Benchmarking FPGA High-Level Synthesis

Zhihua Li[†,‡], Colin Yu Lin[†], Juan Huang[†], Yuanqiang Li[†], Cheng Liu[¶],
Liqun Yang[†,‡], Hayden Kwok-Hay So[¶], and Haigang Yang[†,§]
† System on Programmable Chip Research Department,
Institute of Electronics, Chinese Academy of Sciences, Beijing, China
‡ The University of Chinese Academy of Sciences, Beijing, China
¶ Department of Electrical and Electronic Engineering, University of Hong Kong, Hong Kong
§ Corresponding Author: yanghg@mail.ie.ac.cn

***Abstract*—Due to the lack of benchmarks, currently, FPGA High Level Synthesis(HLS) tools cannot be evaluated effectively. To solve this problem, this paper develops a suite of open-source benchmarks in C/C++ format for HLS. 20 benchmarks, which cover numerous fields including network, communication, multimedia and image processing, are carefully selected from the well-known SPEC CPU 2006, MiBench, MediaBench and LINPACK benchmarks. Since the current High Level Synthesis is not supportive of specific characteristics in C/C++, to apply CPU benchmarks to HLS, modifications and refactoring of code are carried on the 20 C/C++ benchmarks without changing the original functions. During the process, this paper proposes template metaprogramming to accommodate the recursion algorithm in HLS and an efficient memory allocation algorithm to support functions for dynamically allocating memory, such as malloc. After conversion, we implement the 20 benchmarks on Vivado, Xinlinx, and evaluate the resources cost by each benchmark. The experimental results show that the benchmark suite cost BRAM from 124KB to 321451KB, DSP from 1232 to 23455, FF from 4566 to 12256, and LUT from 2134 to 124324.**

I. Introduction

The increasing design complexity pushes the design community to seek high level design abstractions with higher productivity over the conventional register transfer level (RTL). High-level synthesis (HLS), which enables the automatic synthesis of high-level specifications to low-level cycle-accurate RTL specifications for efficient implementation on ASICs and particularly FPGAs, is one of the most promising solutions. There are already a great number of HLS tools so far from both the industry and academia as summarized in [**?**].

However, the HLS tools usually could only support a subset characteristics of a specified high level language and the synthesizable subsets also vary a lot, therefore, a benchmark that could fit most of the HLS tools is still greatly needed. In addition, HLS tools target random logic synthesis and the benchmark should cover a large number of representative domains of applications. For these purposes, we build a benchmark HLS-Bench which involves 20 applications selected from LINPACK, MediaBench, MiBench, and SPEC CPU 2006. It covers various application domains including networking, communication, security, multimedia, image processing and so on. In addition, instead of using a static data set, we have three diverse data sets for each of the application, which helps insight in the scalability of the HLS tools. To make sure the benchmark be easily supported by the general HLS tools, we have performed a series of code transformation like replacing dynamic link list data structure with static arrays, changing the recursive implementation with non-recursive one, developing dynamic memory manager to support some of the *malloc* and *free* and so on. This benchmark suite is available *HLS-Bench* under LGPL license.

Finally, we have the benchmark implemented using Xilinx Vivado HLS and evaluate the timing, performance as well as the hardware overhead. The experiments show that the benchmark covers a diverse scale of implementation where bram, dsp, ff and lut consumption range from 124kb to 321451kb, 1232 to 23455, 4566 to 12256 and 2134 to 124324 respectively.

The contributions of this paper can be summarized as below:

- An open source benchmark suite in c/c++ for FPGA HLS is presented, which covers a wide range of application domains.
- Code transformations are studied and implemented to fit various HLS tools with limited syntax support
- A state-of-art commercial FPGA HLS tool is evaluated using the HLS-Bench.

In Section II, we review the evolution of both ASIC and FPGA HLS benchmarks. In Section III, we briefly describe features of each application in HLS-Bench. In Section IV, code transformation strategies that the ease the availability of HLS tools are discussed. In Section V, we present the implementation results of HLS-Bench using Vivado HLS. Finally, in Section VI, we conclude this paper.

