Dynamic Heap Management in High-Level Synthesis for Many-Accelerator Architectures

Argyris Kokkinis
Aristotle University of Thessaloniki
Thessaloniki, Greece
akokkino@physics.auth.gr

Dionysios Diamantopoulos
IBM Research Europe
Ruschlikon, Switzerland
did@zurich.ibm.com

Kostas Siozios
Aristotle University of Thessaloniki
Thessaloniki, Greece
ksiop@auth.gr

***Abstract*—Dynamic Memory Management (DMM) in High-Level Synthesis has been introduced as a promising solution for optimizing the accelerators' memory usage and reducing the occupied on-chip area. Schemes for dynamic memory allocation have been suggested for many-accelerator architectures where memory sharing and resource reusing has the potential to increase the number of synthesized accelerators, rising the throughput per Watt ratio. However, in those architectures, the simultaneous execution of many accelerators may reduce memory efficiency, increasing the Memory Allocation Failures (MAFs) as a consequence of the sub-optimal utilization of the shared memories. MAFs due to memory fragmentation can reach up to 38.5% of the overall memory allocation failures when accelerators with heterogeneous allocation sizes are executed in parallel in a shared memory space. In this manuscript we propose an HLS methodology for minimizing MAFs for many-accelerator DMM frameworks that are caused by on-chip inefficient memory utilization. Our proposed methodology is orthogonal to the static memory allocation techniques of the Xilinx Vitis suite and was evaluated using Xilinx Vitis/Vitis HLS 2020.1 on an Alveo U200 FPGA device as an extension of the Memluv DMM framework. In the experimental results we show that our proposed methodology may decrease up to 38.5% the MAFs due to fragmentation and up to 91% the overall allocation fails with a controllable increase on the utilized resources and a on the accelerators' latency.**

***Index Terms*—HLS, FPGA, Dynamic Heap Management, Memory efficiency**

I. Introduction

The recent advancements in embedded high-value applications and the emerging technologies targeting cognitive systems and time-critical applications have shifted the computing paradigm towards high performance and low-power designs [1]. At the same time, the continued scaling of scientific computations has led to the employment of heterogeneous computing architectures for accelerating computationally intensive tasks and meeting the performance requirements [2]. FPGA architectures have been proven capable of delivering high-performance computations in a limited power budget [3] and data center acceleration cards abound of computing resources that can be exploited for designing high-performance solutions by integrating multiple accelerators on FPGA devices [4]. In an High-Level Synthesis (HLS) context, such architectures are developed based on C/C++ algorithmic representations that allow a rapid implementation of high-performance accelerators and have been suggested as an alternative to the typical Register-Transfer level (RTL) design flow for increasing the accelerators' throughput and reducing their power expenses [5]. Nevertheless, previous studies indicated that the rapid starvation of the Field-Programmable Gate Arrays FPGAs' on-chip static memories called Block RAMs (BRAMs)[1]limits the accelerators' scalability [7]. In order to mitigate this phenomenon; dynamic memory management frameworks (DMM) that allow accelerators' to share and reuse BRAM resources at execution-time by using dynamic memory allocation schemes have been suggested [7] - [14]. Those memory allocation schemes group BRAMs, creating pools of shared local memories that are referred to as heaps.

In this paper we propose a methodology for optimizing the heap utilization in many-accelerator DMM frameworks. The proposed mechanism is agnostic of the dynamic memory allocation scheme and aims to minimize the Memory Allocation Failures (MAFs) that are either due to memory fragmentation or sub-optimal utilization of the instantiated heaps. Throughout this manuscript a MAF is referred to the condition where an accelerator's memory allocation request cannot currently being served by the memory allocator due to lack of available memory space.

The proposed mechanism was evaluated as an extension of the open-source HLS DMM framework, Memluv [7]. To the best of our knowledge this is the only HLS DMM framework targeting many accelerator architectures. We show that by using the proposed mechanism, the MAFs that are caused by memory fragmentation can be decreased up to 38.5% on average, while the MAFs that are due to sub-optimal heap usage can be decreased up to 91% on average, in comparison to the Memluv framework.

The rest of this manuscript is structured as follows: In Section II the motivation of this work is elaborated and previous works are discussed. In Section III the proposed methodology is presented. In Section IV the evaluation results are shown. Section V concludes this paper.

