Hosseinabady, M., & Nunez-Yanez, J. L. (2015). Optimised OpenCL workgroup synthesis for hybrid ARM-FPGA devices. In 2015 25th International Conference on Field Programmable Logic and Applications (FPL 2015): Proceedings of a meeting held 2-4 September 2015, London, United Kingdom. [7294016] (International Conference on Field-programmable Logic and Applications Proceedings). Institute of Electrical and Electronics Engineers (IEEE). 10.1109/FPL.2015.7294016

Optimised OpenCL Workgroup Synthesis for Hybrid ARM-FPGA Devices

Mohammad Hosseinabady and Jose Luis Nunez-Yanez
Department of Electrical and Electronic Engineering University of Bristol, UK.
Email: {m.hosseinabady, j.l.nunez-yanez}@bristol.ac.uk

***Abstract*—This paper presents a workgroup synthesis mechanism to compile an OpenCL kernel to FPGA-based accelerators embedded in a multi-core CPU system-on-a-chip (SoC). The OpenCL kernels considered in this paper exhibit regular data access patterns. Coping with the limited amount of internal memory in embedded FPGAs, the workgroup synthesis utilises a novel data access pattern formulation to describe the parallelism already provided by the OpenCL kernels. To provide an OpenCL framework prototype to validate the proposed technique, a source-to-source compiler that transforms the OpenCL kernel into C/C++ code is developed. Then vendor-specific high-level synthesis tools are used to convert the C/C++ code into the FPGA bitstream. Results based on popular real applications show up to 89.8% improvement in the execution time compared to the commercial FPGA OpenCL implementations.**

***Index Terms*—Hybrid ARM-FPGA Embedded System, ZYNQ, OpenCL, HLS, POCL, FPGA**

I. Introduction

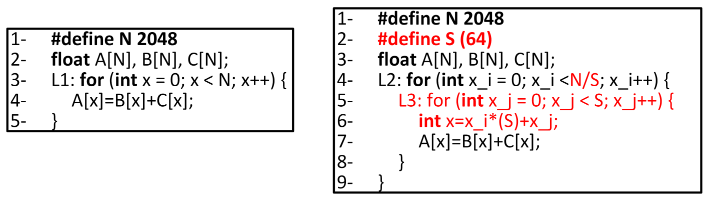
FPGAs (Field Programmable Gate Arrays) are an alternative high performance technology that offers bit-level parallel computing in contrast with the word-level parallelism deployed in GPUs and CPUs. Bit level parallel computing fits certain algorithms that cannot be parallelized easily with traditional methods. Recent developments have aimed to make the FPGAs more attractive to the software designers by integrating full-fledged processors couple to the FPGA logic focusing on the CPU part as the first component to boot and become available. Xilinx Zynq SoC [1] and Altera Cyclone V SoC [2] are typical examples of these hybrid platforms. These hybrid platforms provide great potentials for a hardware-software co-design environment, however the design of hardware and software requires two different schemes. Whereas hardware parts are usually described by using a hardware description language (HDL), software parts are expressed by C-like programming languages. Therefore, utilising the parallelism in the FPGA is not an easy task and requires a high-skill of HDL programming. High-Level Synthesis (HLS) tools which receive the design in a high-level language such as C/C++ try to alleviate this issue. Although an HLS makes the hardware design more feasible, it still requires a thorough knowledge of hardware design techniques. Recently, FPGA manufacturers have been pioneering OpenCL for FPGAs aiming to overcome their low-level programming models. This critical innovation could make FPGA devices accessible to software programmers as an additional computing resource without changes to the algorithm source code. A few commercial and research OpenCL frameworks have been proposed by industry and researchers [3–7]. The commercially available OpenCL frameworks recommend that designers should structure their OpenCL kernel as a single workitem kernel (instead of NDRange kernels) if it is possible to improve performance [8]. Using single workitem kernel helps the tools to effectively utilise the limited memory bandwidth between the FPGA and main memory, but it could limit code/performance portability of the OpenCL kernel. Therefore, more research is required to import the variety of parallelism features and models available in the OpenCL specification effectively into FPGAs. Utilising different levels of parallelisms provided by OpenCL applications in an FPGA-based OpenCL framework has motivated us to do this research. Note that efficiently utilising hardware resources (including logic and internal memories) in an FPGA for OpenCL applications is a challenge especially for small size FPGAs which is the case in the Xilinx Zynq SoC, the chosen hybrid SoC in this paper. To cope with this issue, we define data access patterns in OpenCL kernels including *intra*-and *inter-workitems* data access patterns. Using these patterns, this paper proposes an automated workgroup synthesis to group workitems so that their required data can fit in the FPGA internal memory and the parallelism in the hardware increases. Loop *unrolling* and *pipelining* are used to increase the parallelism in the body of loops, whereas *array partitioning* are used to increase the data access ports which in turns increases the data access parallelism. The proposed array partitioning provides different memory ports for adjacent workitems which allows the synthesis tool to effectively apply the loop pipelining. To evaluate the proposed workgroup synthesis, an OpenCL prototype framework is developed to process a large data set on a hybrid FPGA-ARM platform based on Xilinx Zynq SoC. The experimental results confirm the efficiency of the proposed technique by using a few well-known OpenCL benchmarks which show 89.8% speedup compared to commercial FPGA OpenCL implementations. The novelties of the proposed framework are:

- Generation of OpenCL kernel hardware and corresponding device drivers for a hybrid ARM-FPGA platform.
- Systematic formulation of inter-and intra-workitem data access patterns in an OpenCL kernel to be used for workgroup synthesis in order to cope with the limited memory resources in the FPGA.

