Memory Mapping for Multi-Die FPGAs

Nils Voss*†, Pablo Quintana†, Oskar Mencer†, Wayne Luk* and Georgi Gaydadjiev*†
*Department of Computing, Imperial College London, UK
Email: nv916@ic.ac.uk, w.luk@imperial.ac.uk, g.gaydadjiev@imperial.ac.uk
†Maxeler Technologies, London, UK

***Abstract*—This paper proposes an algorithm for mapping logical to physical memory resources on Field-Programmable Gate Arrays (FPGAs). Our greedy strategy based algorithm is specifically designed to facilitate timing closure on modern multi-die FPGAs for static-dataflow accelerators utilising most of the on-chip resources. The main objective of the proposed algorithm is to ensure that specific sub-parts of the design under consideration can fully reside within a single die to limit inter-die communication. The above is achieved by performing the memory mapping for each sub-part of the design separately while keeping allocation of the available physical resources balanced. As a result the number of inter-die connections is reduced on average by 50% compared to an algorithm targeting minimal area usage for real, complex applications using most of the on-chip's resources. Additionally, our algorithm is the only one out of the four evaluated approaches which successfully produces place and route results for all 33 applications and benchmarks.**

***Index Terms*—Memory Allocation, Memory Mapping, Greedy Heuristic, FPGA, Multi Die**

I. Introduction

The recent trends of big data, cloud computing and machine learning call for significant increases in compute power and energy efficiency beyond what current general purpose computers are able to provide. Furthermore, with CMOS technology feature sizes approaching interatomic distances, highly heterogeneous systems which can outperform state-of-the-art Central Processing Units (CPUs) in both performance and energy efficiency have been considered as a valid alternative.

While the usage of General Purpose Graphics Processing Units (GPGPUs) is becoming more and more common practice, alternative approaches make use of Application-Specific Integrated Circuits (ASICs), like Google's Tensor Processing Unit (TPU) [1], True North by IBM [2], specialised coarse grain reconfigurable devices [3] or FPGAs. The use of FPGAs is of special interest, since they can provide better energy efficiency and performance than GPGPUs and especially CPUs [4]–[8], while offering higher flexibility and significantly lower development costs compared to ASICs. As a result multiple industry vendors adopted FPGAs in their high performance computing environments. One of the most notable efforts in this direction is the general availability of FPGA-based cloud instances on the Amazon AWS EC2 cloud [9]. Additionally, Microsoft uses FPGAs in their Azure cloud [10] and some server vendors, like Dell, offer servers with FPGA support [11]. This is further emphasised by Xilinx's current focus on the datacenter, the driver behind the development of the Alveo accelerator cards [12]. However, there are still many unresolved challenges around the large scale deployment and usage of FPGA based systems. Most notable, efficient programming FPGAs and migrating existing designs from one FPGA generation to the next is often not straight forward and might require considerable time and resources. As a result FPGA usage for High-Performance Computing (HPC) is still far from true large-scale adoption.

To alleviate the programmability challenge many different programming frameworks, paradigms and languages exist. The implementation is typically described at a high level and then automatically instantiated on the available FPGA hardware resources. For example, a logical memory defined by the programmer is automatically mapped to the available physical resources by the High-Level Synthesis (HLS) tool.

The automatic mapping of logical into physical hardware resources has become a lot more challenging with modern FPGA devices, especially in the case of memory resources. The reason for this is twofold. Intel as well as Xilinx have introduced additional memory types with their latest chips [13], increasing the diversity of the available memory resources an HLS tool has to manage. Additionally, Xilinx' biggest FPGAs consist of multiple silicon dies on the same interposer, also known as Super Logic Regions (SLRs) [14]. One example of such a device is the VU9P which also powers the Amazon EC2 F1 instances. In order to facilitate timing closure, it is beneficial when a hardware structure described by the designer can fully reside within a single SLR to avoid slow and scarce inter-SLR connections. Since the number of individual memory resources within each SLR is limited, efficient logical to physical memory mapping is important. The reason for this is that over allocation of a specific memory resource, e.g., allocating more resources than available within a single SLR, will automatically lead to a design in which the hardware structures span across multiple SLRs. Such SLR crossing will hinder timing closure and often produce slower designs. As a result, it is important for an HLS tool to allocate different memory resources so that the programmed structure resides as much as possible within a single SLR.

This paper proposes a greedy algorithm which automatically allocates hardware memory resources for user defined memories, considering modern technology trends like SLRs and increasingly heterogenous on-chip memory resources. The aim of our algorithm is to balance the allocation of different memory resources for individual sub-parts of the design and minimise the number of inter-SLR connections, which will facilitate timing closure and reduce routing congestion. We target the latest Xilinx technology, however, the algorithm is generally applicable to systems with similar properties.

The main contributions of this paper are as follows:

- A simple greedy Balanced Memory Mapping (BMM) algorithm, which optimises timing closure by reducing the number of SLR crossings;
- Two additional mapping strategies, Threshold Based Memory Mapping (TBM2) and Wastage Reducing Memory Mapping (WRM2), to provide a realistic comparison baseline, inspired by previous work and circumventing industry tools shortcomings;
- Careful evaluation of the proposed algorithms based on a set of use cases for small, medium and large workloads.

The remainder of the paper is organised as follows. Section II provides a background overview and section III discusses related work. In section IV the proposed algorithm is presented. The evaluation of this algorithm is given in section V. Finally section VI concludes the paper.