II. Related Works

Some of the commencing efforts in HLS benchmarking were made in the late 1980s. High Level Synthesis Workshop 1992 Benchmarks [**?**] and 1995 High Level Synthesis Design Repository [**?**] were two standard benchmarks released for HLS. However, HLS tools in [**?**] [**?**] actually referred to compiling the HDL code to circuits and they were written in VHDL. These benchmarks were transplanted to C later, but they were tiny including less than one hundred lines of C code. Therefore, they are no longer suitable for benchmarking the latest HLS tools.

CHStone [**?**] is then proposed to provide a more practical benchmark suite for the HLS tools. It consists of 12 easy-to-use programs written in C and includes a number of application domains such as arithmetic, security, microprocessor, media

processing and so on. Moreover, the programs in CHStone are much larger compared with the HLS92 and HLS95. *However, construct data structure and dynamic memory allocation, which are removed for the HLS tools at that time, now turn to important supporting features of the latest HLS tools.* Moreover, just as the authors declared, CHStone doesn't intend to evaluate the commercial HLS tools.

S2CBench written in SystemC was proposed to enable the direct comparison of commercial HLS tools [**?**]. It consists of 12+1 programs which target a variety of applications. The 12 benchmarks comply with the latest SystemC synthesizable subset draft, while the rest one is FFT and it is non-synthesizable due to the trigonometric and floating point operations. The non-synthesizable FFT is added in order to help the users to understand how the different HLS tools work around these special occasions. Another unique feature of S2CBench is that each application is accompanied by its corresponding testbench. Nevertheless, the application domain covered in S2CBench is relatively limited. Typical applications such as Multimedia and network are not involved.

BDTI High-Level Synthesis Tool Certification Program *citation is needed* adopts a video motion analysis application and a wireless receiver to evaluate the high-level synthesis tools. The application is big enough for evaluation but the application domain is again quite limited.

HLS-Bench developed in this paper covered a large range of application domains. Also it uses data structures and syntax supported in the latest HLS tools, which helps evaluate the HLS tools. Moreover, each application is provided with three diverse data sets and it is beneficial to gaining insight into the HLS tools.

III. Benchmark Suite for FPGA HLS

In this section, we describe a brief overview of Benchmark Suite for FPGA HLS, and then features of individual programs in our benchmarks are summarized.

A. Overview

FPGA HLS suite is a collection of 20 C/C++ programs. Some of the main objectives of our benchmarks are: 1) To evaluate performance of commercial HLS tools such as synthesis optimization techniques. 2) To provide FPGA HLS researchers a set of benchmark programs which are easy to use. 3) To provide FPGA HLS designer some utilized functions such as rewrited string library functions and techniques such as dynamically memory allocation.

Our 20 benchmark programs which have been selected from various application domains covering most fields like network, communication, safety, multimedia, image processing and so on. They are from the well-known CPU benchmarks such as SPEC CPU 2006, MiBench, MediaBench and LINPACK benchmarks. Tables I summarize the features of the benchmark programs. It shows the brief description and the sources of the programs. Table II describes some characteristics of the programs such as the number of lines of C code, the number of loop and condition clause, the number of struct type and the number of dynamically memory allocation and deallocation clause. As shown in two tables, the characteristics

TABLE I. FPGA HLS Benchmarks

Benchmark	Source	Domain
linpack	LINPACK Benchmark	Arithmetic
astar	SPEC CPU 2006	Network and Games
qsort	MiBench	Automotive and Industrial Control
stringsearch	MiBench	Office
fft	MiBench	Telecomm
crc32	MiBench	Telecomm
adpcm	MiBench	Telecomm
susan	MiBench	Multimedia and Consumer
ispell	MiBench	Office
Dijkstra	MiBench	Network
bitcount	MiBench	Automotive
basicmath	MiBench	Automotive and Arithmetic
rijndael	MiBench	Security
sha	MiBench	Security
blowfish	MiBench	Security
More...		

of our benchmark programs are: 1) Contain eight complex multimedia processing algorithms including the encoding and decoing of image, audio and video. Their number of lines of C/C++ code exceeds 1500. 2) Have composite data types such as struct which is supported for most nowaday FPGA HLS tools. 3) Involved with dynamic memory allocation and deallocation. ???(number) benchmark programs contains malloc and free function in C language. 4) Two CPU benchmarks (qsort and bitcount) involves recursion, we change it in a iterative way. In a word, our benchmark programs are brought from widely-used applications in the real world. They cover most of the C/C++ features which FPGA HLS designers usually expect to be included in their design.

