Automatic data placement for CPU-FPGA heterogeneous multiprocessor System-on-Chips

Shiqing Li, Yixun Wei, Lei Ju*
School of Software, Shandong University, Jinan, China

***Abstract*—Efficient utilization of restrained memory resources is of paramount importance in CPU-FPGA heterogeneous multiprocessor system-on-chip (HMPSoC) based system design for memory-intensive applications. State-of-the-art high level synthesis (HLS) tools rely on the system programmers to manually determine the data placement within the complex memory hierarchy. In this paper, we propose an automatic data placement framework which can be seamlessly integrated with the commercial Vivado HLS. We first show counter-intuitive results that traditional frequency and locality based data placement strategy designed for CPU architecture leads to non-optimal system performance in CPU-FPGA HMPSoCs. Built on top of our memory latency analysis model, the proposed integer linear programming (ILP) based framework determines whether each array object should be access via the on-chip BRAM, shared CPU L2-cache, or DDR memory directly. Experimental results on the Zedboard platform show an average 1.39X performance speedup compared with a greedy-based allocation strategy.**

***Index Terms*—Data placement, memory architecture, FPGA, heterogeneous multiprocess system-on-chip, high level synthesis**

I. Introduction

Field Programmable Gate Array (FPGA) becomes an increasingly popular design choice for computer systems ranging from low power embedded systems to high performance computing architectures. Traditional FPGA design with register transfer level (RTL) programming requires considerable architectural- and circuit-level experiences, which is error-prone and time-consuming. High level synthesis (HLS) compiles a C/C++ kernel into the corresponding hardware description language (HDL) module. In recent years, HLS tools gain widespread use in the design of complex FPGA heterogeneous systems, which shortens time-to-market and reduces the system design complexity [1].

HLS tools allow the generated HDL code to be optimized for various design objectives including performance, energy efficiency, or hardware cost [1] [2]. Both commercial HLS tools (e.g., Vivado HLS [3]) and open-source tools (e.g., LegUp [4]) provide built-in directives for the selection of optimization techniques such as loop unrolling, pipelining, and array partitioning. Recent work propose heuristic methods to enable rapid design space exploration in HLS to find near-optimal optimization configurations of a given kernel [5]–[7]. However, these work target on the management of the logic resources such as look-up tables (LUTs) and flip-flops (FFs). On the other hand, memory management, especially for the restrained on-chip memory resources, becomes increasingly important for memory-intensive applications including DNN and video processing.

* Corresponding author. Email: julei@sdu.edu.cn

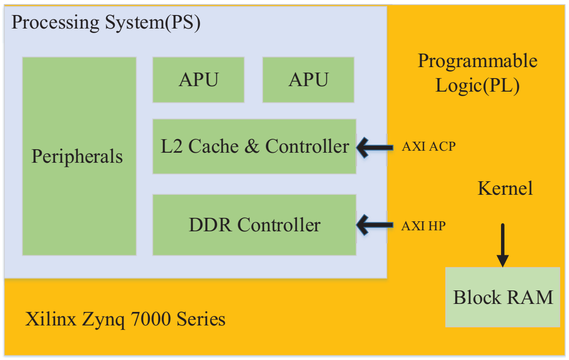


Fig. 1. Xilinx Zynq-7000 architectural overview

In HLS tools, the default allocation for arrays is BRAM. However, when the arrays' capacity exceeds the BRAM's capacity, the programmer needs to modify source code manually. Meanwhile, the contemporary CPU-FPGA heterogeneous SoC architecture usually employs a complex memory hierarchy in order to cope with the diversified memory access behavior of applications. Fig. 1 shows the architectural overview of Xilinx Zynq-7000 All Programmable SoC [8]. Each FPGA kernel running on the Programming Logic (PL) part is capable to visit data from the FPGA on-chip BRAMs, the off-chip DDR memory via accelerator coherency port (ACP) and CPU's shared level-2 cache (refer as *ACP* in this paper), or the off-chip DDR memory directly via high performance port (HP) which bypasses the CPU caches (refer as *HP* in this paper). Therefore, a manual data placement decision made by the system programmer may lead to non-optimal system performance due to the complex design space.

