
Peer reviewed version

License (if available):
Other

Link to published version (if available): 10.1109/FPL.2015.7294016

Link to publication record in Explore Bristol Research

PDF-document

© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

University of Bristol - Explore Bristol Research

General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available: http://www.bristol.ac.uk/pure/about/ebr-terms
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 coupled 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-workitem 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 intra- and inter-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 technique 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
TABLE I: Results of synthesizing the simple vector addition on Xilinx Zynq

<table>
<thead>
<tr>
<th>Latency</th>
<th>Max Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single loop</td>
<td>4 (super-fast, infeasible)</td>
</tr>
<tr>
<td>L1: unroll</td>
<td>2008 (fast, low resource)</td>
</tr>
<tr>
<td>L1: pipeline</td>
<td>41 (very fast, feasible)</td>
</tr>
<tr>
<td>Nested Loop</td>
<td>41 (very fast, feasible)</td>
</tr>
</tbody>
</table>

able to explain its efficiency by comparing with software techniques and the only commercially 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 Zynq 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 parallelisms 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 synthesised 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, let’s consider a simple vector addition kernel that 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.

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.
calledCodeGen 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. 3: Proposed framework and design flow](image)

**TABLE II: Analogy between OpenCL and FPGA implementations**

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Memory</td>
<td>DDR3 memory</td>
</tr>
<tr>
<td>Local Memory</td>
<td>BRAM</td>
</tr>
<tr>
<td>Cash/Constant Memory</td>
<td>BRAM</td>
</tr>
<tr>
<td>Private Memory</td>
<td>Register or LUT</td>
</tr>
<tr>
<td>Local Synchronization via barrier</td>
<td>Different serial function</td>
</tr>
<tr>
<td>Parallelism inter-and intra-workitems</td>
<td>PIPELINE and UNROLL</td>
</tr>
<tr>
<td>Parallelism inter workgroups</td>
<td>Different Compute units</td>
</tr>
<tr>
<td>Parallel data access intra-workgroup</td>
<td>Array Partitioning</td>
</tr>
<tr>
<td>Parallel data access inter-workgroup</td>
<td>Parallel memory access</td>
</tr>
</tbody>
</table>