B. Benchmark programs

Every program in our FPGA HLS benchmarks is described below. It is a brief overview of the programs included in the benchmark suite.

(FIXME change font)linpack: It is a measure of a system's floating point computing power. It measures how fast to solve a dense n by n system of linear equations Ax = b, which is a common task in engineering.

astar: It is a computer algorithm that is widely used in pathfinding and graph traversal. It uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals). As astar traverses the graph, it follows a path of the lowest expected total cost or distance, keeping a sorted priority queue of alternate path segments along the way.

qsort: The qsort test sorts a large array of strings into ascending order using the well known quick sort algorithm. Sorting of information is important for systems so that priorities can be made, output can be better interpreted, data can be organized and the overall run-time of programs reduced [**?**].

stringsearch: This benchmark searches for given words in phrases using a case insensitive comparison algorithm.

fft: This benchmark performs a Fast Fourier Transform on an array of data. Fourier transforms are used in digital signal processing to find the frequencies contained in a given input signal.

crc32: This benchmark performs a 32-bit Cyclic Redundancy Check (CRC) on a file(FIXME). CRC checks are often

TABLE II. FPGA HLS BENCHMARKS

Benchmark	Lines of code	Clauses of condition	Clauses of loop	Strcut types (C struct and C++ class)	Dynamically memory allocation	Change recursion to iteration
linpack	741	57	38	0	0	No
astar	550	41	11	0	0	No
qsort	75	1	0	2	1	Yes
stringsearch	407	25	27	0	0	No
fft	385	10	7	0	0	No
crc32	254	0	2	0	0	No
adpcm	351	22	3	1	0	No
susan	2281	162	60	3	5	No
ispell	925	36	34	4	1	No
Dijkstra	150	5	3	3	0	No
bitcount	885	21	18	0	0	Yes
basicmath	350	1	4	1	0	No
rijndael	1767	36	33	1	0	No
sha	350	6	14	1	0	No
blowfish	534	4	9	1	0	No
More	TOADD					

used to detect errors in data transmission.

adpcm: Adaptive Differential Pulse Code Modulation (ADPCM) is a variation of the well-known standard Pulse Code Modulation (PCM). A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1.

susan: Susan is an image recognition package. It was developed for recognizing corners and edges in Magnetic Resonance Images of the brain. It is typical of a real world program that would be employed for a vision based quality assurance application. It can smooth an image and has adjustments for threshold, brightness, and spatial control.

ispell: Ispell is a fast spelling checker. It supports English word spell correction suggestions.

Dijkstra: The Dijkstra benchmark constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm. Dijkstra's algorithm is a well known solution to the shortest path problem and completes in O(n^2) time.

basicmath: The basic math test performs simple mathematical calculations. For example, cubic function solving, integer square root and angle conversions from degrees to radians are all necessary calculations for calculating road speed or other vector values.

bitcount: The bit count algorithm tests the bit manipulation abilities by counting the number of bits in an array of integers. It does this using five methods including an optimized 1-bit per loop counter, recursive bit count by nibbles, non-recursive bit count by nibbles using a table look-up, nonrecursive bit count by bytes using a table look-up and shift and count bits.

rijndael encrypt/decrypt: Rijndael was selected as the National Institute of Standards and Technologies Advanced Encryption Standard (AES). It is a block cipher with the option of 128-, 192-, and 256-bit keys and blocks.

sha: SHA is the secure hash algorithm that produces a 160-bit message digest for a given input. It is often used in the secure exchange of cryptographic keys and for generating digital signatures. It is also used in the well-known MD4 and MD5 hashing functions.

blowfish encrypt/decrypt: Blowfish is a symmetric block cipher with a variable length key. It was developed in 1993 by Bruce Schneider. Since its key length can range from 32 to 448 bits, it is ideal for domestic and exportable encryption.