II. Background

A. SLRs

In order to increase chip area, while keeping yield and production costs in check Xilinx has introduced SLRs in recent FPGA generations. This is achieved by a technology called Stacked Silicon Interconnect (SSI) by Xilinx, where multiple FPGA dies are mounted on a single silicon interposer [14].

As a result the inter-die communication can only be performed using a limited number of slower wires on the silicon interposer. For example the Xilinx VU9P has just above 20,000 inter-die connections in total. However, SLR crossings are only available between neighbouring SLRs. As a result routing between different and especially non adjacent SLRs is challenging and requires special attention.

B. Memory Resources

The Xilinx UltraScale FPGAs contain three different physical memory types [15], [16]:

1) Distributed RAM;
2) BlockRAM;
3) UltraRAM.

One logic slice of the Xilinx Ultrascale FPGAs contains eight 6-input Look Up Tables (LUTs) that are used to construct a single 512 bits distributed RAM. In the Xilinx documentation this is referred to as SLICEM. Multiple SLICEM can be combined together to form deeper memories, however, this comes with a significant overhead. Individual SLICEMs can be tiled in a multitude of different configurations in terms of depth, width and number of read and write ports.

BlockRAM (BRAM) modules are separate physical hardware memory units. Each BRAM of the UltraScale architecture can store up to 36 Kbits of data and can be used as a single or two independent memory units. In both cases they consist of two read and two write ports and it is possible to tile them into different depth and width configurations. As an example a 68 bit wide and 850 deep single port logical memory would occupy two BRAM modules with a tiling of 36x1024. The number of tiling options is further increased considering the number of supported read and write port combinations.

Finally, the UltraRAMs (URAMs) represent an additional dedicated memory resource. One URAM module can store up to 288 Kbits of data, but has only one single write and one read port. Additionally, URAMs can be used only as 72 bit wide and 4,096 deep memories and some specific functions, e.g., dual clock FIFO implementations, are not supported. To summarise, URAMs are the least flexible from all available memory types but usually contribute most to the overall on-chip memory capacity of the Xilinx Ultrascale devices.

III. Related Work

In [17] the authors present an algorithm which maps logical memories to shared physical memories, by taking advantage of dual port functionality. The algorithm tries to tile logical memories and reduce resource wastage by letting two logical single port memories share the same physical dual port memory. Our algorithm considers the effects of multi SLR platforms with more heterogenous memory resources. Moreover, this specific or similar optimisations can be easily included.

In [18] a technique to improve energy efficiency is presented. The power usage can be reduced by disabling the clock enable in cases where a memory is idle. Additionally logical memories are allocated purely based on their depth. Our algorithm uses a more advanced method to choose between physical memories considering width and depth. The proposed power optimisations are orthogonal to our algorithm.

To our knowledge there is no previous work on memory allocation, which takes SLRs into consideration, however, multi chip partitioning algorithms like [19]–[21] have a similar optimisation target. The above algorithms partition the complete design, whereas in this work only the mapping of memories to different physical resource types ignoring other components is discussed. These decisions provide a preprocessing stage to multi chip partitioning algorithms. In our case it is possible to ignore other resource types, since the penalty of moving data between SLRs is still order of magnitudes smaller than between different FPGAs.

In contributions like [22] multiple algorithms and tools are used to perform design space exploration across multiple HLS pragmas. In this work we focus on a single design space dimension in the context of multi-die FPGAs and aim at finding a reasonable solution within it by using a simple algorithm with the main optimisation criterion being balancing the memory allocation rates between different resources.

IV. Algorithm Description

Generally a good memory mapping algorithm should:

1) use as few resources as possible; and
2) facilitate timing closure.

In order to address the first objective the utilisation of the available memory resources needs to be considered. As shown in eq. 1, the utilisation of a given physical memory resource (BRAM or URAM) is the maximum utilisation of its valid tilings. The utilisation of each tiling is the ratio between the user defined logical memory size and the product of the physical memory unit size (both in #bits) and the required number of physical memories. To minimise hardware wastage the memory type with the best utilisation is selected.

$$\max\left(\frac{logical\ memorysize}{unitsize * \#units}\right)_{tiling=1}^{N} \quad (1)$$

We call a group of hardware resources with high interconnectivity placed in close proximity on the FPGA fabric a *design unit*. Design units can be specified explicitly by the user, implicitly by using language structures or by creating a high-level floorplan model. As such a design unit could, for example, contain all resources of a single computational unit, e.g., a kernel. Alternatively, if a floorplan is used, a design unit could also contain all the resources which are mapped to a certain SLR. In order to reduce SLR crossings and as a result aid timing closure, it is of major concern to balance the allocation of memory resources between different design units. Consider a case where a specific design unit requires more BRAMs than a single SLR capacity. As a result BRAMs from neighbouring SLRs have to be allocated increasing SLR crossings and routing congestion. As such redirecting some of the memories to URAMs is beneficial even though this introduces overheads in terms of allocated memory bits. In short, balanced allocation between BRAMs and URAMs is expected to improve SLR locality of individual design units. In the authors experience SLR crossings and the related routing congestions limit timing closure. Our experience is based on thousands of place and route attempts for dozens of real applications on the Xilinx VU9P. As a result the major goal for a timing optimised multi-die aware memory mapping algorithm is avoidance of unnecessary SLR crossings.