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

```c
1- void kernel_c(float* mem, u32 a_offset, u32 b_offset, ...) {
2- // pragmas to define port types
3- ...
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. 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. 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.
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 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 in the workitem which has been removed from the code for the sake of underlying idea of formalising the kernel access pattern. The image convolution algorithm modifies image pixel values considering neighbouring pixels. A mask is an array which defines the kernel access pattern for accessing the input image. An access pattern, can be realised by 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)\textsuperscript{T} indices which according to the Equs. 1 and 2, it is m × 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.

\[
\begin{pmatrix}
  x \\
  y
\end{pmatrix} =
\begin{pmatrix}
  a & b \\
  c & d
\end{pmatrix}
\begin{pmatrix}
  i \\
  j
\end{pmatrix} +
\begin{pmatrix}
  e & f \\
  g & h
\end{pmatrix}
\begin{pmatrix}
  l \\
  k
\end{pmatrix} +
\begin{pmatrix}
  r \\
  s
\end{pmatrix}
\]

\[
i = 0 \rightarrow n; j = 0 \rightarrow m
\]

\[
l = 0 \rightarrow p; k = 0 \rightarrow q
\]

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 Equs. 6-7 the size of intra-workitem pattern in the generalised pattern is shown in Eq. 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.

\[
pattern_{size} = range(e,l + f,k + r)range(g,l + h,k + s)
\]

\[
range(x) = max(x) - \min(x)
\]

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 defines the position of workitems in the entire workspace. If global, group, and local indices are denoted by (i, j)\textsuperscript{T}, (gi, gj)\textsuperscript{T} and (li, lj)\textsuperscript{T}, respectively, then Eqn. 11 shows the relation among them [6].

\[
\begin{pmatrix}
  i \\
  j
\end{pmatrix} =
\begin{pmatrix}
  gw & 0 \\
  0 & gh
\end{pmatrix}
\begin{pmatrix}
  gi \\
  gj
\end{pmatrix} +
\begin{pmatrix}
  1 & 0 \\
  0 & 1
\end{pmatrix}
\begin{pmatrix}
  li \\
  lj
\end{pmatrix}
\]

\[
0 \leq li < gw, 0 \leq lj < gh
\]

in which gw and gh are the group width and group height, respectively. By substituting (i, j)\textsuperscript{T} in Eqn. 5 with Eqn. 11, Eqn. 13 will be obtained which in turn can be written as Eqn. 14.
This factor is empirically can be derived by running a few examples for float type.

The number of bytes required for saving a data point (for example it is
denotes the number of compute units available,
sizeof denotes the area of data that should be transferred to the FPGA. In this inequality the coefficient \( n \)
overhead, GW should be maximized. Therefore, Equs. 20-23 show the

These denote the area of data that should be transferred to the FPGA. Similarly to Eq. 8, the size of workitem access pattern is equal to Eq. 17.

\[
GW = range(a.l_i + b.l_j + c.l + f.k + r)
\]

\[
GH = range(a.l_i + d.l_j + g.l + h.k + s)
\]

\[
\text{pattern_size} = GW \times GH
\]

\[
0 \leq li < gw; 0 \leq lj < gh
\]

\[
0 \leq l < p; 0 \leq k < q
\]

We follow a heuristic scheme to apply the optimization techniques to the
code. This heuristic 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 pragramma to the inner loop and the piping to the outer loop. An initial value of \( S \) denoted by \( S_{\text{init}} \)
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., Eq. 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., Eq. 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, \( \text{sizeof}(\text{type}) \) represents the number of bytes required for saving a data point (for example it is four for float type) \( |\text{BRAM}| \) is the total size of BRAMs in the FPGA in bytes, and \( \alpha \) is the synthesis tool efficiency factor to utilise the BRAM. This factor is empirically can be derived by running a few examples and getting an average of the amount of data in the design divided by the

\[
\begin{align*}
\begin{bmatrix}
 x \\
 y \\
\end{bmatrix} &= \begin{bmatrix}
 a & b \\
 c & d
\end{bmatrix} \begin{bmatrix}
 gw & 0 \\
 0 & gh
\end{bmatrix} \begin{bmatrix}
 gi \\
 gj
\end{bmatrix} + \\
\begin{bmatrix}
 a & b \\
 c & d
\end{bmatrix} \begin{bmatrix}
 l_i & 0 \\
 0 & l_j
\end{bmatrix} + e \begin{bmatrix}
 f \\
 g
\end{bmatrix} \begin{bmatrix}
 l & 0 \\
 0 & k
\end{bmatrix} + r
\end{align*}
\]

Considering the merged workitems as a large new workitem, the first
term in the right hand side of Eq. 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 height 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.

\[
\begin{align*}
\begin{bmatrix}
 x \\
 y \\
\end{bmatrix} &= \begin{bmatrix}
 a & gw \\
 c & gw
\end{bmatrix} \begin{bmatrix}
 gi \\
 gj
\end{bmatrix} + \\
\begin{bmatrix}
 a & b \\
 c & d
\end{bmatrix} \begin{bmatrix}
 l_i & 0 \\
 0 & l_j
\end{bmatrix} + e \begin{bmatrix}
 f \\
 g
\end{bmatrix} \begin{bmatrix}
 l & 0 \\
 0 & k
\end{bmatrix} + r
\end{align*}
\]

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. \( S_{\text{init}} = \) size of the first NDRange dimension transferred to FPGA;
7. \( S = S_{\text{init}}; \)
8. Array partition factor = \( S/2 \)
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;

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 \( 7 \times 7 \) and image of size \( 1902 \times 1080 \), we have

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

amount of used BRAM. It is 0.7 for the Zynq board and Vivado-HLS.

\[
\begin{align*}
\text{Equ. 23 defines the objective function.} \\
\sum n \text{sizeof}(\text{type}) \times GW_i \times GH_i < \alpha |\text{BRAM}| \\
0 \leq li < gw; 0 \leq lj < gh \\
0 \leq l < p; 0 \leq k < q \\
\text{maximise} \ GW
\end{align*}
\]

\[
\begin{align*}
5 \times [ & \frac{4}{4} \times (gh + 7)(gw + 7) + \\
& \frac{4}{4} \times gh \times gw ] \\
1 \leq gh < 1080; 1 \leq gw < 1920
\end{align*}
\]

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), nboby problem, and black scholes option pricing [15] (bs). The image convolution processes an HD image of size \( 1920 \times 1080 \) with a \( 7 \times 7 \) 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 \( 2048 \times 2048 \) and its kernel code contains a loop. The nboby 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 nboby 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

<table>
<thead>
<tr>
<th></th>
<th>ic</th>
<th>mm</th>
<th>nbody</th>
<th>bs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total kernel execution time (msec)</td>
<td>96.71</td>
<td>5015.51</td>
<td>55.99</td>
<td>278.2</td>
</tr>
<tr>
<td>Drivers runtime overhead (msec)</td>
<td>8.9</td>
<td>424.5</td>
<td>5.2</td>
<td>18.3</td>
</tr>
<tr>
<td>Overhead</td>
<td>9.2%</td>
<td>8.5%</td>
<td>9.29%</td>
<td>6.58%</td>
</tr>
</tbody>
</table>

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 305.25 MB/s that saturates the bandwidth provided by an AXI port which theoretically is about 400 MB/s with 100 MHz clock frequency.

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

<table>
<thead>
<tr>
<th></th>
<th>ic</th>
<th>mm</th>
<th>nbody</th>
<th>bs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq. MHz</td>
<td>135.81</td>
<td>144.88</td>
<td>124.0</td>
<td>141.16</td>
</tr>
<tr>
<td>Memory bits</td>
<td>872.416</td>
<td>420.544</td>
<td>644.808</td>
<td>649.248</td>
</tr>
<tr>
<td>Logic</td>
<td>10,290</td>
<td>5,849</td>
<td>13,568</td>
<td>15,802</td>
</tr>
<tr>
<td>LUT</td>
<td>16,263</td>
<td>8,266</td>
<td>18,677</td>
<td>22,415</td>
</tr>
</tbody>
</table>

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

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

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