TODO: add more benchmarks here...

IV. CODE TRANSFORMATIONS

We select 20 benchmarks from the well-known CPU benchmarks. As mentined above, to apply these benchmarks to FPGA HLS, some important refactoring and modifications to the original benchmark codes should be done. This section will discuss some code transforming ttrategy for FPGA HLS benchmark.

A. Code Refactoring

Code refactoring is the process of restructing existing computer code - changing the factoring - that either presevers the behavior of the software, or at least does not modify its conformance to functional requirements [**?**]. It improves the code readability and reduced complexity, making the source code of more maintainability and extensibility. Some of choosed CPU benchmarks code mixed data process logic with the data input (such as read processing data from file which is not synthesizable by FPGA HLS tools) or output (such as write processed data to file or add some timing function to calculate the procecure executing time). We have to take the data process logic apart and let it be the FPGA HLS benchmark. Besides, some functions of the benchmarks are too long, we trimed them with severval smaller functions. Good news is that most CPU benchmarks have test input file and the golden result output file, we can quickly find the error of illegal code modifications. So the code refactoring work is of not too much impedient.

B. Pointer

A pointer is an address to a location in memory. Some of the common uses for pointers in a C/C++ program are function parameters, array handling, pointer to pointer, and type casting. The inherent flexibility of this language construct makes it a useful and popular element of C/C++ code. However, The FPGA HLS compiler supports pointer usage that can be completely analyzed at compile time. An analyzable pointer usage is usage that can be fully expressed and computed without

the need for runtime information. Thus, the use of a pointer to reference a dynamically allocated region in memory such malloc, new function call in C/C++ code is is not supported with HLS, because the destination address of the pointer is only known during program execution, see Figure 1. This does not mean that pointer usage for memory management is unsupported when using the HLS compiler. Figure 2 shows a valid coding style in which pointers are used to access a memory. This code is valid, because all uses of pointer pointer_to_data can be analyzed and mapped back to array data. Because array data is created by automatic memory allocation, HLS can fully determine the properties of data.

```
int *pointer_runtime = malloc(sizeof(int) * 100);
```

Fig. 1. Unsupported Dynamic Memory Allocation in HLS

```
int data[100];
int *pointer_to_data = data;
```

Fig. 2. Supported Pointer in HLS: Managing Array Access with a Pointery

Another supported pointers is in accessing external memory. When using HLS, any pointer access on function parameters implies either a variable or an external memory. HLS defines an external memory as any memory outside of the scope of the compiler-generated RTL. This means that the memory might be located in another function in the FPGA or in part of an off-chip memory, such as DDR???. But you should tell the HLS compiler the memory size so as to let pointer be completely analyzed at compile time. Take Figure 3 of accumulating an array data as example. This code is invalid, because the data pointer is (or contains) an array with unknown size at compile time. Figure 2 is the fixed version of Figure 3. It is synthesizable because the array size of data is certain at compile time.

```
int accumulate1(int *data,int length) {
    int sum = 0;
    for (int i = 0; i < length; ++i) {
        sum += data[i];
    }
    return sum;
}
```

Fig. 3. Unsynthesizable code: accumulate an array

```
#define LENGTH 100
int accumulate2(int data[LENGTH]) {
    int sum = 0;
    for (int i = 0; i < LENGTH; ++i) {
        sum += data[i];
    }
    return sum;
}
```

Fig. 4. Synthesizable code: accumulate an array.

During the code transforming for FPGA HLS benchmarks from CPU benchmarks, there is much pointer usage as function parameter like accumulate1 in Figure 3. Thus, we have to change it in form of size-fixed array pointer as shown in Figure 4.

C. Library Functions

In CPU benchmarks, there are many C string and memory library functions call such as memset, memcpy, strlen and strcmp and so on. But these functions are unsupported in FPGA HLS compiler. We rewrite them in our own way. It is a simple case, taking the rewrited string lengh calculating function strlen as example in Figure 5.

```
#define MAX_CHAR_SIZE 1024

unsigned hls_strlen(char string[MAX_CHAR_SIZE]) {
    unsigned length = 0;
    for (length = 0; string[length] != 0; ++length);
    return length;
}
```

Fig. 5. Rewrited HLS strlen Function.