The rest of this paper is organised as follows. The next section reviews some of the previous work. Section 3 documents motivations and contributions of the paper in more details. The proposed methodology is explained in Section 4. Section 5 describes the experimental results. Finally, Section 6 concludes the paper.

II. Previous work

Generating FPGA-based hardware components for the computation extensive parts in a parallel language such as OpenCL has attracted both academic and industrial research in the last few years. Altera Corporation has proposed an OpenCL SDK platform for OpenCL-to-FPGA compilation [9]. This platform can show compelling results for small size applications such as the shorter FFT lengths which are common in radar processing [5]. In contrast to this platform, we propose a workgroup synthesis techniques to efficiently map OpenCL applications that process large amount of data on a very resource-restricted FPGA in the Xilinx Zynq SoC. Owaida et al. [3] propose a source-to-source converter to generate a C code from the OpenCL kernel code. They utilise compiler techniques such as prediction, code slicing and modulo scheduling to convert the C code into an HDL code to be synthesised by synthesis tools considering a hardware architecture. The hardware architecture consists of a datapath and a streaming unit, resembling the VLIW-like architecture. Although, they have considered popular techniques to adapt the architecture for implementation on the FPGA, using a predefined architecture as a parallel hardware style restricts the resource exploitation and utilisation of intrinsic parallelism in an FPGA. Contrary to this work, we use vendor specific high-level synthesis tools to generate the final bitstream from the C++ code. An OpenCL to FPGA is presented in [4] in which the FPGA is connected to the host program (located on a desktop) via PCIe bus. The paper does not clearly explain the optimisation techniques nor gives the execution time for the benchmarks. In contrast, we focus on embedded systems with shared memory (no PCIe) and a specific optimisation technique to be



```
#define N 2048
float A[N], B[N], C[N];
L1: for (int x = 0; x < N; x++) {
    A[x]=B[x]+C[x];
}
```

(a) Single *for* loop

```
#define N 2048
#define S (64)
float A[N], B[N], C[N];
L2: for (int x_i = 0; x_i <N/S; x_i++) {
    L3: for (int x_j = 0; x_j < S; x_j++) {
        int x=x_i*(S)+x_j;
        A[x]=B[x]+C[x];
    }
}
```

(b) Nested *for* loop

Fig. 1: Simple Vector Adder

TABLE I: Results of synthesizing the simple vector addition on Xilinx Zynq

		Latency	Max Resource
Single loop	L1: unroll Array: register	4 (super-fast, infeasible)	1818% (DSP48E)
	L1:pipeline Array: BRAM	2008 (fast, low resource)	0% (DSP48E)
Nested Loop	L2: pipeline L3: unroll Array Partition	41 (very fast , feasible)	58% (DSP48E)

able to explain its efficiency by comparing with software techniques and the only commercial available OpenCL platform for FPGA during this research. Xilinx has recently introduced SDAccel [10] for data centre application acceleration leveraging FPGAs. The current version of this developing environment does not support Xilinx Zinq SoC platform.

III. Motivations and Contributions

Efficiently utilising different levels of parallelisms provided by OpenCL applications in FPGA-based OpenCL framework has motivated us to propose a technique to transform inter-and intra-workitem parallelisms described in the OpenCL kernels to the fine-grained parallelism provided by FPGAs. The key technique in implementing a kernel in an FPGA is utilising the intrinsic parallelism provided by the FPGA instead of artificially creating distinctive parallel processing elements (as it is done in GPUs) using the FPGA resources. For this purpose, a group of workitems is merged using C/C++ loops to be synthesized by HLS tool. Utilising loop unrolling and pipelining (the two most popular HLS tool parallelism techniques) along with array partitioning, the implemented kernel on FPGA can show a very low latency. To clarify this idea, lets consider a simple vector addition kernel that its corresponding FPGA code that merges the workitems with a *for* loop shown in Fig. 1(a). Table I contains the latency and maximum resource utilisation synthesizing this code using Xilinx Vivado HLS tools targeting Zynq SoC. The shortest latency is when all additions execute in parallel. For this reason, the L1 *for* loop is unrolled and all arrays are mapped to the registers to have their own access ports. The second row in Table 1 shows the result for this case which its latency is 4 clocks, however, the resource utilisation is very high which makes the implementation infeasible. If we apply the pipeline parallelism technique to the L1 *for* loop, the third rows in the table shows its latency which is 2008 clocks and its resource utilisation is very low.

If the for loop is transformed to two nested loops as shown in Fig. 1(b), and the inner loop is unrolled and the outer loop is pipelined, and arrays are partitioned with block size of 32, then the resulted latency (shown in the last row of Table 1) is 41 and the maximum resource utilisation is 58% which results in a very fast yet a feasible implementation. Note that, in order to transform a loop into a nested loop there should not be any dependency among the loop iterations which is the case for workitems (without barriers) in an OpenCL kernel. In addition, choosing the right size of iterations in the nested loops and array partitioning block size require the data access pattern workitems to be analysed.