II. Motivation and Related Work

Previous studies in DMM frameworks have indicated that in order to increase the memory efficiency and optimize the accelerators' performance each accelerator should be assigned to a private memory space (i.e heap) [7] [8]. Although, this

[1]The term BRAM is used by Xilinx FPGAs for characterizing on-chip dual port memories of 18Kbits [6]

1946-1488/22/$31.00 ©2022 IEEE
DOI 10.1109/FPL57034.2022.00051

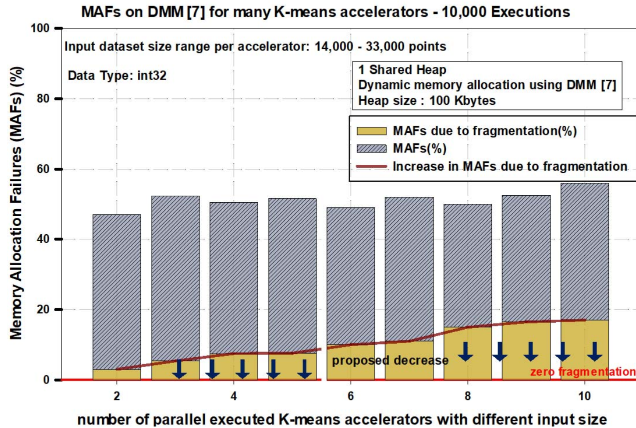


Fig. 1: MAFs due to memory fragmentation at parallel execution of 20 K-means accelerators

scheme maximizes the performance gains, it undermines the accelerators' density, as more BRAM resources are utilized for the implementation of multiple heaps. Instantiating few heaps and sharing their memory resources among the implemented accelerators' has been suggested as a possible solution for increasing the number of on-chip accelerators. This technique may increase the accelerators' density up to 3.8x on average [7], however MAFs caused by memory fragmentation is a bottleneck for optimizing the memory usage when multiple accelerators of heterogeneous memory allocation sizes are executed in parallel and share the same memory space. Figure 1 depicts an exemplary case where 20 K-means accelerators are executed in parallel, utilizing the Memluv DMM framework for dynamic memory allocation and share one heap. The horizontal axis shows how many of those 20 parallel executed accelerators perform clustering on datasets of different sizes and therefore have heterogeneous memory allocation sizes, while the summary of their memory requirements is fixed. The vertical axis shows the percentage of MAFs from their executions, when each case is executed 10,000 times. Based on this diagram, the percentage of allocation failures that are due to memory fragmentation may reach up to 17.4 % on average of the overall executions when 20 K-means accelerators cluster datasets of 10 different sizes.

The MAFs due to memory fragmentation are dependant on the heterogeneity of the allocation sizes of the running accelerators [9]. Memory patterns composed of memory allocation and de-allocation requests of different sizes are more likely to fail due to fragmentation as it is shown in figure 1. For the rest of this manuscript, the different memory allocation sizes per heap will be mentioned as Distinct Allocation Sizes (DAS).

Previous works [7] – [14] on DMM frameworks for HLS try to optimize the memory usage by selecting appropriate memory allocation schemes.

Xue and Thomas [10] proposed an HLS friendly DMM framework that utilizes both off-chip DDR memories and local BRAMs, using a buddy tree allocation scheme for the allocation. Although this allocator is efficient for reducing external fragmentation it may cause waste of memory due to internal fragmentation [16].

The same allcation scheme is used in another work by Liang et al. [8]. Hi-DMM is a DMM framework that is developed as an Intellectual Property (IP) block on the FPGA's programmable logic region. The synthesized HLS accelerator communicates with the different implemented memory allocators through AXI bus.

Giamblanco et al. [11] presented a library of different HLS allocation mechanisms that can be used in different application scenarios depending on the performance-memory efficiency requirements. The `gnumem` memory allocator that is introduced in this work, is based on the Doug Lea's allocation mechanism [20] which performs memory block coalescing when a memory block adjacent to an unoccupied region is de-allocated. Although this technique reduces fragmentation it is not always applicable.

In this manuscript, a dynamic heap management methodology that is agnostic of the memory allocator and minimizes MAFs is proposed. This work aims to decrease MAFs by reducing memory fragmentation and by redirecting the accelerator' memory allocation requests to the instantiated heaps in order to increase memory efficiency.

III. Methodology

The proposed HLS flow consists mainly of two stages: 1) An offline analysis stage where the conditions that enable the mechanism that optimizes the heap usage are determined and 2) an online execution stage where a garbage collection and a heap selection mechanism are executed. This flow is illustrated in figure 2.