In this work, we propose an automatic data placement framework for CPU-FPGA HMPSoCs. The proposed framework comprehensively captures various factors that have impact on the memory subsystem performance. including characteristics of FPGA computation, memory access pattern, and architectural properties. The proposed integer linear programming (ILP) formulation generates an optimal data placement scheme, which can be directly seamlessly integrated into the Vivado HLS environment. The novel contributions of this paper are as follows.

- We perform a systematic study on the memory subsystem performance of CPU-FPGA HMPSoCs, which provides insight understanding of the memory hierarchy and guidelines for the data placement strategy.

- We show counter-intuitive examples that traditional frequency and locality-based data placement techniques designed for the software-controlled on-chip memories in traditional CPU (scratchpad memories, SPMs) or GPU (shared memory) result in non-optimal system performance.
- We formulate the data placement problem into an integer linear programming (ILP) formulation, which automatically decide the allocation of array objects. The output data allocation scheme can be directly used as configuration options in the Vivado Design Suite.
- We evaluate the proposed framework on the Zedboard platform with Xilinx Zynq-7020 HMPSoC, which achieves an average 1.39X performance speedup compared with a greedy-based allocation strategy.

II. Related work

A. FPGA BRAM management

In [9], Cong et al. propose various automated BRAM buffer restructuring approaches and analytical model, which effectively improves the BRAM utilization and system performance. In [10], the *line buffer* data structure has been refined, which allows better unitization of on-chip memory and achieves 100fps when running a Binary Neural Network(BNN). Chen et al. propose parallel data channels implemented via BRAM to accelerate the bitonic sorting [11]. In [12], the authors introduce the DMM-HLS framework that extends conventional HLS with dynamic memory allocation/deallocation mechanisms to be incorporated during many-accelerator synthesis. Besides the typical usage of BRAM as a software-control memory space, literature work propose to organize BRAM as hardware-controlled caches via a custom BRAM controller to reduce the complexity of BRAM management for the software programmers [13], [14]. However, automatic BRAM resource management and data placement in HLS have not been addressed in the literature work.

B. Data allocation

Data placement techniques have been well-studied in the context of CPU architecture with scratchpad memories (SPMs), which is a software-controlled on-chip memory space similarly to BRAMs in the FPGA fabric. Suhendra et al. [15] propose an SPM allocation strategy for CPU multiprocessor system-on-chip, where frequently accessed data are placed to the on-chip SPM space for a better system performance. Lu et al. study the SPM management for stack data at function granularity [16].

For a hybrid SPM-cache on-chip memory organization, Verma et al. [17] propose a cache-aware scratchpad allocation algorithm. In addition to data accesses frequency, cache interferences are considered as another factor in making the allocation decision. Chang et al. propose contention-aware SPM allocation algorithms to manage the hybrid on-chip memory at run-time, where data incur frequent cache misses are dynamically moved to SPM to improve the memory bandwidth, with additional hardware components are required to monitor and control the run-time data accesses [18], [19].

To the best of our knowledge, this is the first paper that discusses HLS-based automatic data placement problem for off-the-shelf CPU-FPGA HMPSoC architecture, where array objects in an FPGA kernel can be accessed from either on-chip BRAMs, off-chip main memory via shared CPU last-level caches, or off-chip main memory directly (bypassing CPU caches). Compared with literature work on data placement for software-controlled on-chip memory of CPU architecture, we are confronted with a more complex design space, where the characteristics of FPGA computation, memory access patterns, as well as the architectural properties need to be explicitly considered.

III. Performance analysis and Motivation

A. Background

In an FPGA-based system design, it is always preferable to allocate data in the fast on-chip BRAMs to achieve higher memory bandwidth. However, the capacity of SRAM-based BRAMs are restricted due to high power consumption and low storage density. For example, the Zynq-7020 SoC has only 4.9Mbit BRAM in total [8]. Therefore, utilizing the limited on-chip BRAM resources efficiently has significant impact on the overall system performance for memory-intensive FPGA kernels. Moreover, whether to access the DDR memory via the ACP or HP further enlarges the system design space.