Consistent with this idea, we will consider two guidelines in our OpenCL implementation on FPGAs:

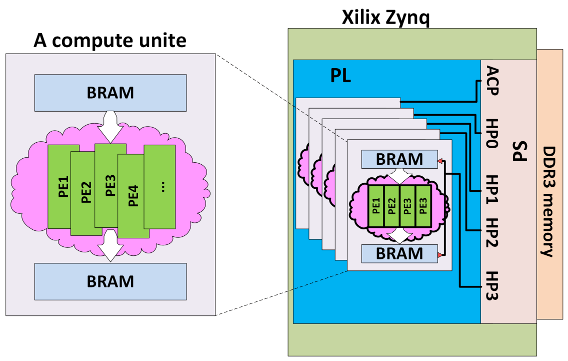


Fig. 2: Overview of the OpenCL device in the PL

- Utilising proper optimisation techniques (including pipelining, unrolling and memory partitioning) to the C/C++ function to be used by HLS.
- Transferring the required data from the DDR3 memory into the FPGA memory (e.g. BRAM) in a burst mode prior to executing the kernels

Note that we use Xilinx Zynq SoC as the FPGA-based platform in this paper. However, the proposed techniques are general enough to be applicable to similar platforms. The Xilinx Zynq consists of two main parts: Programmable Logic (PL) and Processing systems (PS). The PL is based on the Xilinx series 7 FPGA and PS contains dual core Cortex-A9 processor. In order to use FPGA BRAMs for data transfer between the host program and the kernel, we consider a general structure for the hardware in the PL as a compute device shown in Fig. 2. The compute device in the FPGA consists of maximum five compute units which are connected to four different HP ports and the ACP port on the PS side. Each compute unit runs a few workitems. First, the input data is transferred to the FPGA BRAM and then the workitems read their inputs from BRAM and write their results into it and finally the results are transferred to the main memory.

After considering this general structure, four main problems should be solved.

Problem 1- The corresponding C/C++ code for the workgroups should be generated to be synthesised by HLS. This code includes the directives for ports definitions and synthesis optimisations.
Problem 2- An OpenCL runtime system should manage the kernel execution.
Problem 3- The data required by workitems should be determined to be transferred to the BRAM prior to the workgroup execution.
Problem 4- Considering the BRAM size and kernel data size, the right number of workitems should be merged into a workgroup to be mapped on the FPGA. In addition proper optimisation directives should be determined automatically.

The main contribution of this paper is proposing a framework that consists of hardware and software components to cope with these problems. In addition, the specific novelty of this paper is proposing a new data access pattern formulation to cope with the last two problems. Analysing data access patterns, traditionally, is a technique in advanced compilers to automatically extract the parallelism or automatically optimise cache memory accesses by modifying the code. However, the innovation of the proposed technique is taking advantage of the parallelism already defined and provided by the OpenCL kernels in order to organise the input data efficiently to be used by a group of workitems in FPGAs.

IV. Proposed Framework

Fig. 3 shows the different components and design flow of the proposed framework. It consists of three parts: *platform*, *compiler* and *runtime*. A group of XML files as the platform database determines different parameters in the underlying hardware architecture such as the amount of BRAM available in the FPGA. The compiler part, which generates the bitstream for the FPGA, consists of a source-to-source code generator

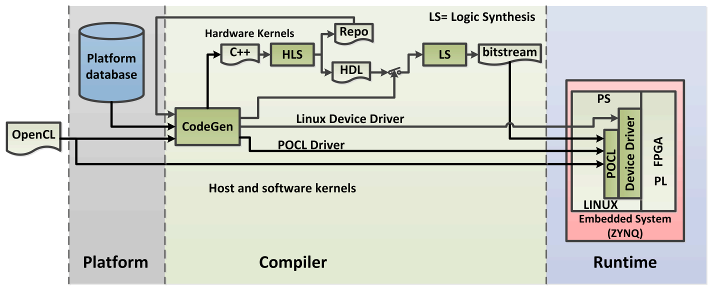


Fig. 3: Proposed framework and design flow

TABLE II: Analogy between OpenCL and FPGA implementations

OpenCL	FPGA
Global Memory	DDR3 memory
Local Memory	BRAM
Cash/Constant Memory	BRAM
Private Memory	Register or LUT
Local Synchronization via Barrier	Different serial function
Parallelism inter-and intra-workitems	PIPELINE and UNROLL
Parallelism inter workgroups	Different Compute units
Parallel data access intra-workgroup	Array Partitioning
Parallel data access inter-workgroup	Parallel memory access

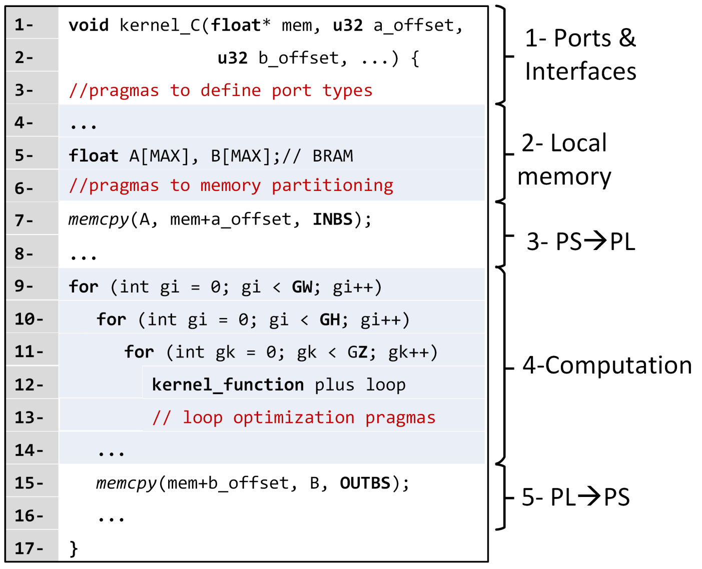
called **CodeGen** and HLS+LS as the synthesis toolset. The runtime consists of the hardware platform for this framework that is based on POCL [11] which is an open source OpenCL runtime, running on Linux operating system. The runtime system is responsible for FPGA reconfiguration and OpenCL execution. In the sequel, we explain different parts of this platform in more detail and how this framework addresses the four problems mentioned in Section III.