A. Offline Analysis

During the analysis stage, the heaps' fragmentation ratio that will lead to the activation of the garbage collection mechanism is determined.

This analysis is performed on a library of different C++ accelerators with varied memory allocation sizes and memory patterns. The DAS from the provided library of accelerators are extracted and are used for generating different memory patterns that correspond to an overlapping execution of those accelerators. The memory patterns are outputs generated by the *pattern generator* block. This block pseudo-randomly repositions the accelerators' memory allocation (i.e *Malloc*) and de-allocation (i.e *Free*) commands, creating pseudo-random sequences of *Malloc* and *Free* pairs that are linked in order to retain their relative order. Those patterns are used in a Monte-Carlo simulation as it is depicted in figure 2. In this simulation, two instances of the same C++ application that is generated per memory pattern are executed. Both of them use the HLS DMM framework [7] for dynamic memory allocation but only one instance has the proposed garbage collection mechanism activated.

To determine the activation thresholds (called Θ) that should be set for each heap in order to minimize MAFs, 10,000 different memory patterns are generated and executed at an algorithmic level. This simulation is performed iteratively for

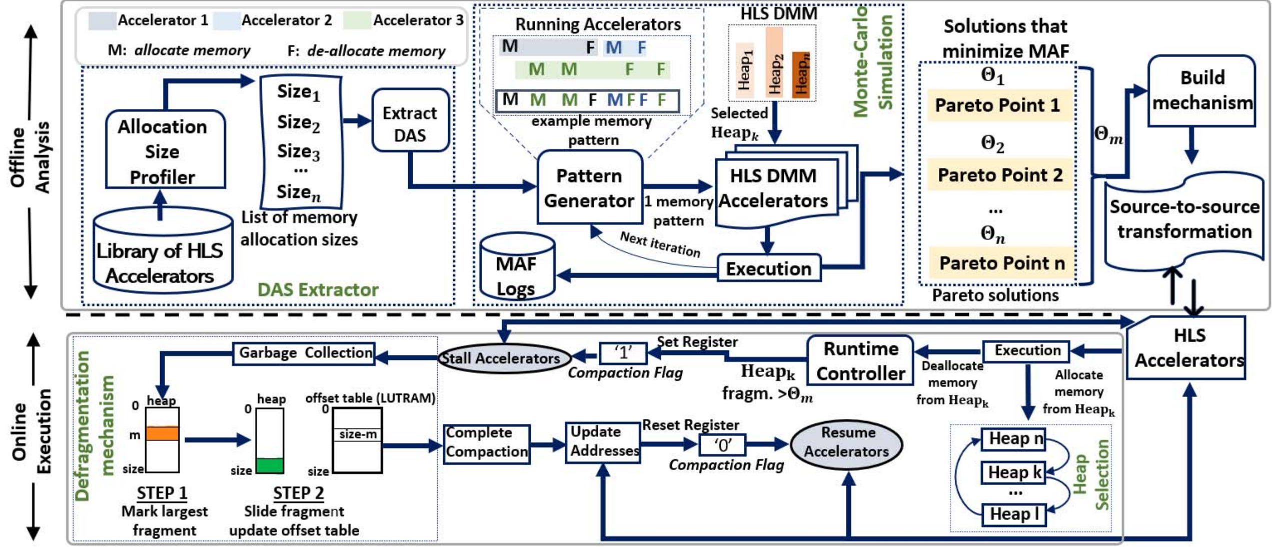


Fig. 2: Proposed methodology for minimizing MAFs: (i) offline analysis, (ii) online execution

each unique heap size of the DMM's instantiated heaps and for fragmentation ratios ranging from 0% to 90%. After each simulation the fluctuations at the heaps' fragmentation and utilization ratios of the two C++ applications are reported.

The output of this analysis is the Pareto curve for the different Θ values that trade-off performance with decrease in MAFs with the Pareto frontier including all those Θ solutions that decrease the fragmentation MAFs. As the accelerators' latency is dependant on the frequency of the garbage collector's executions, each execution induces a latency overhead on the accelerators' performance but optimizes the heap utilization.

B. Execution Phase

During the execution stage there are two mechanisms that are activated depending on the memory fragmentation and memory utilization ratios of the instantiated heaps. 1) The heap defragmentation mechanism and 2) a heap selection operation.