D. Dynamic Memory Allocation

Dynamic memory allocation is one of the memory management techniques available in the C and C++ programming languages. In this method, the user can allocate as much memory as necessary during program runtime. The size of the allocated memory can vary between executions of the program and is allocated from a central physical pool of memory in computer. The function calls typically associated with dynamic memory allocation are malloc, new for allocating memory and free, delete for deallocating memory. An FPGA does not have a fixed memory architecture onto which the HLS compiler must fit the user application. Instead, HLS synthesizes the memory architecture based on the unique requirements of the algorithm. As discussed in IV-B, all code provided to the HLS compiler for implementation in an FPGA must use compile time analyzable memory allocation only. It is the responsibility of the user to manually change the code and remove all instances of dynamic memory allocation.

There are two memory allocation ways that are supported by a C/C++ program and HLS compiler. They are automatic memory allocation and static memory allocation.

The code in Figure 6 shows automatic memory allocation by a HLS compiler. HLS implements this memory style in strict accordance with the behavior stipulated by C/C++. This means that the memory created to store array only stores valid data values during the duration of the function call containing this array. Therefore, the function call is responsible for populating array with valid data before each use.

```
int array[100];
```

Fig. 6. HLS-Compliant Automatic Memory Allocation

The code in Figure 7 shows static memory allocation by a HLS compiler. The behavior for this type of memory allocation dictates that the contents of array are valid across function calls until the program is completely shut down. When working with HLS, the memory that is implemented for array A contains valid data as long as there is power to the circuit.

As discussed before, the code in Figure 8 is not supported by HLS compiler. It allocates a region in memory to store

```
static int array[100];
```

Fig. 7. HLS-Compliant Static Memory Allocation

allocated_size values of 32-bit integer. We should consider to solve this problem in two cases whether the allocated_size is constant or not.

```
int *array = malloc(sizeof(int) * allocated_size);
```

Fig. 8. Dynamic Memory Allocation

If the allocated_size is constant and can be analyzed in compile time, use automatic and static memory allocation techniques are possible methods of modifying this code to comply with HLS. Just modify them with size-fixed array such as in Figure 6 or 7.

In CPU benchmarks, there are a few dynamically allocating memory code in the form of constant allocated size. More is in form of non-constant allocated size. In such a situation, we propose a dynamically allocating memory algorithm using bitmaps to solve this problem. Since the allocated memory size is merely known during runtime execution and the user usually deallocate this memory when the application exits, we use static memory allocation technique in our algorithm. With a bitmap, the static memory is divided into allocation chunks. Every chunk has units with a certain size. One unit can be as small as a byte or as large as several kilobytes. It is up to the size of basic data structure such as C/C++ char, int or struct in which type data is to be allocated dynamically at runtime. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure 9 shows five chunks of memory of static memory and the corresponding bitmap.

A bitmap provides a simple way to keep track of memory in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation chunk. The size of the allocation chunks is an important design issue. The smaller the allocation chunk, the larger the bitmap. During FPGA HLS design, the static memory is stored externally. What this can often imply when hardware is synthesized is that either the data in memory is expected every clock cycle, or there is some sort of off-chip storage in either registers or memory. More memory allocation chunks can increase data access parallelism in the HLS generated implementation. However, even with an unit as small as 4 bytes, 32 bits of memory will require only 1 bit of the map. A memory of 32n bits will use n map bits, so the bitmap will take up only 1/33 of memory. If the allocation chunk is chosen large, the bitmap will be smaller and has an adverse effect on data access parallelism in FPGA HLS design.

The dynamically allocating memory algorithm is shown in Algorithm 10. When the user request K units memory at runtime, it will search the bitmap to find a run of K consecutive 0 bits in the map and return corresponding memory location in the static area to user. Finally, set the K consecutive 0 bits in bitmap to 1.

The dynamically deallocating memory algorithm is simple.