A. *Code Transformation: solution for the first problem*

To cope with the first problem, we have developed a source-to-source code generator (i.e., CodeGen in Fig. 3) using Clang [12], an open source compiler front-end tool, which supports OpenCL C extensions. For this purpose, we examine and change the Abstract Syntax Tree (AST) using features provided by Clang such as matchers [12] to get the data access information and modify the code. The CodeGen calls the HLS tool in a closed loop as shown in Fig. 3 to check the FPGA resource utilisation and guarantee the implementation of the kernel. The detail of this generator is explained in Subsection IV-C of this section. Table II shows some of the analogies that we have considered between entities in the OpenCL and FPGA.

Fig. 4 shows the basic code structure of the kernel to be synthesised by Vivado-HLS. This code consists of five main sections. The first section, in Lines 1-3, determines ports and interfaces for receiving and submitting data. The mem pointer in Line 1, which will be realised by AXI master interface, denotes the base address of the data in the main memory and *a_offset* and *b_offset* determine the offset addresses of the data that should be transferred between the PL and the PS. The second section defines variables in BRAM as the local memory to be used by workitems. The third section copes with transferring the data from the PS to the PL. In this section, the *memcpy* function (in Line 7) that has been used to transfer data between PL and PS will be synthesised to hardware that implements AXI burst data transfers by Vivado-HLS. The size of data transfer in bytes in each iteration is defined by the parameter INBS. The fourth section contains three nested loops to combine workitems in a rectangular cuboid in a 3D OpenCL NDRange.

The dimension sizes of the rectangular cuboid are determined by GW, GH and GZ parameters. For the sake of simplicity and brevity, this paper only explains the 2D NDrange. The body of these nested loops denoted by *kernel_function* merges workitems in a group. In this way, each compute unit runs a few workitems. However, the synthesis tool may combine these workitems with techniques such as pipelining and unrolling. Transforming a loop to nested loops to increase the



```
1-  void kernel_C(float* mem, u32 a_offset,
2-                   u32 b_offset, ...) {
3-  //pragmas to define port types
4-  ...
5-  float A[MAX], B[MAX];// BRAM
6-  //pragmas to memory partitioning
7-  memcpy(A, mem+a_offset, INBS);
8-  ...
9-  for (int gi = 0; gi < GW; gi++)
10-    for (int gi = 0; gi < GH; gi++)
11-      for (int gk = 0; gk < GZ; gk++)
12-        kernel_function plus loop
13-        // loop optimization pragmas
14-    ...
15-    memcpy(mem+b_offset, B, OUTBS);
16-    ...
17- }
```

Fig. 4: Basic structure of the OpenCL kernel in the PL

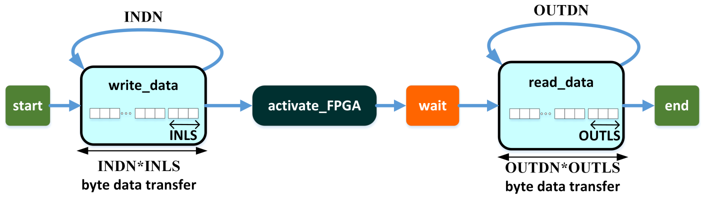


Fig. 5: OpenCL device driver and Linux device driver state machine

parallelisms is explained in Subsection IV-D of this section and is not shown in Fig. 4 for the sake of simplicity. The last section, transfers the data from BRAM to DDR3 memory and the parameter OUTBS determines the size of data transfer. The parameters in Fig. 4 should be defined through an optimisation technique which explained in the sequel.

B. *Runtime system: solution for the second problem*

To handle the second problem mentioned in the contribution subsection, the CodeGen algorithm also generates the Linux and POCL device drivers to be used by the POCL runtime system. The role of these drivers is to communicate with the compute device on the FPGA. Using a state machine, Fig. 5 shows the behaviour of these device drivers. Whereas the Linux device driver defines the states and their buffers in the Linux kernel space, the POCL device drivers determines and controls the flow among these states from the user space. Four main states are distinguished in this state machine: *write_data*, *activate_FPGA*, *wait*, and *read_data*. The *write_data* and *read_data* states provide two buffers in the kernel space. If the POCL driver wants to send data to the PL, it writes the data into the buffer calling the *write_data* state. If the data is in a continuous address space in memory then this state is activated only once. If the data are apart by a constant stride as it may be the case for array structures considered in this paper, then this state should be called more than once. The parameter INDN and INLS define the number of iterations and the size of data transfer in each iteration, respectively. Note that the size of the total data transferred to PL is equal to $INLS * INDN$ which should be equal to the INBS parameter in Line 6 of Fig. 4. After transferring the data to the buffer, the POCL driver calls the *activate_FPGA* state and then it goes to the wait state. Then the FPGA reads the data from the kernel space buffer, performs its task and puts the results into the read buffer. Finally, the POCL driver can read the data to the user space. The OUTDN and OUTLS parameters are the same as corresponding parameters in the *write_state*. The function *dma_alloc_coherent* from the Linux standard kernel has been used for memory allocation which provides continuous

```
1-  __kernel void image_filter_kernel(
2-            __global float*  InImage,
3-            __global float*  OutImage) {
4-     int i = get_global_id(0);
5-     int j = get_global_id(1);
6-     float sum = 0;
7-     for (int l = 0; l < m; l++)
8-        for (int k = 0; k < m; k++) {
9-           int xIn=i+l-m/2;  int yIn=j+k-m/2;
10-          ...
11-          sum+=Mask[l][k]*InImg[yIn*n+xIn];
12-       }
13-    OutImage[j*n+i]=sum;
14- }
```

Fig. 6: Image convolution OpenCL kernel

cache coherent memory especially for communication with HP ports which are not cache coherent like the ACP port.