1) ***Heap defragmentation****:* The heap defragmentation operation refers to the activation of the garbage collection mechanism that is automatically triggered when the memory fragmentation ratio exceeds the selected activation threshold (Θ_m). The task of defragmenting the heap is performed by applying a memory compaction algorithm. Specifically, this mechanism includes mainly three sub-components namely: i) a distributed RAM structure, ii) a 1-bit flag register and iii) the garbage collection mechanism as it is illustrated at the lower left part of figure 2.

The distributed RAM structure (called `offset table`) is a memory block implemented in Look-Up Tables (LUTs). This component is used for the task of storing the number of positions of the memory objects that have been relocated within the heap at the event of memory compaction. In more detail, HLS tools such as Vitis HLS [17] and LegUp [18] support a single address space and therefore designs with pointer-to-pointer arithmetic are not synthesizable. Hence, every element in the `offset table` LUTRAM is linked to a heap memory element indicating the number of positions that each word in memory has been shifted.

The flag register (called `compaction flag`) is a 1-bit register that is used for blocking the accelerators from accessing the heaps that are currently under processing for defragmentation by the garbage collector. The number of the implemented `compaction flag` registers is equal to the number of the instantiated heaps, with each register linked to a heap. Since the execution order of the parallel running accelerators is not under constraints; the heaps' fragmentation and utilization ratios may fluctuate at any time during the accelerators' execution, and thus there is unpredictability on the activation time of the garbage collection mechanism.

Hence, when the garbage collection is activated on a specific heap; the status of its corresponding `compaction flag` register is set high blocking the accelerators' from accessing this heap and inducing memory violations. The status of this register is reset to zero when the defragmentation operations are completed.

Memory Compaction Algorithm: The heap defragmentation task is performed by a Mark-Compact garbage collection algorithm [19]. During the algorithm's marking phase, the heap is linearly traversed and the *live* [2] memory objects are marked by setting high the bits of a heap-size register. After the completion of this phase, the heap's largest fragmented memory region is identified. Then, during the compaction phase, the garbage collector reallocates the occupied memory blocks to the bottom of the heap, by shifting the live memory blocks at the bottom heap addresses, resulting to the defragmentation of the heap's largest fragmented region. The pseudo-code of the described process is presented in the algorithm 1. During

[2]as *live* we consider the already allocated memory objects

Algorithm 1 Compaction algorithm

1: **procedure** COMPACTION($Heap, Offset_Table$)
2: $current_size \leftarrow 0$ ▷ size of a free block
3: $fragment_size \leftarrow 0$ ▷ size of the largest fragment
4: $Occupied_pos \leftarrow 0$ ▷ end of fragment
5: **for** $i = 0$ to $Heap.size$ **do** ▷ find largest fragment
6: **if** $CheckHeapWord(i) = unallocated$ **then**
7: $current_size \leftarrow current_size + 1$
8: **else**
9: **if** $current_size > fragment_size$ **then**
10: $fragment_size \leftarrow current_size$
11: $Occupied_pos \leftarrow i$
12: **end if**
13: $current_size \leftarrow 0$
14: **end if**
15: **end for**
16: **for** $i = Occupied_pos$ to $Heap.size$ **do**
17: **if** $CheckHeapWord(i) = allocated$ **then**
18: $Heap[i - fragment_size] \leftarrow Heap[i]$
19: $Offset_Table[i] \leftarrow -fragment_size$
20: **end if**
21: **end for**
22: **end procedure**

the execution of the described algorithm, the relocation of each live memory object is followed by the corresponding update on the `offset table` LUTRAM. When an occupied heap element is shifted then its linked element in the `offset table` keeps track of the memory positions that it has been relocated.

Code transpilation: The accelerators' code is source-to-source transformed, resulting to the generation of an equivalent C++ code that utilizes two functions, namely the `Update` and the `checkFlag`. Those functions are part of the framework's API and allow the accelerators to update the addresses of the relocated memory objects in the event of memory compaction and stall their execution.

```
void* Update(void* address, uint_t nbytes,
uint_t heap_ID)
```

```
uint8_t CheckFlag(uint_t heap_ID)
```

In more detail, the `checkFlag()` function returns the content of the `compaction flag` register for the corresponding heap (`heap_ID`). When the garbage collection mechanism is enabled, the accelerators monitor through the `checkFlag()` function the status of the heaps' `compaction flag` register and stall their operation. As the stall in the accelerators' execution is required before any memory access operation in order to avoid memory errors; this block of code is placed before the accelerators accessing dynamically allocated memory. In order to avoid access conflicts caused by the parallel running accelerators reading simultaneously the content of the heaps' `compaction flag` register, multiple such registers are instantiated per heap and are uniquely assigned to each accelerator.