Fig. 9. TODO FIXME (a) HLS static memory with five memory allocation chunks. Every chunk has eight units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap.

```
Prerequisite:
M, pre-allocated static memory
B, bitmap for allocation chunks, all bits initialized with 0
K, user requested K units memory at runtime
S, memorizing allocated memory, used for deallocate

Seek for K consecutive 0 bits in the bitmap B.
Let the found location be F.
Set K consecutive 0 bits to 1 at the beginning F in bitmap B.
Return the corresponding location in M to the location F of bitmap B.
Record F, K and the above returned result in S.
```

Fig. 10. (TODO, FIXME) Dynamic Memory Allocation Algorithm in HLS

As shown in Algorithm 10. There is a record for user requested memory allocation every time. Thus, when user wants to deallocate the requested memory at runtime, just find the corresponding location in bitmap and set them to 0.

(FIXME ugly)The proposed dynamically allocating memory algorithm using bitmaps make HLS support functions as malloc and free. The work transforming CPU benchmarks to FPGA HLS is more convenient and efficient.

E. Data Structure and Algorithm Modifications

In some cases when applying the CPU benchmarks to FPGA HLS, we have to modify the basic data structures utilized in these benchmarks. The reason lies in that the current HLS compiler constrains the use of pointer, which makes it unable to use the pointer variable in the C/C++ struct. This means that the HLS can not use the data structure like list, which usually contains the pointer variables pointing to the next node in one of its nodes' struct. Thus, we have to modify the list and the corresponding algorithm of the benchmarks which use it. Since the HLS compiler supports fixed-size data array, the data array can be utilized to replace list. Employing this idea, we do modifications to the benchmarks as follows. 1) We realize the queue in dijkstra CPU benchmark with array instead of the list. 2) The original hash table data structure in ispell benchmark utilized list to solve the collision existing in hash, whereas we make use of quadratic probing to achieve this and employ the way based on array copy to realize the rehasing by using the HLS dynamically allocating algorithm discussed in section IV-D.

It is a relatively difficult and rather time-consuming work to modify the data structure of benchmarks. In order to modify them, it is necessary to understand most parts of the benchmarks' codes, especially the code related to critical data structure and algorithm. The test sets and the golden results provided in the original CPU benchmarks gurantee the correctness of modifications we do on the benchmarks.

F. Recursive Algorithm

Recursion is the process a procudure goes through when one of the steps of the procecure involves invoking the procecure itself. A procecure that goes through recursion is said to be recursive. As a computer programming technique, this is called divide and conquer and is key to the design of many important algorithms.

However, recursion is not supported by nowaday hls compiler. Some specific recursion can be solved to obtain a non-recursive definition. Thus we have to change some recursive algorithms in CPU benchmark to non-recursive way(FIXME ugly!).

For example, in qsort CPU benchmark, it uses the system library function qsort to utilize the sort of points data in three dimension space. The qsort library function uses classic recursion algorithms to implement quick sort, given in Figure 11 in pseudocode. It sorts elements low through high of an array. If you want to get more details about the quick sort, you can refer to [**?**].

```
quicksort(array, low, high):
  if low < high:
    p := partition(array, low, high)
    quicksort(array, low, p - 1)
    quicksort(array, p + 1, high)
```

Fig. 11. Pseudocode of Recursive quick sort Function.

We modified the recursive version of quick sort in iterative way as shown in Figure 12. As you see, we make a stack using array to store array element position.

```
iterative-quicksort(array, low, high):
  create an auxiliary stack SA to store low, high
  push initial values of low and high to SA
  while SA is not empty:
    pop high and low from stack SA
    p := partition(array, low, high)
    if (p - 1) > low:
      push low and (p - 1) to stack
    if (p + 1) < high:
      push (p + 1) an high to stack
```

Fig. 12. Pseudocode of iterative quck sort Function.

Some specific recursion can be solved to obtain a non-recursive definition such as the iterative quick sort discussed above. In some cases, it is hard even impossible to "translate" from recursion to non-recursion. But there is not much complex recursion in CPU benchmarks during our transforming code work.