C. Data access pattern: solution for the third problem

In order to tackle the third problem, the data access pattern for a workitem input or output array should be defined and explained. For this purpose, we assume the array indices to access data are affine functions [13] of NDRange global indices and loop indices inside the kernel. The affine functions can described many different commonly access patterns (such as tiling [13], row-/column-major access, stride, scaling, compression, shifting, permutation, reversal, skewing [14] and sliding window) in many applications from image and video processing domains exhibit these characteristics.

Data access pattern for an array variable describes the regularity in accessing the array elements in a part of a code. An *intra-workitem access pattern* for an array variable is a pattern that defines the data access within a workitem. An *inter-workitem access pattern* is a pattern by which workitems access a region of data. The kernel access pattern consists of intra- and inter-workitem access patterns. Taking the image convolution algorithm as a simple example, this subsection explains the underlying idea of formalising the kernel access pattern. The image convolution algorithm modifies image pixel values considering neighbouring pixels. A mask in this algorithm determines how to involve the neighbouring pixels to calculate each new pixel. A mask is an array of values with one element as an anchor. The algorithm places the anchor of the mask on an image pixel and adds the multiplication of each mask element by the image pixel that it overlays. Fig. 6 shows the OpenCL kernel for the image convolution algorithm. Note that this is an illustrative code and it is not optimised and complete. In this implementation, the NDRange is the index space of the output image of size $n \times n$. Therefore, there is a workitem corresponding to each pixel in the output image. We assume that the mask (of size $m \times m$) is defined in the kernel which has been removed from the code for the sake of brevity.

The indices in the image convolution kernel (Lines 9, 10 of Fig. 6) can be described by Equs. 1 and 2 which can also be merged into Equ. 3 which defines the kernel access pattern for accessing the input image.

$$xIn = i + l - m/2; l = 0 \to m; i = 0 \to n \tag{1}$$

$$yIn = j + k - m/2; k = 0 \to m; j = 0 \to n \tag{2}$$

$$\begin{pmatrix} xIn \\ yIn \end{pmatrix} = \overbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}}^{\substack{inter-workitem \\ access\,pattern}} + \overbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} l \\ k \end{pmatrix} + \underbrace{\begin{pmatrix} -m/2 \\ -m/2 \end{pmatrix}}_{offset}}^{\substack{intra-workitem \\ access\,pattern}} \tag{3}$$

$$l, k = 0 \to m; i, j = 0 \to n \tag{4}$$

In Equ. 3, $(xIn, yIn)^T$ vector is the indices of data in the input image, $(i, j)^T$ vector denotes the workitem index (or NDRange index) and $(l, k)^T$ indicates the data access index inside a workitem. In addition, the first term on the right-hand side of Equ. 3 which denotes the inter-workitem access pattern and the second one determines the intra-workitem access pattern. Note that, an access pattern, can be realised by nested *for* loops in the C code such as the nested loop in Lines 7 and 8 of Fig. 6 which implements the intra-workitem access pattern. The rectangle area of data that are accessed by an intra-workitem access pattern has a key role in data-transfer optimisation in mapping an OpenCL kernel on FPGA. The minimum size of this rectangle area is called pattern size. This size is determined by the range of $(l, k)^T$ indices which according to the Equs. 1 and 2, it is $m \times m$ for the image convolution application shown in Fig. 6. In addition, the minimum size of each dimension is defined the range of the corresponding index. The minimum size of each dimension in image convolution example is m.

A generalized form of the kernel data access pattern for a 2D array variable can be defined as Equs. 5-7.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \overbrace{\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}}^{\substack{inter-workitem \\ access\,pattern}} + \overbrace{\begin{pmatrix} e & f \\ g & h \end{pmatrix} \begin{pmatrix} l \\ k \end{pmatrix} + \underbrace{\begin{pmatrix} r \\ s \end{pmatrix}}_{offset}}^{\substack{intra-workitem \\ access\,pattern}} \tag{5}$$

$$i = 0 \to n; j = 0 \to m \tag{6}$$

$$l = 0 \to p; k = 0 \to q \tag{7}$$

This access pattern can easily be generalised to 3D NDRange by adding extra row and column to the inter- and intra-workitem matrices. The range of indices l and k define a rectangle area of the access pattern that is used by the corresponding workitem. According to the ranges in Equ.s 6-7 the size of intra-workitem pattern in the generalised pattern is show in Equ. 8 . These formulations solve the third problem. In the sequel, the solution for the last problem explains how to use these formulations to efficiently map an OpenCL kernel on the FPGA.