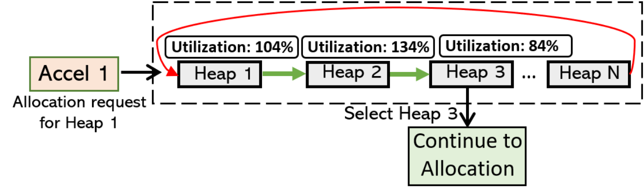


Fig. 3: The proposed heap selection mechanism performs linear search on the instantiated heaps

C++ Accelerator

```
A=(int*)Alloc(size,heap_ID);
{
   //operations
    stall macro
   //memory access
}
Free(A,size,heap_ID);
```

```
If(checkFlag(heap_ID))
{ A=(int*)Update(A,size,heap_ID);
  while(checkFlag(heap_ID));
}
```

Fig. 4: Source code transformation

The `Update()` function is used for reassigning the address of the dynamically allocated memory object that was previously stored in different memory region. At the back-end, the `Update()` function performs a linear search at the heap's offset table LUTRAM and calculates the object's new memory position by adding its previously, assigned by the allocator address with the sliding factor that is stored in the offset table. Figure 4 depicts the transformed C++ code that utilizes the `CheckFlag` and `Update` functions before any memory access by the accelerator.

*2) **Heap selection**:* A mechanism that transparently directs the memory allocation requests to specific heaps is proposed as a solution for minimizing MAFs due to sub-optimal heap utilization. Typically, when an accelerator issues a memory allocation request, the heap that shall be used for this allocation is specified either by the accelerator at execution-time [7] or it is defined offline [8]. However, when many accelerators are executed in parallel, the specified heap may not be able to serve this request due to lack of available unoccupied memory space which in turn may lead to a MAF.

In the proposed mechanism, after the execution of a `Malloc`/`Free` request, the heaps' utilization ratios are reported and stored in a distributed memory implemented in LUTs. Each, word in this memory is linked to one of the instantiated heaps indicating its utilization percentage (0 to 100).

When a `Malloc` command is executed, the proposed heap selection mechanism performs a linear search on this distributed memory and calculates the heaps' new utilization ratios. A ratio that exceeds 100 indicates that this heap is not currently able to serve this request due to lack of available space, while any ratio smaller than 100 means that the requested memory allocation can be served immediately, if this heap is not highly fragmented. When the first heap that can potentially allocate the requested size is found then the linear search halts and this heap is selected for the current allocation. This mechanism returns to the accelerator, the id

TABLE I: Evaluated algorithms

Algorithm	Min.Input Data	Max.Input Data	Data Type	Domain
K-means	10,000	34,000	int32	Machine Learning
Moving average filter	4,000	30,000	int32	Signal Processing
2D Dense Matrix Multiplication	8x8	50x50	int32	Scientific Computing

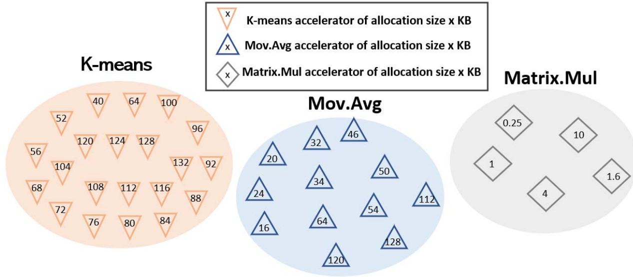


Fig. 5: Pools of accelerators with varied memory allocation sizes

of the heap that was selected. This process is depicted in figure 3.

IV. Evaluation

The proposed methodology was evaluated with Xilinx Vitis HLS 2020.1 tool on a Xilinx Alveo U200 FPGA platform, under the timing contraints of 250 MHz. The evaluation procedure was performed based on the memory pattern classification suggested by Giamblanco et al [11]. The `dmbenchhls` [11] suite classifies the memory allocation patterns into three categories: i) *triangular*, ii) *random*, and iii) *square*. More precisely, in many-accelerator domain triangular patterns typically correspond to sequentially executed kernels. In this case MAFs are not an issue and therefore there are no benefits from the proposed methodology. On the other hand, random memory patterns correspond to requests for memory allocation/de-allocation that would appear during parallel execution of many accelerators, while square memory patterns appear when multiple similar accelerators in terms of latency and memory requests are executed completely in parallel and hence this scenario can be categorized as a sub-set of the random memory patterns in our case. For evaluating our proposed methodology random memory patterns that are generated from real-world applications are used. The evaluation algorithms are summarized in table I. The second and third column of table I show the minimum and maximum input of those algorithms which in turn determines their minimum and maximum memory allocation sizes. For experimental purposes the proposed methodology was evaluated per accelerator type, as the parallel execution of similar accelerators is a common technique for increasing the design's overall throughput [20]. Therefore, for this evaluation we create three pools of different type of algorithms; each pool is composed of multiple accelerators with varied memory allocation sizes, as it is illustrated in figure 5.

