C. H. Wu, and S. Y. L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic, New York, NY, USA, 1992.

[13] C. Bobda, *Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications*, Springer, New York, NY, USA, 2007.

[14] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proceedings of the 43rd IEEE/ACM Design Automation Conference*, pp. 433–438, ACM, New York, NY, USA, 2006.

[15] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kmmoona, T. Czajkwoski et al., "LegUp: an open source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems*, vol. 1, no. 1, article 1, 2012.

[16] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu, "Data path allocation based on bipartite weighted matching," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 499–504, June 1990.

[17] J. Cong and J. Xu, "Simultaneous FU and register binding based on network flow method," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 1057–1062, IEEE, Los Alamitos, CA, USA, March 2008.

[18] T. Kim and X. Liu, "Compatibility path based binding algorithm for interconnect reduction in high level synthesis," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '07)*, pp. 435–441, IEEE, Los Alamitos, CA, USA, November 2007.

[19] U. Dhawan, S. Sinha, S.-K. Lam, and T. Srikanthan, "Extended compatibility path based hardware binding algorithm for area-time efficient designs," in *Proceedings of the 2nd Asia Symposium on Quality Electronic Design (ASQED '10)*, pp. 151–156, IEEE, Los Alamitos, CA, USA, August 2010.

[20] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Robust FPGA intellectual property protection through multiple small watermarks," in *Proceedings of the 36th Annual Design Automation Conference (DAC '99)*, pp. 831–836, IEEE, Los Alamitos, CA, USA, June 1999.

[21] A. L. Oliveira, "Robust techniques for watermarking sequential circuit designs," in *Proceedings of the 36th Annual Design Automation Conference (DAC '99)*, pp. 837–842, IEEE, Los Alamitos, CA, USA, June 1999.

[22] G. Qu and M. Potkonjak, "Fingerprinting intellectual property using constraint-addition," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 587–592, IEEE, Los Alamitos, CA, USA, June 2000.

[23] A. Deshpande, "Verification of IP-Core based SoC's," in *Proceedings of the 9th IEEE International Symposium on Quality Electronic Design (ISQED '08)*, pp. 433–436, IEEE, Los Alamitos, CA, USA, 2008.

[24] M. S. McCorquodale and R. B. Brown, "UMIPS: a semiconductor IP repository for IC design research and education," in *Proceedings of the American Society for Engineering Education Annual Conference & Exposition*, June 2004.

[25] IP-XACT Standard, IEEE 1685–2009. IEEE standard for IP-XACT, standard structure for packaging, integrating and reusing IP within tool flows.

[26] Y. Lu and H. Zhou, "Efficient design space exploration for component-based system design," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '12)*, pp. 466–472, IEEE, Los Alamitos, CA, USA, 2012.

[27] Y. Liu, Y. Yang, and J. Hu, "Clustering-based simultaneous task and voltage scheduling for NoC systems," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '10)*, pp. 277–283, IEEE, Los Alamitos, CA, USA, November 2010.

[28] A. Baganne, I. Bennour, M. Elmarzougui, R. Gaeich, and E. Martin, "A multi-level design flow for incorporating IP cores-case study of 1D wavelet IP integration," in *Proceedings of the IEEE Design Automation and Test in Europe (DATE '03)*, pp. 250–255, Los Alamitos, CA, USA, 2003.

[29] C. Trummer, C. Ruggenthaler, C. M. Kirchsteiger et al., "Searching extended IP-XACT components for SoC design based on requirements similarity," *IEEE Systems Journal*, vol. 5, no. 1, pp. 70–79, 2010.

[30] R. Metzger and Z. Wen, *Automatic Algorithm Recognition and Replacement*, MIT Press, Boston, Mass, USA, 2000.

[31] D. Batten, S. Jinturkar, J. Glossner, M. Schulte, and P. D'Arcy, "New approach to DSP intrinsic functions," in *Proceedings of the 33rd Annual Hawaii International Conference on System Siences (HICSS-33)*, IEEE, Los Alamitos, CA, USA, January 2000.

[32] R. Stallman, *Using and Porting GNU CC*, version 2. 7. 2. 1., Free Software Foundation, 1996.

[33] H. Li, W. He, Y. Chen, L. Eeckhout, O. Temam, and C. Wu, "SWAP: parallelization through algorithm substitution," *IEEE Micro*, vol. 32, no. 4, pp. 54–67, 2012.

[34] C. Alias and D. Barthou, "Algorithm recognition based on demand-driven data-flow analysis," in *Proceedings of the 10th Working Conference on Reverse Engineering*, pp. 296–305, IEEE, Los Alamitos, CA, USA, November 2003.

[35] J. Cong and W. Jiang, "Pattern-based behavior synthesis for FPGA resource reduction," in *Proceedings of the 16th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '08)*, pp. 107–116, New York, NY, USA, February 2008.

[36] T. Ly, D. Knapp, R. Miller, and D. MacMillen, "Scheduling using behavioral templates," in *Proceedings of the 32nd Design Automation Conference*, pp. 101–106, IEEE, Los Alamitos, CA, USA, June 1995.

[37] A. Prakash, S. K. Lam, C. T. Clarke, and T. Srikanthan, "FPGA-aware techniques for rapid generation of profitable custom instructions," *Microprocessors and Microsystems*, vol. 37, no. 3, pp. 259–269, 2013.

[38] K. Atasu, L. Pozzi, and P. Lenne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proceedings of the 40th Design Automation Conference*, pp. 256–261, IEEE, Los Alamitos, CA, USA, June 2003.

[39] P. Bonzini and L. Pozzi, "Polynomial-time subgraph enumeration for automated instruction set extension," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 1331–1336, IEEE, Los Alamitos, CA, USA, April 2007.

[40] P. Brisk, A. Kaplan, R. 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 '02)*, pp. 262–269, ACM, New York, NY, USA, October 2002.

[41] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '04)*, pp. 69–78, ACM, New York, NY, USA, September 2004.

[42] "LLVM Compiler Infrastructure," 2013, http://www.llvm.org.

[43] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, New York, NY, USA, 2nd edition, 2006.

[44] R. Metzger, "Automated recognition of parallel algorithms in scientific applications," 1995.

[45] Xilinx LogiCORE IP CORDIC v4. 0. March 1, 2011. DS249.

[46] R. W. Sinnott, "Virtues of the haversine," *Sky and Telescope*, vol. 68, no. 2, article 159, 1984.