V. Experiments

Most of our FPGA HLS benchmark programs are large applications consisting of multiple hundreds lines of code. We have confirmed that all of the programs are synthesizable. In this section, we synthesize the FPGA HLS benchmarks proposed in this paper in FPGA HLS tool to watch the resource cost and timing information. The commercial Xilinx Vivado [**?**] (2013.4 Version) is employed as the FPGA HLS tool in this experiment. Meanwhile, the benchmarks are implemented on xc7v2000tflg1925-1 device, one typical chip in virtex-7 series whose speed grade is -1. The reason why we choose the chip is that its available resources are sufficient so that we can implement our benchmarks without warrying the lack of resources.

The available resources on this chip are summarized in table III. The BRAM Block (BRAM) is a configurable memory module that attaches to a variety of BRAM Interface Controllers. It is a dual-port RAM module instantiated into the FPGA fabric to provide on-chip storage for a relatively large set of data. The type of BRAM memories available in xc7v2000tflg1925-1 device can hold either 18k bits. The DSP48E block is an arithmetic logic unit (ALU) embedded into the fabric of the FPGA. It supports many independent functions of digital signal processing. These functions include multiply, multiply accumulate (MACC), multiply add and so on. As shown in table III, the DSP48E block in xc7v2000tflg1925-1 device available is 2160.

TABLE III. Available resources of xc7v2000tflg1925-1

BRAM_18K	DSP48E	Flip-Flop	LUT
2584	2160	2443200	1221600

TABLE V. HLS sythesis result for cosine, sine and exponent, logarithm function

	Utilization Estimates			
C math function	BRAM_18K	DSP48E	FF	LUT
cos	19	17	2574	8896
sin	19	17	2573	8821
exp	0	26	1124	2666
log	0	61	1755	1464

During experiment, We set the target clock period to 10ns, and other options to default. Beisides, we do not do optimizations on benchmarks during the experiment. The sythesis result for FPGA HLS benchmark programs are shown in Table IV.

Since the bitcount, basicmath and blowfish benchmarks do not rely on any specific parameters, the variety of FPGA resources utilized are not included in Table IV. According to Table IV, we can draw conclusions as follows (1) The clock period of the benchmarks do not change with the scale of data processed. (2) The utilization of DSP increases slowly with the input parameters, in some cases keeps unchanged. (3) Dijkstra, stringsearch and astars use more RAM with input paramters increasing while other benchmarks use the same RAM. The reason lies in that Dijkstra, stringsearch and astars use global variables to solve problems. For example, the Dijkstra benchmark uses a global array to store the distance information between graph nodes and the space needed by these global variables increases with the scale of the data processed. Since the global variables are stored in the FPGA RAM in HLS, the utilization of RAM increases with the input parameters. For the benchmarks which do not consume changed RAM with input parameters, they hardly use global variables and almost use the divide and conquer algorithm to implement the benchmark. For example, the AES algorithm utilized in the rijndael benchmark does not encrypt directly on the whole input data flow, but divides the data to 16 blocks, and processes the small data blocks in turn, through which reduces the use of memory.

Among the HLS benchmarks, the program basicmath costs the FF and LUT most while there is only 1 conditional block and 4 circulation blocks in less than its 350 lines of code. In addition, this program neither includes dynamic memory allocation, nor uses global or static variabilities. However, 38 BRAM_18K are consumed by basicmath. Through analysis, we discover that basicmath employs trigonometric functions, cosine and sine, exponent and logarithm operation to slove a cubic polynomial. We analysize and show the resources cost by the functions mentioned above in basicmath in Table . It can be seen that the cosine and sine functions cost lots of