B. Memory latency analysis

In order to understand the memory hierarchy performance of CPU-FPGA HMPSoC, we build a memory access latency model in this work. We design a set of micro-benchmarks to measure the memory access latency for various memory hierarchy levels on Zedboard platform with Xilinx Zynq-7020 SoC under 100MHz clock frequency. The measurement results are shown in Table I based on Zedboard, and will be utilized in our static data placement framework in the following sections.

TABLE I
Measured average access latency on Zedboard

Notation	Description	Cycles
$BRAM_r$	BRAM read latency	2
$BRAM_w$	BRAM write latency	1
ACP_r^b	ACP read latency in burst mode	2.08
ACP_w^b	ACP write latency in burst mode	1.23
ACP_{rh}^{nb}	ACP read hit latency in non-burst mode	25.49
ACP_{wh}^{nb}	ACP write hit latency in non-burst mode	22.20
ACP_{rm}^{nb}	ACP read miss latency in non-burst mode	33.78
ACP_{wm}^{nb}	ACP write miss latency in non-burst mode	33.56
HP_r^b	HP read latency in burst mode	2.08
HP_w^b	HP write latency in burst mode	1.25
HP_r^{nb}	HP read latency in non-burst mode	33.65
HP_w^{nb}	HP write latency in non-burst mode	27.00

In order to measure the access latency for an individual memory hierarchy, we design a micro-benchmarking FPGA

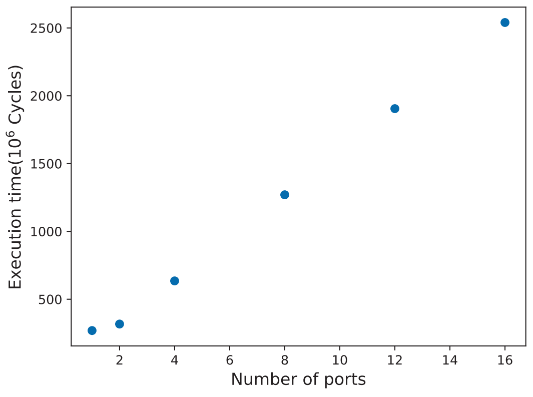


Fig. 2. Longest execution time with parallel accesses to an HP port.

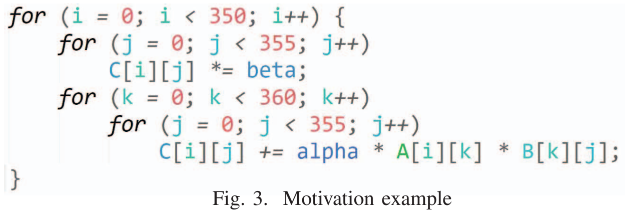
kernel to perform the particular type of memory accesses millions of times. For example, a kernel with repeated accesses to a huge matrix in column-ordering via ACP can be used to measure the average ACP read miss (L2 cache read miss) latency, where the matrix size is designed to ensure that all column-ordering accesses lead to cache misses given the L2 cache size.

Note that the AXI protocol supports a "burst mode" data transaction, in which multiple data items are transferred based upon a single address. In this mode, there is only one address transaction for multiple sequential data transactions with consecutive memory addresses. Vivado HLS automatically detects whether each array access instruction can be operated in the burst mode, and generates corresponding HDL code that benefits from the AXI burst mode transaction. Based on our measurement results, it makes little difference if an array access with burst mode is visited via BRAM, ACP(i.e., ACP_r^b, ACP_w^b), or HP (i.e., HP_r^b, HP_w^b).

We observe that on the Zedboard platform, the latency discrepancy between L2 cache hit and miss (e.g., between ACP_{rh}^{nb} and ACP_{rm}^{nb}) are *less significant* compared with that for a typical CPU architecture. It is probably due to the implementation of L2 cache access mechanism via AXI protocol. However, the measurement results could be different on other CPU-FPGA HMPSoC platforms.