To address the above we propose the following Balanced Memory Mapping (BMM) algorithm. BMM runs per design unit and its input is the list of logical memories. In few cases a logical memory can only be mapped to a specific physical memory, due to specific hardware features, e.g., dual clock domain support. Such special logical memories are handled first to ensure mapping to the appropriate hardware resources.

Afterwards another preprocessing stage decides which of the remaining logical memories should be mapped to distributed RAM. Since the logic resources used to implemented distributed RAM are less scarce, this mapping can be based on a simple heuristic and does not need to consider SLRs. This heuristic is based on the BRAM utilisation calculated using eq. 1. It is not needed to test URAM utilisation, since URAMs are significantly larger than BRAMs and because of the tiling restrictions will never achieve a better utilisation for a given logical memory than BRAMs. The decision on mapping to distributed RAM uses a simple BRAM threshold. When logical memory BRAM utilisation is lower than $1/8$ it will be mapped to distributed RAM. Otherwise, the algorithm decides between URAMs and BRAMs on a later stage. The BRAM threshold value was deduced by considering the BRAM tiling with the smallest possible depth of 512 and a width of 36 bits. The above datapath width was chosen based on our experience that 18 bits is typically not sufficient for small memories implemented in distributed RAM. If we consider a 36 bits wide logical memory matching BRAM widths, it will be mapped to distributed RAM when its depth is 64 or less. It should be noted that a depth of 64 matches a distributed RAM tiling. Comparing utilisation provides a simple metric which accounts for both, width and depth. The reason for this low utilisation requirement is that the amount of distributed RAM available is very small in comparison to the other memory resources. Additionally logic resources are also used to implement arithmetic and other user design features. As such saving logic resources is in general considered beneficial. However, in the context of static dataflow designs with many shallow FIFOs we have to use distributed RAM. For those FIFOs BRAM utilisation will be very low.

When beneficial, BRAM threshold value can be customised for each specific design depending on the overall logic utilisation. For example, when a design requires an exceptional amount of logic resources it is possible to skew the balance towards BRAMs by decreasing the threshold. As a result BRAM threshold best value is application specific; however, the setting used here achieved good results for all use cases.

After the first memory mapping stage all memories not mapped to distributed RAM are grouped by design units. Design units' memories are stored in a global list, sorted by decreasing URAM utilisation. This list is used by the memory allocation algorithm to decide if a particular logical memory should be implemented as BRAMs or URAMs.

Fig. 1 shows the BMM algorithm in more detail. The algorithm uses a score, which is initialised to zero and is used to decide in which order the memories are allocated. The score is based on BRAM cost and used to track of the balance between BRAMs and URAMs. As a result when the algorithm selects to map a given logical memory to BRAMs the score is increased by the number of BRAMs allocated. When URAMs are selected, the score will be decreased as explained next. Since the score is based on BRAM cost we need to find a factor relating BRAM to URAM cost. This is achieved by using the ratio between the available BRAM and URAM modules. Consequently, if a logical memory is mapped to URAMs the score will be decreased by the product of this ratio and the number of URAMs needed to implement the logical memory resource. This procedure is repeated for each design unit until the corresponding list of unmapped memories is emptied. As a result our algorithm will perform best in the case of single design unit per SLR. This will avoid segmentations and the consequent suboptimal mapping resulting in slightly increased memory utilisation. However, when the HLS toolflow does not support floor planning, the above is highly unlikely and resources have to be grouped into design units based on other properties. The evaluation presented here assumes the less advantageous later option, where design units are implicitly inferred by language structures.

Within the mapping loop of the BMM algorithm there are three possible cases which are handled separately. If the score is close to zero the allocation between BRAMs and URAMs is

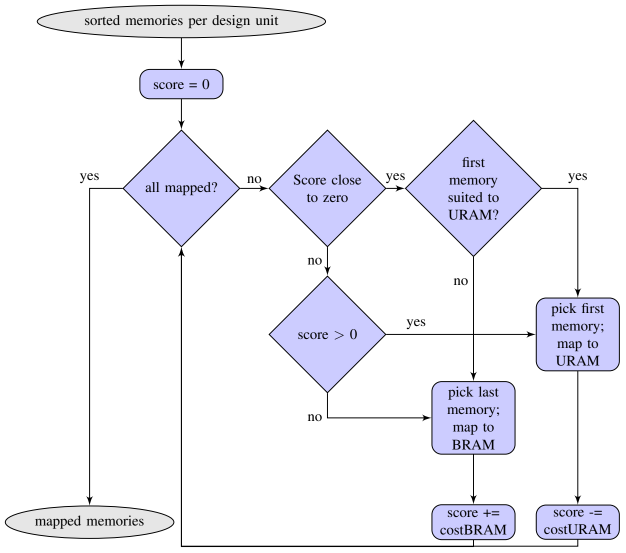
sorted memories per design unit
score = 0
all mapped?
yes
no
Score close to zero
yes
no
first memory suited to URAM?
yes
no
score > 0
yes
no
pick first memory; map to URAM
pick last memory; map to BRAM
score += costBRAM
score -= costURAM
mapped memories

Fig. 1. BMM algorithm with greedy, score-based proportional mapping

considered as balanced. In this case the next unmapped logical memory is picked from the beginning of the list. Since the list is ordered by URAM utilisation, highest URAM suitability of the selected unmapped logical memory is ensured. The memory will be actually mapped to URAMs only when its utilisation is bigger than the URAM threshold. Otherwise a new memory from the end of the list, hence suited to BRAMs, is selected and mapped to BRAMs. This URAM threshold together with the score enforce when a logical memory is mapped to URAMs and can be modified by the designer. In the authors experience a URAM threshold of 0.6 provides good results. This means that logical memories with at least 60% URAM utilisation are directly mapped. The URAM threshold ensures that design units with only a single logical memory benefit from the best suited physical memory resource.