$$pattern\_size = range(e.l + f.k + r)range(g.l + h.k + s) \tag{8}$$

where

$$range(x) = max(x) - min(x) \tag{9}$$

$$0 \le l < p, 0 \le k < q \tag{10}$$

D. Optimisation: solution for the fourth problem

To solve the last problem, we have to create large workitems by merging a few workitems together such that the FPGA can buffer the data accessed by the newly merged workitems. For this purpose, we utilise the workgroup concept in OpenCL. Considering the workgroup concept, a workitem in an NDRange can be defined by two groups of indices, the first one (called *group index*) determines the workgroup in the index space and another (called *local index*) defines the offset of the workitem in the corresponding workgroup. Combining group and local indices results in the global index which denotes the position of workitems in the entire workspace. If global, group, and local indices are denoted by $(i, j)^T$, $(gi, gj)^T$ and $(li, lj)^T$, respectively, then Equ. 11 shows the relation among them [6].

$$\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} gw & 0 \\ 0 & gh \end{pmatrix} \begin{pmatrix} gi \\ gj \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} li \\ lj \end{pmatrix} \tag{11}$$

$$0 \le li < gw, 0 \le lj < gh \tag{12}$$

in which gw and gh are the group width and group height, respectively. By substituting $(i, j)^T$ in Equ. 5 with Equ. 11, Equ. 13 will be obtained which in turn can be written as Equ. 14.

$$\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}a & b\\c & d\end{pmatrix}\begin{pmatrix}gw & 0\\0 & gh\end{pmatrix}\begin{pmatrix}gi\\gj\end{pmatrix}+ \\ \begin{pmatrix}a & b\\c & d\end{pmatrix}\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}\begin{pmatrix}li\\lj\end{pmatrix}+\begin{pmatrix}e & f\\g & h\end{pmatrix}\begin{pmatrix}l\\k\end{pmatrix}+\begin{pmatrix}r\\s\end{pmatrix} \tag{13}$$

$$\begin{pmatrix}x\\y\end{pmatrix}=\overbrace{\begin{pmatrix}a.gw & b.gh\\c.gw & d.gh\end{pmatrix}\begin{pmatrix}gi\\gj\end{pmatrix}}^{\substack{inter-workitem\\ access\,pattern}}+ \\ \overbrace{\begin{pmatrix}a & b\\c & d\end{pmatrix}\begin{pmatrix}li\\lj\end{pmatrix}+\begin{pmatrix}e & f\\g & h\end{pmatrix}\begin{pmatrix}l\\k\end{pmatrix}+\begin{pmatrix}r\\s\end{pmatrix}}^{\substack{intra-workitem\\ access\,pattern}} \tag{14}$$

Considering the merged workitems as a large new workitem, the first term in the right hand side of Equ. 14 denotes the inter-workitem access pattern and the rest of the terms in the right hand side represents the intra-workitem access pattern. Note that, this new intra-workitem access pattern can be realised by four nested for loops. The width and hight of the data region determined by the access pattern are as Equs. 15 and 16. These denote the area of data that should be transferred to the FPGA. Similar to Equ. 8, the size of workitem access pattern is equal to Equ. 17.

$$GW = range(a.li + b.lj + e.l + f.k + r) \tag{15}$$

$$GH = range(c.li + d.lj + g.l + h.k + s) \tag{16}$$

$$pattern\_size = GW \times GH \tag{17}$$

$$0 \leq li < gw; 0 \leq lj < gh \tag{18}$$

$$0 \leq l < p; 0 \leq k < q \tag{19}$$

We follow a huristic scheme to apply the optimization techniques to the code. This huristic transforms the inner loop encompassing a workitem into two nested loops of size S (similar to the code in Fig. 1(b)) and applies the unrolling optimisation pragma to the inner loop and the pipeliming to the outer loop. An initial value of S denoted by S_{init} is defined by the maximum size of data transferred to the FPGA for the first dimension in the OpenCL index space. The final value of S is determined by an iterative algorithm in which S is divided by 2 in each iteration. As each memory partition provides two ports in the FPGA, the size of the memory partition is defined by $S/2$. Newly merged workitems are divided among available compute units. The workitems assigned to a compute-unit run sequentially on that. As transferring data from the main memory to FPGAs BRAM is time consuming, a streaming technique in which data is received, processed and submitted in a pipeline fashion is desirable. In addition, the burst data transmission should be used for receiving and submitting the data used in each pipeline. Therefore, if the rectangle region of data required/produced by the generated workgroup are in a continuous memory area, it can be transferred only by one iteration of burst data transfer. This means that INDN and OUTDN parameters in Fig. 5 are equal to one. However, if the rows in the rectangle region are separated by a constant stride, then each row should be transferred separately. In this case, the number of rows in the group denoted by GH (i.e., Equ. 15) defines INBN (in Fig. 5) for input data and OUTBN (in Fig. 4) for output data. In this case, the size of burst (i.e., INLS or OUTLS in Fig. 5) is the size of one row in the region which is denoted by GW (i.e., Equ. 16). In order to reduce the runtime overhead, GW should be maximized. Therefore, Equs. 20-23 show the optimization problem to find the best group size. Equs. 20 determines that the sum of the data for all array inputs and outputs should be less than the total BRAM size in the FPGA. In this inequality the coefficient n denotes the number of compute units available, $sizeof(type)$ represents the number of bytes required for saving a data point (for example it is four for float type) $|BRAM|$ is the total size of BRAMs in the FPGA in bytes, and α is the synthesis tool efficiency factor to utilise the BRAM. This factor is empirically can be derived by running a few examples and get an average of the amount of data in the design divided by the