Another observation that we have made is the memory access delay on an individual AXI slave port (either ACP or HP) is proportional to the number of parallel accesses to this port due to bus and controller interferences. For example, Fig. 2 shows the relationship between number of master ports (read requests from parallel FPGA kernels) and the longest kernel execution time (due to interference delay). Note that Zedboard provides 4 separate HP ports. Therefore, even distribution of the parallel HP requests among HP ports generally leads to better performance.

Finally, we would reiterate that the proposed data placement framework is generally applicable to any platforms with similar memory hierarchy, since it makes no assumption on any specific values or relative magnitude of the above-mentioned memory access latencies.



```
for (i = 0; i < 350; i++) {
    for (j = 0; j < 355; j++)
        C[i][j] *= beta;
    for (k = 0; k < 360; k++)
        for (j = 0; j < 355; j++)
            C[i][j] += alpha * A[i][k] * B[k][j];
}
```

Fig. 3. Motivation example

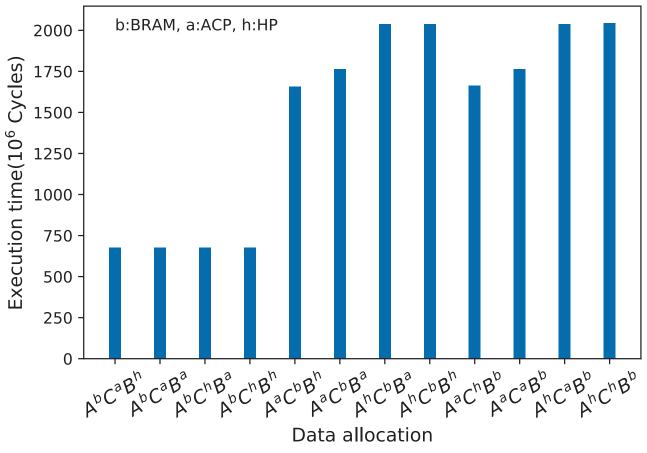


Fig. 4. Kernel execution time under different data placement.

C. Motivating example

In this section, we show that the data placement has significant impact on the system performance based on the Zedboard platform with Zynq-7020 SoC. Fig. 3 displays our motivating example of general matrix multiply (GEMM) from Polybench [20] (modified to show the actual data size). We assume that at most 1 of the 3 arrays can fit into the on-chip BRAMs. The experimental results are presented in Fig. 4, where $A^aB^bC^h$ indicates an allocation scheme that array A, B, and C are allocated to ACP, $BRAM$, and HP ports, respectively.

The results show that the optimal allocation ($A^bC^aB^h$) leads to 3.14X speedup compared with the worst allocation ($A^hC^hB^b$). Moreover, we would like to highlight some results that may be counter-intuitive to traditional CPU-based SPM allocation schemes.

Obs. 1 Array C has significantly higher read and write frequency compared with other two arrays. However, all schemes that map array C to on-chip BRAMs lead to non-optimal performance. On the other hand, since all accesses to array C are burst transaction, accessing C via ACP or HP and utilizing BRAM for other data objects may result in better system performance.

Obs. 2 Although array A has better temporal locality w.r.t. B and C, we should not access A via ACP and L2 cache. In this example, it is best to allocate A to BRAMs since accesses to A are the only non-burst transactions.

Obs. 3 The scheme $A^aC^hB^b$ outperforms $A^aC^aB^b$ even though we have more accesses via the faster ACP. This is because when multiple parallel (non-burst) accesses are mapped to the same slave port (ACP or one of the HPs), the port interferences retard the throughput of the particular port.