It should be mentioned that the URAM threshold in most cases has a very limited impact, due to the algorithm capability to balance allocation between BRAMs and URAMs within the same design unit. However, the URAM threshold becomes important for certain rare edge cases. These consist of designs composed of multiple design units with only one or two memories of significant size. The default value of 0.6 tries to find a good balance with the target of decreasing the overall memory wastage. However it might be necessary to adapt the URAM threshold manually in those rare edge cases.

In the case of a positive score more BRAMs than URAMs are used and again the logical memory from the beginning of the list is mapped to URAMs without considering the URAM threshold. Finally, in the last case of a negative score a memory from the list end will be mapped to BRAMs.

Each time a memory is mapped to a specific resource the score is adjusted as described above. It will be increased in the case of BRAM mapping or decreased in the case of URAMs. As a result BRAMs and URAMs are allocated at a comparable rates with respect to the overall availability.

By ordering the memories based on URAM utilisation it is ensured that always the memories most suited to URAMs or BRAMs are mapped first. This greedy strategy ensures that as many memories as possible are mapped to their best suited physical memory resource therefore saving area and addressing the first objective of the algorithm. To further improve memory resources utilisation it is possible to combine our algorithm with already existing memory allocation approaches, e.g., by tiling logical memories and making use of dual port memories as in [17]. However, it is necessary that all additional optimisation algorithms do not disturb accurate estimation of the number of physical memory resources that have to be allocated. Additionally, the second algorithm objective is fulfilled by allocating memories to URAMs and BRAMs at the same rate and hence minimising SLR crossings, which causes routing congestion reduction and aid timing closure.

V. Evaluation

In order to evaluate the proposed BMM algorithm we perform place and route on multiple designs for the Xilinx VU9P FPGA using Vivado 2017.4. In all cases Maxeler's MaxCompiler is used and only the memory selection is influenced to enforce VHDL with the desired memory macros instantiated. As a result, when the designs satisfy the same timing constraints, the achieved throughput remains the same as well as the logic and arithmetic resources utilisation. For all designs we use the set of implementation strategies able to achieve the best results in meeting a specific frequency. In cases where designs could not fit on the chip, we report the synthesis results on area utilisation to emphasise the reason why designs failed to fit. For all experimets we use the URAM and BRAM thresholds suggested in the last section. In section V-A the memory mapping algorithms used to compare against are described. Section V-B introduces the test cases used and V-C provides the experimental results.

A. Algorithms

To evaluate the proposed BMM algorithm, three other mapping algorithms are used in our comparison below.

In the first case all mapping decisions are left to the Xilinx Vivado toolchain by using the *XPM_MEMORY* core and setting the *memory_style* to *auto*. Vivado, however, currently only uses distributed RAM and BRAMs [23]. Since URAMs are not yet supported by Vivado for the sake of fair comparison we introduce two additional algorithms. They both implement traditional memory mapping approaches and extend them with URAM support to form a realistic comparison base line.

The first additional algorithm, called Threshold Based Memory Mapping (TBM2), uses the same mapping to distributed RAM as BMM, but it simply uses the same fixed URAM threshold as in BMM to decide which memories should be mapped to URAMs or to BRAMs. As a result, if the URAM utilisation for a logical memory is above 0.6 it will be mapped to URAMs otherwise BRAMs will be selected. This algorithm implements standard techniques which, for example, only consider the depth of a logical memory as in [18]. It aims only at efficient utilisation of each individual physical resource.

Additionally, we introduce the Wastage Reducing Memory Mapping (WRM2) algorithm. This third and final algorithm, as depicted in fig. 2 improves on the TBM2, by alleviating over usage of a single resource when other memory resources are still available. First, it uses the same mapping to distributed RAM as described before. Next, all remaining unmapped memories are ordered by URAM suitability as with BMM. In contrast to the BMM this ordered logical memory resources list is global and not on a per design unit basis.

The memories on that list are then allocated using the same URAM threshold of 0.6 as for the previous algorithms. In addition, WRM2 also keeps track on how much of the available resources are already allocated. When more than 80% of one physical memory resource is allocated, the remaining memories will be mapped to the other resource type. If both resources exceed 80% this limit will be increased in steps of 10%. For example, when 80% of BRAMs are used, the algorithm will only map to URAMs until the overall URAM usage also exceeds 80%. Due to ordering by suitability, when mapping to URAMs, the algorithm will only pick memories from the front of the list and consequently, when mapping to BRAMs it consider memories from the back. As a result, this algorithm tries to maximise physical memory utilisation and always aims at mapping logical memories to the best suited physical memory resource while not over-allocating resources. This approach allows us to study the area overhead introduced by the proposed BMM algorithm.