Initially, we explore the efficiency of the proposed methodology to minimize the MAFs due to memory fragmentation for the three type of algorithms. In this experiment, we pseudo-randomly select multiple accelerators from the same pool to share the same heap and perform the analysis that was discussed in Section III. The decrease in MAFs and the accelerators' latency are reported for different activation thresholds Θ, as these are produced from the Pareto curves that are generated from the Monte-Carlo analysis. In this analysis, the Memluv DMM framework was used for the accelerators' dynamic memory allocation requests, as this is the only publicly available DMM framework for HLS that explicitly targets many-accelerator architectures.

Figures 6a-8a summarize the results of this analysis for alternative activation thresholds Θ. Specifically, the Θ values that correspond to heap fragmentation ratios of 0,10,15 and 20% are presented. The reference solution for this analysis (depicted with black color) corresponds to the results that were retrieved from the Memluv framework. In case we have a Θ=0% threshold, then the garbage collector is executed after a `free()` request rises the fragmentation ratio to any non-zero value. As the Θ increases the garbage collection is triggered less frequently which in turn leads to MAFs that approach the reference line (i.e the solution without defragmentation). In the case of matrix multiplication accelerators (figure 8a) Θ values above 10% have a small effect in reducing the MAFs due to fragmentation as the MAFs for the $\Theta = 15\%$ and $\Theta = 20\%$ solutions are almost identical to the reference line.

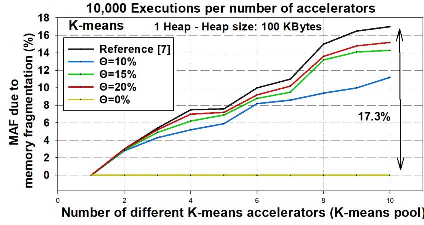
Based on this analysis, the $\Theta = 0\%$ solution that nullifies heap fragmentation reduces the MAFs by 17.3%, 38.5% and 14.7% for the three evaluated types of algorithms.

TABLE II: Overhead in resources and performance for different heap depths and for 32-bit wide words

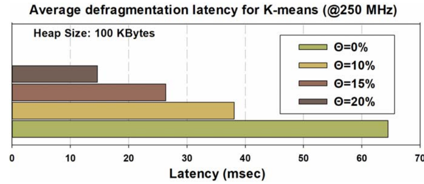
Heap Depth	LUTs	FFs[3]	Min.Latency	Avg.Latency	Max.Latency
512	2977	1566	40 us	56 us	73 us
1024	3232	1663	79 us	88 us	0.14 ms
8192	6345	1787	0.64 ms	0.94 ms	1.17 ms
25600	25345	2467	1.98 ms	2.93 ms	5.18 ms
30720	29654	2843	2.34 ms	3.27 ms	5.47 ms

Similarly, the bar-plots in figures 6b-8b show the average latency overhead that is imposed by the proposed defragmentation mechanism for different Θ thresholds. Lower Θ values result to more frequent activation of the garbage collection mechanism and therefore more stalls occur during the accelerators' operation. On the contrary, high Θ lead to fewer stalls and hence less frequent activation of the garbage collection mechanism for optimizing the memory usage of the corresponding heap. The maximum average latency is observed at the K-means accelerators where for $\Theta = 0\%$ there is a performance overhead of 64.46 ms (at a clock frequency of 250 MHz). For the moving average and matrix multiplication accelerators the average latency overhead for the same value of Θ is 42.48 and 17.8 msec respectively. Table II summarizes the minimum, average and maximum latency (at 250 MHz) of one execution stall for different