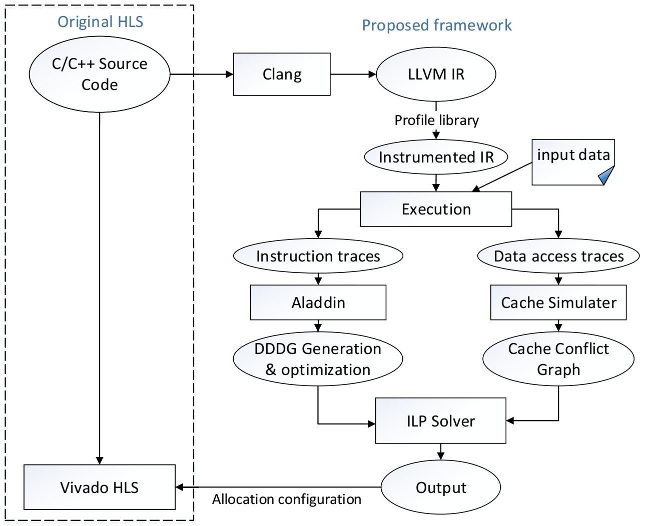
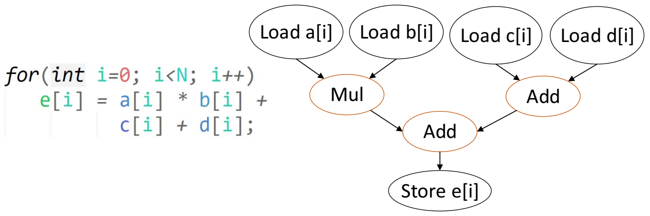


Fig. 5. The proposed framework



```
for(int i=0; i<N; i++)
    e[i] = a[i] * b[i] +
           c[i] + d[i];
```

Fig. 6. Source code and generated DDDG.

IV. Date placement framework

Fig. 5 shows the proposed static data placement framework for CPU-FPGA HMPSoC, and its integration with the original HLS toolchain. The source code is first compiled into Low-Level Virtual Machine (LLVM) intermediate representation (IR) [21] via the Clang front end. Second, we generate the dynamic data dependence graphs (DDDGs) via the instruction traces to represent the control- and data-flow of the FPGA kernel without generating RTL implementations. Meanwhile, the obtained data access traces are fed into a cache simulator to obtain the cache interference information. Finally, we formulate this problem into an integer linear programming (ILP) formulation, where the resulted optimal data allocation scheme is written into a ".tcl" configuration file for the subsequent HLS compilation.

A. LLVM and DDDG

The LLVM has been widely used in program analysis and optimization. We use the *llvm::Pass* module to add profiling instrumentation to get the data access and instruction traces. We then use to the Aladdin simulator [22] to generate DDDGs from the instruction traces, which has been proved to represent the FPGA kernel's control and data flow accurately ([7], [22]). Various techniques including tree height reduction and read-after-write dependency can be applied to generate optimized DDDGs.

Fig. 6 shows an example C source code and the generated optimized DDDG for the loop body (the control flow is omitted for simplicity). In this case, all load instructions are executed in parallel, and the execution time is bounded by the longest path in the graph.

B. ILP formulation

Decision variables. We first define a set of binary decision variables to indicate whether an array is allocated to a particular memory hierarchy. For example,

$$in\_bram(a_i) = \begin{cases} 1, & if\ array\ a_i\ is\ allocated \\ & to\ BRAM; \\ 0, & Otherwise. \end{cases} \quad (1)$$

indicates whether array a_i is allocated to BRAM. Similarly, we define the binary variables $in\_acp(a_i)$, $in\_hp[0](a_i)$, $in\_hp[1](a_i)$, $in\_hp[2](a_i)$, $in\_hp[3](a_i)$ to indicate whether Array i is allocated to the ACP port or one of the 4 HP ports (Zedboard implements 4 separated HP ports), respectively. Given that each array can be accessed from a fixed port, we have

$$in\_bram(a_i) + in\_acp(a_i) + \sum_{k=0}^{3} in\_hp[k](a_i) = 1 \quad (2)$$

Execution time of a basic block. The execution time of a basic block bb_i can be recursively calculated given its DDDG representation, i.e.,

$$T_{bb_i}^{exec} = T_{sink(bb_i)} + \max T_n | \forall n \in pred(sink(bb_i)) \quad (3)$$

where $sink(bb_i)$ is the unique sink node of the DDDG for bb_i, $pred(n)$ returns all predecessor nodes of n in the DDDG. If a particular node represents an operator (e.g., add, multiplication, etc) or a register access, its execution time can be provided by HLS. Otherwise, if a node represents a memory load instruction (similar to the store instructions) of array a_i, we have