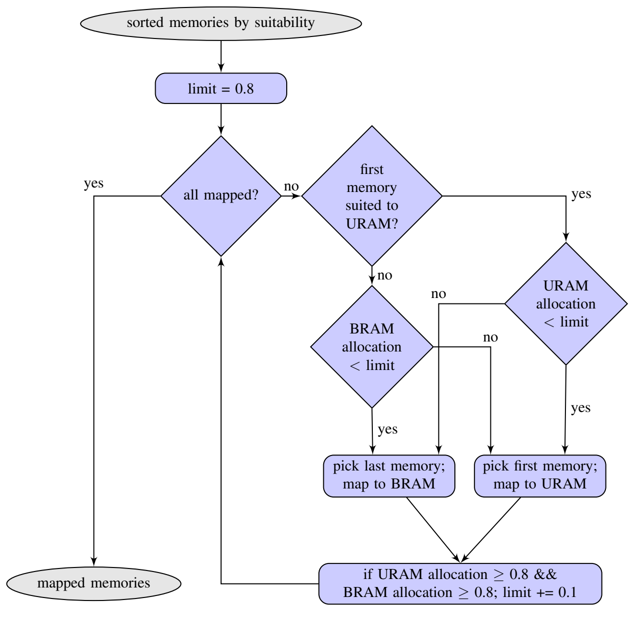
sorted memories by suitability
limit = 0.8
all mapped?
yes
no
first memory suited to URAM?
yes
no
URAM allocation < limit
no
BRAM allocation < limit
no
yes
yes
pick last memory; map to BRAM
pick first memory; map to URAM
mapped memories
if URAM allocation ≥ 0.8 && BRAM allocation ≥ 0.8; limit += 0.1

Fig. 2. WRM2 algorithm with greedy, global area optimised mapping

To the best of our knowledge no alternative multi-die aware memory mapping algorithm exists with the target to facilitate timing closure that can be used for direct comparison.

B. Test Cases

The four algorithms in our study are applied to three different sets of use cases. The first set consists of 16 small benchmarks (S1-S16), which only occupy a small area on the VU9P and can comfortably reside within a single SLR.

The second set of use cases consists of eight different medium sized designs (M1-M8). These examples occupy more than a single SLR, but still do not fully fill up the VU9P chip. M1-M5 are small synthetic examples, while M6-M8 are different versions of an FPGA based implementation of a real application, SPECFEM3D [24]. SPECFEM3D is a widely used HPC workload, which simulates different geophysical events, like wave propagation through different materials.

Finally, the last set of use cases consists of nine real applications (L1-L9). These applications represent real HPC workloads, which typically use most of the available on-chip resources, span multiple SLRs and make extensive use of the PCIe and DDR interfaces and hence decrease the overall number of available BRAMs and URAMs.

L1-L5 are machine learning applications. L1 is a fully connected network, while L2-L5 implement two convolutional neural networks with and without Winograd transform and all incorporate three copies of the same network implementation. L2 and L3 do not use Winograd and differ only in enforcing the design copies to specific SLRs or not. L4 and L5 use Winograd and follow L2 and L3 in their placement constraints.

L6 is a nanoscale material simulation application called Quantum ESPRESSO [25] another widely used HPC workload. L7 is an FPGA implementation of BQCD [26] a quantum chromodynamics application. L8 implements the ocean engine of NEMO [27] a commonly used weather simulation tool. Lastly, L9 is a dense matrix matrix multiplication[1], a main building block of many HPC applications.

The smaller synthetic test cases are included to study how the proposed algorithm behaves for smaller applications which do not make use of all on-chip resources even though this was not its typical optimisation target. The use cases M6-M8 as well as L1-L8 are real world applications which are deployed in HPC environments and as a result represent the primary target for the design of the presented algorithm. Especially M6-M8 as well as L6-L8 represent a significant portion of the workloads running currently on HPC systems.

C. Results

Fig. 3 and fig. 4 show the BRAM and URAM usage for the small use cases. It can be observed that all algorithms apart from BMM use the exact same number of BRAMs and URAMs. Since the BMM algorithm tries to find a balance between allocating BRAMs and URAMs, it maps some of the logical memories to URAMs. As a result the overall memory usage in allocated bits is increased by 38%. As this may seem quite high, the total number of allocated memory blocks and therefore the expected power consumption stay close. All four algorithms allocate the same amount of distributed RAM.

[1]https://github.com/nilsv/Dense-Matrix-Multiplication

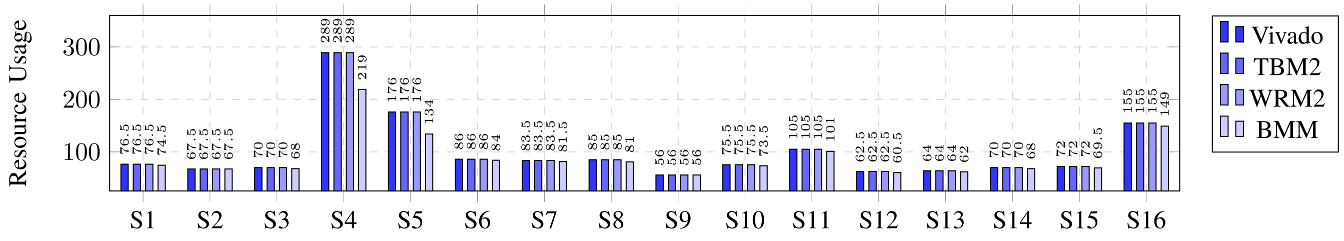

Fig. 3. BRAM usage for the four different algorithms on the test set of small applications.