[3] Flip-Flops

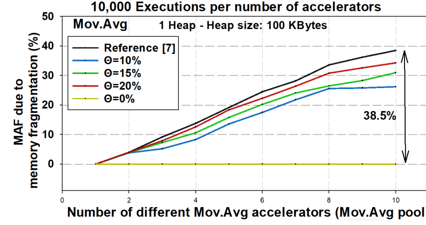


(a) MAFs due to fragmentation for different Θ values

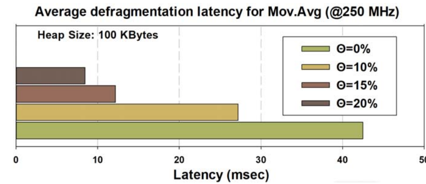


(b) Defragmentation latency for different Θ

Fig. 6: K-means accelerators

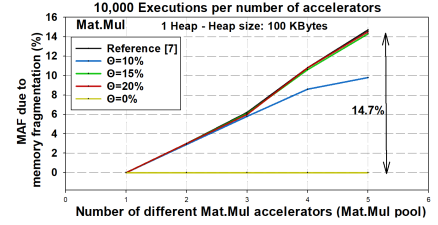


(a) MAFs due to fragmentation for different Θ values

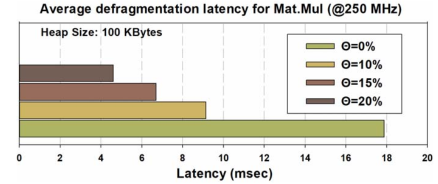


(b) Defragmentation latency for different Θ

Fig. 7: Mov.Avg accelerators

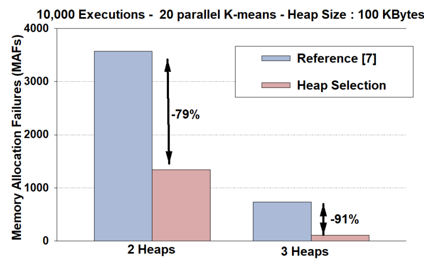


(a) MAFs due to fragmentation for different Θ values

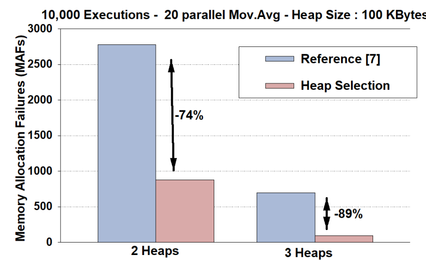


(b) Defragmentation latency for different Θ

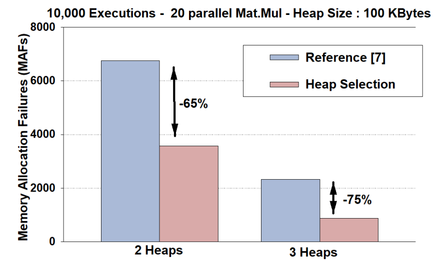
Fig. 8: 2D Matrix.Mul accelerators



(a) K-means accelerators



(b) Mov.Avg accelerators



(c) 2D Matrix.Mul accelerators

Fig. 9: Evaluation results of the heap selection mechanism for the three types of accelerators

heap sizes, as well as the FPGA resources that are utilized by the proposed mechanism on Alveo U200. It is noted that the DSP (Digital Signal Processing) and BRAM utilization by the proposed mechanism is zero. Furthermore,the garbage collector's maximum operating frequency is 360 MHz on Alveo U200.

In order to evaluate the effectiveness of the heap selection mechanism to minimize MAFs, we performed a Monte-Carlo simulation for testing different memory patterns. Each memory pattern is generated according to the analysis methodology that was described in Section III and corresponds to parallel executed accelerators from the three types (i.e K-means, Mov.Avg and 2D Mat.Mul). Figures 9a-9c depict the ratio of MAFs that were produced from 10,000 different memory patterns for each one of those accelerators when the requests for dynamic memory allocation are shared among two and three heaps respectively. The reference MAFs ratio was derived from the Memluv DMM framework where the proposed heap selection mechanism is not utilized. Initially, the memory allocation requests of each accelerator are evenly distributed to the instantiated heaps. In each Monte-Carlo iteration, 20 accelerators from the corresponding pool are pseudo-randomly selected for execution.

In all three cases, the utilization of three heaps decreases the overall MAFs due to the distribution of the accelerators' memory allocation requests to multiple heaps, as expected. By applying the heap selection mechanism, the allocation requests are redirected to different heaps at execution-time which in turn leads to an efficient memory utilization. As a result, a decrease of up 91% is achieved when the allocation requests from the K-means accelerators are shared in 3 heaps, while the minimum decrease appears when 20 Matrix Mul. accelerators are shared between two heaps, where the decrease in MAFs in 65% on average.