$$T_{load(a_i)} = T_{loadBRAM(a_i)} + T_{loadACP(a_i)} + T_{loadHP(a_i)} \quad (4)$$

where

$$T_{loadBRAM(a_i)} = in\_bram(a_i) * BRAM_r \quad (5)$$

$$\begin{aligned} T_{loadACP(a_i)} = in\_acp(a_i) * (burst(a_i) * ACP_r^b + \\ (1 - burst(a_i)) * (ACP_{rh}^{nb} * HitRate(a_i) + \\ ACP_{rm}^{nb} * MissRate(a_i))) \end{aligned} \quad (6)$$

$$\begin{aligned} T_{loadHP(a_i)} = \sum_{k=0}^{3} in\_hp[k]_i * (burst(a_i) * HP_r^b + \\ (1 - burst(a_i)) * HP_r^{nb}) \end{aligned} \quad (7)$$

The memory access latencies are predetermined constants for the given platform (refer to Section III-B). For the corresponding load instruction, if the array accesses are in the burst model, we have $burst(a_i) = 1$; otherwise $burst(a_i) = 0$. The $HitRate(a_i)$ and $MissRate(a_i)$ are the L2 cache hit and miss rate for the given instruction, which is obtained from the cache simulation as shown in Fig. 5.

Port contention. Given that the port interference delay is proportional to number of parallel accesses to the port (Section III-B), We consider the port contention as an additional execution time overhead for parallel memory accesses with potential port interference. Let par be a set of parallel accessing array objects in the DDDGs, and $P(bb)$ be the set all such parallel array sets in a basic block bb, we have the port contention delay for each execution of bb as follows,

$$\begin{aligned} T_{bb}^{port} = &\sum_{par \in P(bb)} \Big(\sum_{\forall a_i \in par} in\_acp(a_i) * cost_{acp} \\ &+ \sum_{k=0}^{3} \big(\sum_{\forall a_i \in par} in\_hp[k](a_i) * cost_{HP} \big) \Big) \end{aligned} \quad (8)$$

where $cost_{ACP}$ and $cost_{HP}$ is the measured port contention delay between two parallel accessing arrays.

Cache interferences. In this work, we use the *standalone* mode in Xilinx SDK. Therefore, the data memory traces contain the physical address to be used in accessing the physically-index physically tagged cache. Similar to [17], we build the cache conflict graph to estimate the overall cache interferences between each pair of arrays. Two arrays a_i and a_j may have cache interferences only if both of them are allocated to the ACP port, we have

$$conflict(a_i, a_j) = \begin{cases} 1, & if\ in\_acp(a_i) \wedge in\_acp(a_j) \\ 0, & Otherwise. \end{cases} \quad (9)$$

The overall cache interference cost can be estimated as

$$\begin{aligned} T^{cache} = &\sum_{i=0}^{N-1} \sum_{j=i+1}^{N} (conflict(a_i, a_j) * \\ &conflictCount(a_i, a_j) * cost_{cc}) \end{aligned} \quad (10)$$

where N is the total number of arrays in the kernel, $conflictCount(a_i, a_j)$ is the number of cache conflicts between the two arrays given in the cache conflict graph, and the per-conflict cost

$$cost_{cc} = ACP_{rm}^{nb} - ACP_{rh}^{nb}$$

given the memory access latencies in Section III-B.

Objective. Finally, given the DDDG representation of an FPGA kernel, the overall execution time can be estimated as

$$T_{total} = \sum_{bb_i \in B} (T_{bb_i}^{exec} + T_{bb_i}^{port}) * N_{bb_i} + T^{cache} \quad (11)$$

where B is the set of all basic blocks, and N_{bb_i} is the known execution count of basic block bb_i. The objective of the ILP formulation is to minimize the above-mentioned overall execution, subject to the BRAM capacity constraint

$$\sum_{i=0}^{N} (in\_bram(a_i) * size(a_i)) \le size_{bram} \quad (12)$$

where N is the total number of arrays in the kernel, $size(a_i)$ and $size_{bram}$ are the size of array a_i and total BRAM capacity, respectively. The results of the ILP formulation are the optimal data placement scheme which can be automatically accepted by HLS to perform data allocation.