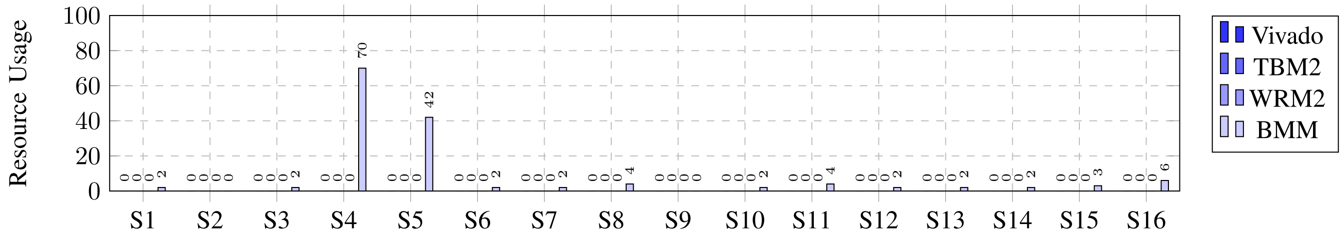

Fig. 4. URAM usage for the four different algorithms on the test set of small applications.

The memory resource usage for the set of medium sized use cases is shown in fig. 5 and 6. The standard Vivado algorithm does not map any logical memories to URAMs. As a result M8 does not fit into the chip, due to a significant overallocation of distributed RAM. The simple TBM2 algorithm and WRM2 both reached the same mapping decisions, since no single resource exceeded 80%. The proposed BMM algorithm again uses URAMs and BRAMs more balanced than the other algorithms. As a result the overall number of allocated bits is increased by 85% compared to the WRM2 algorithm. However, it should be noted that using large amounts of a particular resource usually makes timing closure harder. The allocation of distributed RAM is very similar between all algorithms. Only for the use case of M8 the standard Vivado algorithm allocates significantly more distributed RAM.

Finally, the resource usage for the large test cases is shown in fig. 7 and 8. Vivado and the simple TBM2 algorithm both fail to place and route test cases L2, L3, L6, L8 and L9. Additionally, the high resource usage for the standard Vivado algorithm prevents successful place and route of L7.

L2 and L3 designs instantiate three copies of the same large design unit. In the case of L2, Vivado placement constraints are used to force each design unit in a single SLR, while in L3 has no placement constraints. The WRM2 algorithm has difficulties with designs like the above. It maps one of the three design units mainly to BRAMs and the other two mainly to URAMs. This causes all three design units to span across multiple SLRs. As a result L2 fails to meet its placement constraints and L3 fails due to very high routing congestion, even though the total count of allocated BRAMs and URAMs is similar between WRM2 and the proposed BMM algorithm. Only the latter is able to find a mapping, facilitating successful place and route completion.

L9 provides a similar test case as above consisting of three individual design units. WRM2 again allocates resources very unevenly between the three design units, mapping one entirely to URAMs and the other two mostly to BRAMs. However, the mapped workload is less complex making successfully place and route possible, even though the number of SLR crossings is increased by a factor of 2.6.

The mapping behaviour for designs with multiple big design units was the main motivation behind the proposed BMM algorithm. Since both memory resources are allocated at the same rate, it is guaranteed that no single resource is heavily overused. Only when the overall memory usage of a design unit is larger than a single SLR capacity, multiple SLRs will be used. This ensures that the toolchain does not limit place and route of valid, well designed architectures.

In the case of L8 the WRM2 algorithm also fails to facilitate successful place and route completion. Here WRM2 makes use of all available URAMs, which leads to a violation of a placement constraint introduced by the Xilinx DDR IP core causing place and route to fail. This could be potentially avoided by predicting the BRAM usage more accurately, since only 85% of BRAMs are used. In general, we noticed that WRM2 requires precise estimations in order to avoid overallocation of a single resource.

To summarise, only the proposed BMM algorithm manages to successfully generate place and route results for all large designs, even though the memory usage in number of bits is on average 13% higher. However, in cases where the memory usage of both physical resources approaches levels above 80% the BMM and the WRM2 algorithm both allocate a similar amount of BRAMs and URAMs.

Fig. 9 shows the average number of SLR crossings across multiple implementation strategies for all medium and large use cases and all four mapping algorithms.

In order to compare the average number of SLR crossings

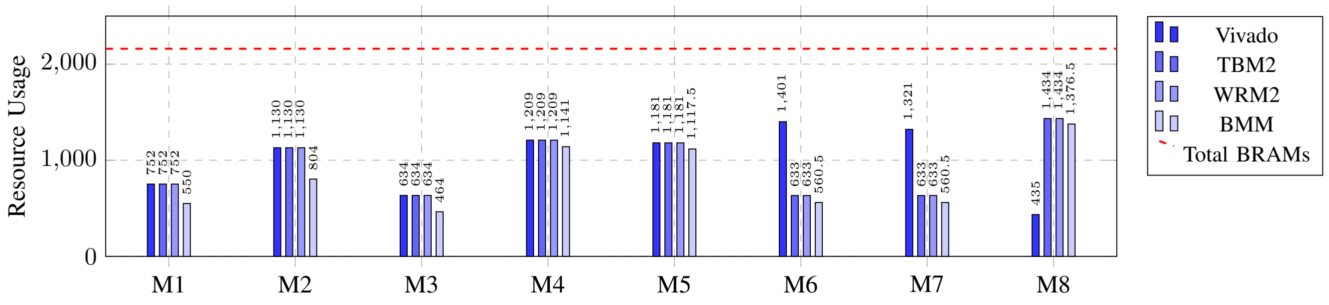

Fig. 5. BRAM usage for the four different algortithms on the test set of medium applications