TABLE IV. SYTHESIS RESULT FOR FPGA HLS BENCHMARK PROGRAMS

Benchmark Feature	Input Parameters	Timing Estimates	Utilization Estimates			
		Clock Period (ns)	BRAM_18K	DSP48E	FF	LUT
linpack Solve ax=b with matrix dimension size N of b	N=50	8.7	0	114	11307	13738
	N=100	8.7	0	114	11522	14018
	N=1000	8.7	0	114	12184	14824
Dijkstra Calculates the shortest path between every pair of nodes in a graph represented by adjacency matrix with size N×N	N=10	7.98	451	0	402	975
	N=100	7.98	482	1	478	1080
	N=1000	7.98	2436	1	551	1190
qsort Sorts an array with size N	N=1000	7.92	35	0	1861	2367
	N=5000	7.92	35	0	1910	2404
	N=10000	7.92	35	0	1929	2417
stringsearch Searches for given words with total length N	N=1k	7.12	2	0	644	1135
	N=4k	7.12	3	0	652	1147
	N=16k	7.12	9	0	660	1159
fft Performs a Fast Fourier Transform on an array of data with size N	N=1M	8.64	19	73	10057	17936
	N=4M	8.64	19	73	10103	17983
	N=16M	8.64	19	73	10149	18028
crc32 Performs a 32-bit Cyclic Redundancy Check on an array of data with size N	N=1M	5.62	1	0	104	191
	N=4M	5.62	1	0	108	197
	N=16M	5.62	1	0	112	204
adpcm Takes 16-bit linear PCM samples and converts them to 4-bit samples. The input samples are an array of data with size N	N=1M	8.47	4	0	809	2056
	N=4M	8.47	4	0	811	2062
	N=16M	8.47	4	0	813	2068
astar Finds a least-cost path from a given initial node to one goal node in the map with width and height N	N=20	8.53	15	28	3549	9432
	N=60	8.53	78	28	3620	9513
	N=100	8.53	295	30	3754	9516
susan Smooth an image and has adjustments for threshold, brightness, and spatial control. The image data is in an array with size N.	N=1k	8.44	300	98	14764	28029
	N=4k	8.44	300	98	14934	28427
	N=16k	8.44	300	98	15104	28825
ispell Spelling checker for English words in a word array with size N.	N=100	7.92	110	10	4598	8038
	N=1000	7.92	110	10	4610	8054
	N=10000	7.92	110	10	4635	8108
rijndael Uses AES to encrypt input stream in a word array with size N	N=1k	8.34	40	2	4434	13808
	N=16k	8.34	40	2	4462	13849
	N=256k	8.34	40	2	4490	13891
sha A hash algorithm that produces a 160-bit message digest for a given array with size N	N=16k	7.19	6	0	2815	5272
	N=128k	7.19	6	0	2824	5281
	N=1M	7.19	6	0	2833	5290
blowfish (a symmetric block cipher with a variable length key)	-	5.67	3	0	879	1526
basicmath (performs simple mathematical calculations)	-	8.64	38	199	24440	43978
bitcount (counts the number of bits in an array of integers)	-	8	7	0	468	1191

FPGA resources. It consumes 17 BRAM_18K to solve a single cosine or sine function. Moreover, we check the verilog codes generated by Vivado through HLS to find that cosine and sine trigonometric functions use CORDIC (for COordinate Rotation DIgital Computer) algorithm to implement, which was first described in 1959 by Jack E. Volder. CORDIC is a simple and efficient algorithm to calculate hyperbolic and trigonometric functions. It is commonly used in microcontrollers and FPGAs as the only operations it requires are addition, subtraction, bitshift and table lookup. [**?**] It is the table lookup employed by CORDIC in solving the cosine and sine functions that leads to the huge consume of RAM. Also, the frequent use of ddition, subtraction, bitshift operations results in more use of DSP, FF and LUT resources. Table V shows us that the exponent and logarithm operations also make an ignorable contribution to the cost of FF and LUT,. Because of the reasons above, the basicmath benchmark consumes the most FF and LUT resources.

VI. CONCLUSIONS

This paper presented an open-source C/C++ benchmark suite for FPGA high-level synthesis which is mainly targeted for designers wanting to evaluate different commercial HLS tools. These benchmarks are carefully selected from well-known CPU benchmarks to represent different application domains. This paper also described synthesis results based on commercial Xilinx Vivado. In addition, we presented some important code transforming strategy such as dynamically memory allocation support in HLS, which is very useful for FPGA HLS designers.

ACKNOWLEDGMENT

This research is funded by National Natural Science Foundation of China (61271149, 61106033).