V. Experiments

A. Experiment Setup

The evaluation results are obtained on the Digilent Zedboard Evaluation Board, which contains two Cortex-A9 ARM cores and a Xilinx Zynq-7020 chip running at 100MHz. We disable the cache prefetch and enable the read/write cacheline allocation. We use Vivado Design Suite v2016.4, LLVM 3.4 with Clang 3.4 to generate the HDL and bitstream. Gurobi with Python interface is adopted as the ILP solver.

Table II shows the selected benchmarks from Polybench [20] and input size (defined by the variables in the "Input Size" column) used in our evaluation. We choose all available benchmarks from Polybench such that each benchmark contains at least three arrays with tunable data size. As results, we have a relatively large design space to explore, where different subsets of the arrays can be fit into on-chip BRAMs (but not all of them).

TABLE II
Benchmarks and input size

Application	**Description**	**Input Size**
2MM	2-Matrix multiply	I:200 J:210 K:205 L:215
3MM	3-Matrix multiply	I:165 J:170 K:175 L:180 M:185
GEMM	Matrix multiply	I:350 J:355 K:360
SYMM	Symmetric matrix multiply	M:330 N:360
SYR2K	Symmetric rank 2k	M:360 N:350
DERICHE	Deriche recursive filter	W:362 H:360
ADI	Alternating direction implicit	TSTEPS:5 N:360
FDTD-2D	Finite-Difference Time-Domain	TMAX:5 X:245 Y:255
GS	Modified Gram Schmidt	M:360 N:350

Given that there is no literature work on data placement for CPU-FPGA HMPSoC, we design a simple *GREEDY* strategy as the baseline. In this strategy, the arrays with the highest per-element access frequency are allocated into BRAMs subject to the BRAM size. For arrays that have identical access frequency and size, the BRAM allocation priority follows the order of array declarations in the source program. All other array objects are accessed via the ACP port, which follows the intuition that caches in general speedup the memory subsystem performance (compared with accessing these data directly from the DDR via HP ports).

B. Experiment results

TABLE III
Allocation results

Application	*greedy*		*opt*			Time
	BRAM	ACP	BRAM	ACP	HP	(ms)
2MM	A,tmp,D	B,C	tmp,C,D	B	A	1.5
3MM	E,F,C,G	A,B,D	B,F,D,G	E	A,C	1.4
GEMM	C	A,B	A	C	B	1.4
SYMM	C	A,B	C	B	A	6.6
SYR2K	C	A.B	B	A	C	4.1
DERICHE	y1	iI,iO,y2	iO	y1,y2,iI	N/A	2.1
ADI	p	u,v,q	v	p,u	q	11.6
FDTD-2D	y,z,f	x	x,z,f	y	N/A	1.5
GS	A	Q,R	A	Q	R	1.2

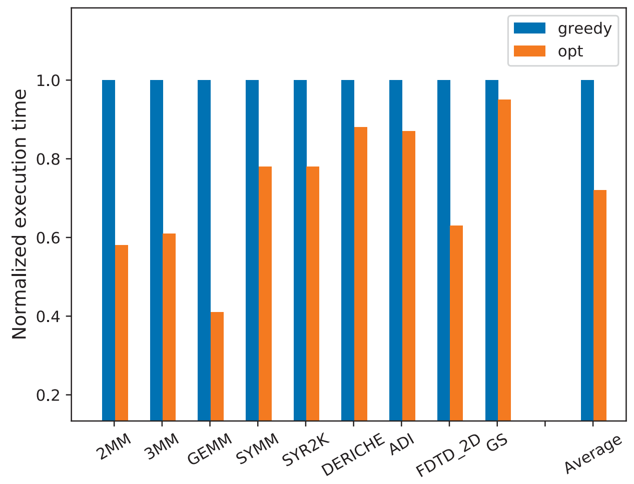


Fig. 7. Experiment results

The experimental results in Fig. 7 show an average 27.89% performance improvement (1.39X speedup) with the proposed data placement framework compared to a greedy approach. Table III shows the allocation results for both approaches, as well as the total analysis time of the proposed framework. We make the following observations in the experimental results.