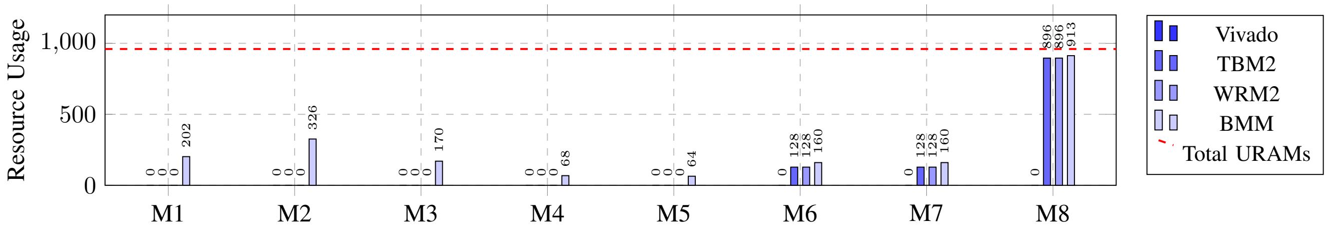

Fig. 6. URAM usage for the four different algortithms on the test set of medium applications

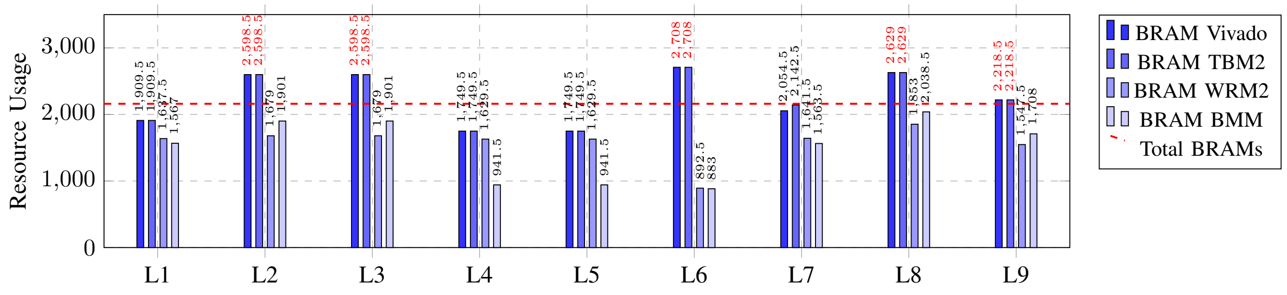

Fig. 7. BRAM usage for the four different algortithms on the test set of large applications

between algorithms, we considered only those cases where place and route finished successfully since this is necessary to obtain a figure on SLR crossings. This means that cases in which those algorithms performed especially poorly are not taken into account, creating a bias against the proposed BMM algorithm. In general, there is no straight forward way to include the test cases currently not taken into account. One could for example assume that in the not routable cases all SLR crossings are fully used, which is not realistic and would create unnecessary bias in favour of the proposed BMM. However, even the used less advantageous test case selection shows the advantages of the proposed algorithm.

For the medium use cases the proposed BMM algorithm achieves a reduction in the average number of SLR crossings by 46%, 11% and 6% compared to the standard Vivado, the TBM2 and the WRM2 algorithms respectively. For the three large cases successfully routed by Vivado and BMM on average the same number of SLR crossings were created. However, in the six other test cases Vivado failed to finish place and route. In comparison to TBM2 and WRM2 the usage of the BMM algorithm leads to an average reduction in number of SLR crossings by 7% and 52% respectively.

Lastly, fig. 10 shows the Total Negative Slack (TNS) for test cases in which at least one algorithm produced a TNS value while failing timing closure. Additionally, at least two algorithms produce TNS values to facilitate comparison. In the case where a design is very congested TNS values are often not generated by Vivado. As a result it is only possible to draw meaningful conclusions from the five test cases shown in the figure. The depicted TNS is the average over multiple implementation strategies.

By comparing fig. 9 and fig. 10 one can observe that there is a strong correlation (not a causation) between the number of SLR crossings and the average TNS. For example in the case of M6 the standard Vivado algorithm creates by far the most SLR crossings, resulting in the worst average TNS. Accordingly, BMM creates the least amount of SLR crossings and achieves the lowest TNS.