V. Conclusions

In this paper, we presented a methodology for optimizing the heap utilization for HLS DMM that aim to increase the number of the FPGA synthesized accelerators by allowing them to allocate/de-allocate BRAM resources at execution-time. The proposed methodology was evaluated on Memluv HLS DMM framework [7] and minimizes the MAFs that occur either due to memory fragmentation or due to sub-optimal utilization of the implemented heaps. Based on a Monte-Carlo analysis we showcase a trade-off between minimizing memory fragmentation and performance which varies depending on the executed accelerators.

Acknowledgment

This work has been supported by the E.C. funded program SERRANO under H2020 Grant Agreement No: 101017168.

REFERENCES

[1] Reed, D., Gannon, D., & Dongarra, J. (2022). Reinventing High Performance Computing: Challenges and Opportunities. arXiv preprint arXiv:2203.02544.

[2] Schulte, M. J., Ignatowski, M., Loh, G. H., Beckmann, B. M., Brantley, W. C., Gurumurthi, S., ... & Rodgers, G. (2015). Achieving exascale capabilities through heterogeneous computing. IEEE Micro, 35(4), 26-36.

[3] Qasaimeh, M., Denolf, K., Lo, J., Vissers, K., Zambreno, J., & Jones, P. H. (2019, June). Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels. In 2019 IEEE international conference on embedded software and systems (ICESS) (pp. 1-8). IEEE.

[4] Zhu, Z., Liu, A. X., Zhang, F., & Chen, F. (2018). FPGA resource pooling in cloud computing. IEEE Transactions on Cloud Computing, 9(2), 610-626.

[5] Nane, R., Sima, V. M., Pilato, C., Choi, J., Fort, B., Canis, A., ... & Bertels, K. (2015). A survey and evaluation of FPGA high-level synthesis tools. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 35(10), 1591-1604.

[6] "Xilinx - Adaptable. Intelligent." Xilinx.com, 2022, www.xilinx.com

[7] Diamantopoulos, D., Xydis, S., Siozios, K., & Soudris, D. (2015). Mitigating memory-induced dark silicon in many-accelerator architectures. IEEE Computer Architecture Letters, 14(2), 136-139.

[8] Liang, T., Zhao, J., Feng, L., Sinha, S., & Zhang, W. (2018). HI-DMM: High-performance dynamic memory management in high-level synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(11), 2555-2566.

[9] Diamantopoulos, D., Xydis, S., Siozios, K., & Soudris, D. (2015, April). Dynamic memory management in vivado-hls for scalable many-accelerator architectures. In International Symposium on Applied Reconfigurable Computing (pp. 117-128). Springer, Cham.

[10] Xue, Z., & Thomas, D. B. (2015, September). SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems. In 2015 25th International Conference on Field Programmable Logic and Applications (FPL) (pp. 1-7). IEEE.

[11] Giamblanco, N. V., & Anderson, J. H. (2019, September). A dynamic memory allocation library for high-level synthesis. In 2019 29th International Conference on Field Programmable Logic and Applications (FPL) (pp. 314-320). IEEE.

[12] Xue, Z., & Thomas, D. B. (2016, May). Synadt: Dynamic data structures in high level synthesis. In 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (pp. 64-71). IEEE.

[13] Dessouky, G., Klaiber, M. J., Bailey, D. G., & Simon, S. (2014, September). Adaptive Dynamic On-chip Memory Management for FPGA-based reconfigurable architectures. In 2014 24th International Conference on Field Programmable Logic and Applications (FPL) (pp. 1-8). IEEE.

[14] Özer, C. (2014). A dynamic memory manager for FPGA applications (Master's thesis, Middle East Technical University).

[15] Lea, D., & Gloger, W. (1996). A memory allocator.

[16] Peterson, J. L., & Norman, T. A. (1977). Buddy systems. Communications of the ACM, 20(6), 421-431.

[17] Xilinx "Vitis High-Level Synthesis User Guide (UG1399), 2021

[18] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., ... & Czajkowski, T. (2011, February). LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (pp. 33-36).

[19] Jones, R., Hosking, A., & Moss, E. (2016). The garbage collection handbook: the art of automatic memory management. CRC Press.

[20] Shan, J., Casu, M. R., Cortadella, J., Lavagno, L., & Lazarescu, M. T. (2019, June). Exact and heuristic allocation of multi-kernel applications to multi-FPGA platforms. In Proceedings of the 56th Annual Design Automation Conference 2019 (pp. 1-6).