- The average performance improvement for `2MM`, `3MM`, `GEMM`, `FDTD-2D` is around 36%. The main reasons for the speedup are (i) the greedy-based allocation maps the array(s) with highest frequency to BRAMs, without considering the burst mode accesses; (ii) mapping parallel no-burst accesses to the same ACP/HP port leads to port contention.
- For `SYMM` and `GS`, both approaches allocate the same array to the BRAM space. Our approach leads to further performance improvement by separating non-burst array accesses to different ports to avoid port contention and L2 cache conflicts.
- For both `SYR2K` and `ADI`, they contain multiple non-burst accessing arrays with poor locality, these arrays should be given priority to BRAM allocation subject to BRAM capacity, and separated to different ports to avoid port and cache conflicts.
- `DERICHE` is a computation-bounded kernel, where the memory subsystem performance has less impact on the overall system performance.

VI. Conclusion

In this work, we propose an HLS-based automatic data placement framework CPU-FPGA HMPSoC architecture. We show that besides the traditional considerations including access frequency and locality, other computation and architectural characteristics need to be captured in order to obtain an optimal allocation scheme. Experimental results on real platform show that the proposed framework leads to significant overall system performance improvement.

Acknowledgment

This research is sponsored by Shandong Provincial Natural Science Foundation No. ZR2017MF034; the Independent Innovation Engineering of Shandong Province under Grant No. 2017CXGC0703; and the Science & Technology Development Program of Shandong Province No.2015GGX106002.

References

[1] R. Nane and et al., "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.
[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.
[3] Xilinx, "Vivado-design-suite," 2016.
[4] A. Canis and et al., "LegUp: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 24, 2013.
[5] S. Wang, Y. Liang, and W. Zhang, "Flexcl: An analytical performance model for opencl workloads on flexible fpgas," in *Design Automation Conference (DAC)*, 2017.
[6] J. Zhao and et al., "COMBA: a comprehensive model-based analysis framework for high level synthesis of real applications," in *International Conference on Computer-Aided Design*, 2017.
[7] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: a high-level performance analysis tool for fpga-based accelerators," in *Design Automation Conference*, 2016.
[8] Xilinx, "Zynq-7000-overview," 2018.
[9] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Bandwidth optimization through on-chip memory restructuring for HLS," in *Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.
[10] R. Zhao and et al., "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *International Symposium on Field-Programmable Gate Arrays*, 2017.
[11] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," in *International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 240–249.
[12] D. Diamantopoulos, S. Xydis, K. Siozios, and D. Soudris, "Dynamic memory management in vivado-hls for scalable many-accelerator architectures," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2015, pp. 117–128.
[13] M. Adler and et al., "Leap scratchpads: automatic memory and cache management for reconfigurable logic," in *International symposium on Field programmable gate arrays*. ACM, 2011, pp. 25–28.
[14] H.-J. Yang, K. Fleming, F. Winterstein, M. Adler, and J. Emer, "Scavenger: Automating the construction of application-optimized memory hierarchies," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 10, no. 2, p. 13, 2017.
[15] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for mpsoc architectures," in *International conference on Compilers, architecture and synthesis for embedded systems*, 2006.
[16] J. Lu, K. Bai, and A. Shrivastava, "Ssdm: smart stack data management for software managed multicores (smms)," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 149.
[17] M. Verma, L. Wehmeyer, and P. Marwedel, "Cache-aware scratchpad allocation algorithm," in *Proceedings of the conference on Design, automation and test in Europe-Volume 2*, 2004.
[18] D.-W. Chang and et al., "Casa: Contention-aware scratchpad memory allocation for online hybrid on-chip memory management," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 12, pp. 1806–1817, 2014.
[19] D.-W. Chang, C. Lin, and L.-C. Yong, "Rohom: Requirement-aware online hybrid on-chip memory management for multicore systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 357–369, 2017.
[20] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www.cs.ucla.edu/pouchet/software/polybench*, 2012.
[21] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on Code generation and optimization*, 2004, p. 75.
[22] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 97–108.