Algorithm 1: OpenCL kernel to C++ CodeGen Algorithm

```
   Data: OpenCL kernel
1  extract array indices in the OpenCL kernel ;
2  $success\_flag = false$
3  $n = 5$ //the number of initial compute units ;
4  while $n > 0$ do
5      Solve the optimization problem in Equs. 20-23 ;
6      $S_{init}$ = size of the first NDRange dimension transferred to FPGA;
7      $S = S_{init}$;
8      Array partition factor = $S/2$
9      Code[n] = Generate the code using CodeGen ;
10     while $S > 1$ do
11         Call Vivado-HLS;
12         if design fits in the PL then $success\_flag = true$; break;
13         else
14             $S = S/2$
15         end
16     end
17     if $success\_flag == true$ then break;
18     $n = n - 1$
19 end
20 if $success\_flag == true$ then Save the results as a possible solution
   and generate the device drivers;
21 return $success\_flag$;
```

amount of used BRAM. It is 0.7 for the Zynq board and Vivado-HLS. Equ. 23 defines the objective function.

$$n \sum sizeof(type) \times GW_i \times GH_i < \alpha|BRAM| \tag{20}$$

$$0 \leq li < gw; 0 \leq lj < gh \tag{21}$$

$$0 \leq l < p, 0 \leq k < q \tag{22}$$

$$maximise \quad GW \tag{23}$$

The next subsection explains how to use this optimization problem to find the parameter in the generated codes considering the synthesis tool.

Putting together all the ideas and techniques explained in this paper, Algorithm 1 shows the pseudo code for the CodeGen tool which generates the C++ code for Vivado HLS and device drivers. Applying Algorithm 1 to the image convolution algorithm explained in Subsection C with a mask of size 7×7 and image of size 1902×1080, we have

$$5 \times [\overbrace{4}^{\substack{datatype\\ float}} \times \overbrace{(gh+7)(gw+7)}^{\substack{inputimage\\ withpadding}} + \overbrace{4}^{\substack{datatype\\ float}} \times \overbrace{gh \times gw}^{outputimage}] \tag{24}$$

$$1 \leq gh < 1080; 1 \leq gw < 1920 \tag{25}$$

By solving this equations $gw = 1920$ and $gh = 1$ and initializing the S to 1920, the final S is $1920/4 = 480$ and array partitioning is 2. The latency for one compute unit without inner loop transformation (i.e., $S = 1$) is 32741 clocks and with inner loop transformation (i.e., $S = 2$) is 26078 which shows 20.35% speedup. The next section reports more results obtained by using the proposed techniques.

V. Experimental Results

To evaluate the proposed framework, we have used five benchmarks including matrix multiplication (*mm*), image convolution (*ic*), nbody problem, and black scholes option pricing [15] (*bs*). The image convolution processes an HD image of size 1920×1080 with a 7×7 mask and its kernel code contains two nested loops with constant bound of 7. The matrix multiplication multiplies two large floating point matrices of size 2048×2048 and its kernel code contains a loop. The nbody problem is applied to 2048 particles and its kernel contains a loop. The black scholes option pricing processes 4000000 options and its kernel does not have any loop. All benchmarks, except *nbody* receive input data larger than the size of BRAM in the Zynq which is $560KB$. Simple OpenCL codes without defining workgroups are developed for these benchmarks allowing the algorithm to find the proper workgroups. All benchmarks

TABLE III: Kernel Execution time and runtime overhead

	ic	mm	nbody	bs
Total kernel execution time (msec)	96.71	5015.51	55.99	278.2
Drivers runtime overhead (msec)	8.9	424.5	5.2	18.3
Overhead	9.2%	8.5%	9.29%	6.58%

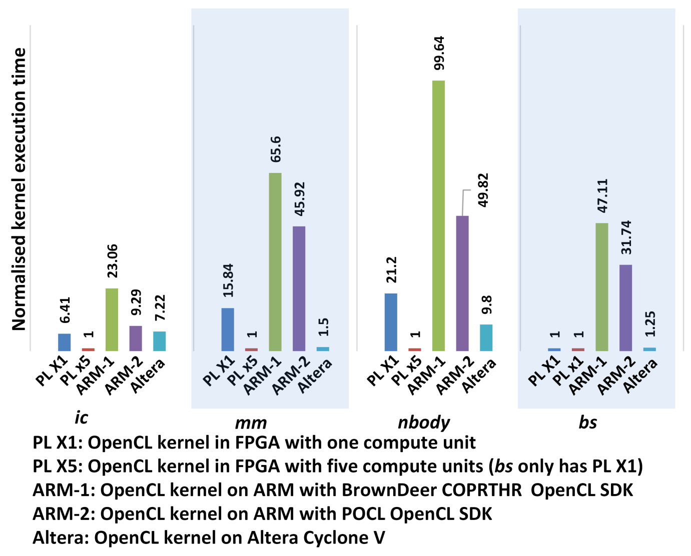


Fig. 7: Normalised kernel execution time comparison

apart from the bs use five compute units. The *bs* benchmark only uses one compute unit due to its high resource utilisation.

Table III contains the total kernel execution time and the drivers runtime timing overhead in $msec$ for different benchmarks. As can be seen, the drivers overhead is less than 10% of the total kernel execution time. The AXI port memory bandwidth during data transfer phase in these executions for each port is about $395.25MB/s$ that saturates the bandwidth provided by an AXI port which theoretically is about $400MB/s$ with $100MHz$ clock frequency.

Fig. 7 compares the execution time of OpenCL kernels of the benchmarks running on different platforms and configurations. The execution times are normalised by the best implementation on Zynq which is denoted by PL X5. Note that Table 3 contains the real execution time for PL X5, therefore the execution time for other implementations can easy be obtained by simple multiplication. The PL X1 and PL X5 which are based on the proposed technique use one and five compute units in the Zynq FPGA, respectively. ARM-1 and ARM-2 are using the ARM processor in the Zynq using COPRTHR [16] and POCL, the two open source OpenCL implementations, respectively. The POCL shows a better performance as it can partially utilise the vectorization exist in the code using the compiler. Altera shows the execution time for running the OpenCL kernel on Cyclone V. As can be seen, the proposed framework shows lower execution time than that of the other implementations. Table IV compares the resource utilisation for the two FPGA implementations. Two main features make the proposed technique implemented in Zynq more efficient compared to the Altera environment: utilising more DSP resource as shown in Table 4 and utilising more ports (maximum five ports) between FPGA and DDR memory. The maximum improvement of the proposed technique compared to the Altera is $100 * (9.8 - 1)/9.8 = 89.8\%$ in *nbody* benchmark.

VI. Conclusion

This paper proposed an automated technique to map on OpenCL description efficiently on the Xilinx Zynq SoC as a hybrid ARM-FPGA embedded system. As the FPGA does not have enough space to save a large data set, the paper proposes a workgroup synthesis technique to automatically divide a large data set into small blocks and map the OpenCL description on FPGA. The propose FPGA implementation for

TABLE IV: FPGA resource utilization, Altera Cyclone V vs Xilinx Zynq

		ic	mm	nbody	bs
Altera	Freq. MHz	135.81	144.88	124.0	141.16
	Memory bits	872,416 (21%)	420,544 (10%)	644,808 (16%)	649,248 (16%)
	DSP	9 (10%)	1 (1%)	24 (28%)	83 (95%)
	Logic	10,290 (32%)	5,849 (18%)	13,568 (42%)	15,802 (49%)
xilinx	Freq. MHz	100	100	100	100
	Memory bits	3,240,000 (64%)	4,392,000 (87%)	2,880,000 (57%)	2,930,526 (58%)
	DSP	50 (22%)	20 (9%)	40 (18%)	204 (92%)
	LUT as logic	16,263 (31%)	8,266 (15%)	18,677 (35%)	22,415 (42%)

OpenCL applications shows up to 89.8% speed-up in compare to the Altera OpenCL running on Cyclone V.

Acknowledgement

The authors would like to thank the reviewers for their valuable comments. This research is a part of the ENPOWER project sponsored by EPSRC.

References

[1] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book Tutorials*, Department of Electronic and Electrical Engineering University of Strathclyde Glasgow, Scotland, UK, 2014.
[2] Altera, “Cyclone v device handbook volume 1: Device interfaces and integration,” Altera Corporation, Tech. Rep. CV-5V2 2015.01.23, 2015.
[3] M. Owaida, N. Bellas, C. D. Antonopoulos, K. Daloukas, and C. Antoniadis, “Massively parallel programming models used as hardware description languages: The OpenCL case,” in *Proceedings of the International Conference on Computer-Aided Design (ICCAD’11)*, 2011, pp. 326–333.
[4] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh, “From OpenCL to high-performance hardware on fpgas,” in *Proceedings of 22nd International Conference on Field Programmable Logic and Applications (FPL’12)*, 2012, pp. 531–534.
[5] Altera, “Radar processing: FPGAs or GPUs?” Altera Corporation, Tech. Rep. WP-01197-2.0, 2013.
[6] Xilinx, “The OpenCL specification,” Khronos OpenCL Working Group, Tech. Rep. Version 2.0, 2014.
[7] M. Lin, I. A. Lebedev, and J. Wawrzynek, “Openrcl: low-power high-performance computing with reconfigurable devices,” in *Proceedings of Field Programmable Logic and Applications (FPL)*, 2010, pp. 458–463.
[8] Altera, “Altera SDK for OpenCL best practices guide,” Altera Corporation, Tech. Rep. 2014.12.15, 2014.
[9] ——, “OpenCL for Altera FPGAs: Accelerating performance and design productivity,” Altera Corporation, Tech. Rep., 2012. [Online]. Available: http://www.altera.com/products/software/opencl/opencl-index.html
[10] Xilinx. (2015) SDAccel development environment. [Online]. Available: http://www.xilinx.com/products/design-tools/sdx/sdaccel.html
[11] P. Jskelinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, “pocl: A performance-portable OpenCL implementation,” in *International Journal of Parallel Programming*, August 2014.
[12] Xilinx. (2015) The llvm compiler infrastructure. [Online]. Available: http://llvm.org/
[13] J. M. Anderson and M. S. Lam, “Global optimizations for parallelism and locality on scalable parallel machines,” in *Proceedings of SIGPLAN 93 Conference on Programming Language Design and Implementation (PLDI)*, 1993, p. 112125.
[14] M. F. P. O’boyle and P. Knijnenburg, “Non-singular data transformations: definition, validity and applications,” *International Journal of Parallel Programming*, vol. 27, no. 3, pp. 131–159, 1999.
[15] NVIDIA OpenCL SDK Code Samples. (2015) OpenCL Black-Scholes option pricing. [Online]. Available: https://developer.nvidia.com/opencl
[16] T. Brown Deer. (2015) CO-PRocessing THReads (COPRTHR) SDK. [Online]. Available: http://www.browndeertechnology.com/coprthr.htm