The same correlation holds true for L1 and L4. However, in the case of L7 BMM creates the highest average TNS even

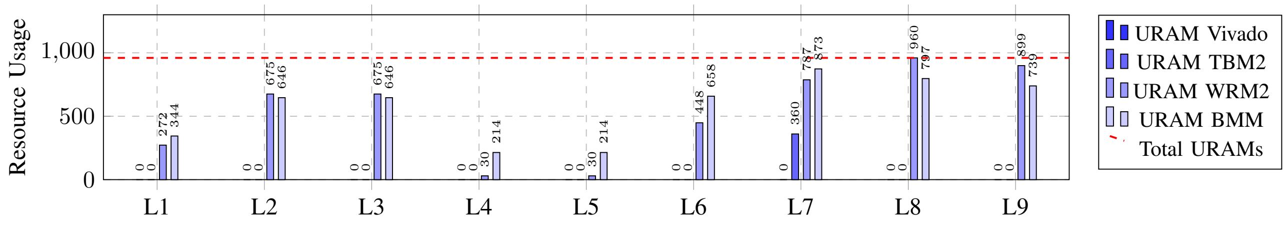

Fig. 8. URAM usage for the four different algortithms on the test set of large applications

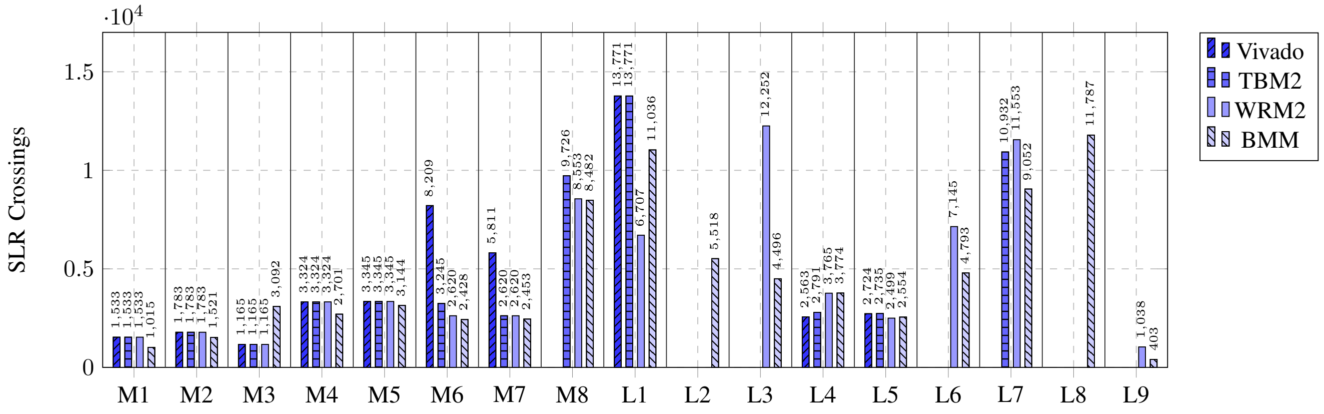

Fig. 9. Number of SLR crossings for the different mapping algorithms on the medium and large test set. Missing bars corresponds to failed builds.

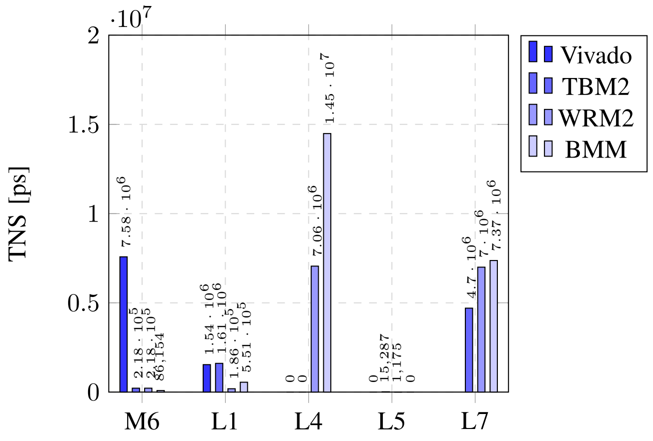

Fig. 10. Average TNS for test cases, where more than one algorithm produced a non zero TNS. Missing bars corresponds to failed builds.

though the number of SLRs crossings is the lowest. The reason for this is that for one of the implementation strategies the TNS is 50,114,977 ps. This single outlier impacts average TNS significantly. If this implementation strategy is excluded the average drops to 1,265,720 ps, which would be the lowest average TNS for this use case.

As a result we conclude that comparing the number of SLR crossings is a good metric to estimate the impact of memory mapping algorithms on TNS. The advantage of this is, that it provides an easier to compare metric, generating more data points in a smaller value range, whereas TNS can be very prone to outliers or other design properties. Similarly, it is hard to take into account if for a few implementation strategies no TNS value is generated, since the design can not be routed.

VI. Conclusion and Future Work

In this paper we presented the Balanced Memory Mapping (BMM) algorithm, which allocates logical to physical memories. The proposed algorithm aims at balancing allocation between different physical memory resources in partitioned large designs, to facilitate locality in multi-die FPGAs. Our proposal was compared against three memory mapping algorithms representing different optimisation goals and commonly used methods, including the standard Xilinx Vivado memory mapping algorithm, using 33 different use cases. Only our proposal managed to successfully produce place and route results for all test cases and managed to reduce the number of inter die connections by an average of 50% compared to the second best performing algorithm.

The proposed algorithm and thresholds values were shown to work well in the case of static dataflow applications. We believe our proposal is applicable to other FPGA design styles; however, the thresholds used in this work will have to be revisited. Future work includes automatic adjustment of thresholds based on area usage predictions as well as more precise BRAM usage predictions. Additionally, it would be of interest to study tiling of the same logical memory on multiple physical memory resources.

References

[1] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. ACM, 2017, pp. 1–12.

[2] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014. [Online]. Available: http://science.sciencemag.org/content/345/6197/668

[3] Wave Computing, "Wave Computing." [Online]. Available: https://wavecomp.ai

[4] L. Gan *et al.*, "Accelerating solvers for global atmospheric equations through mixed-precision data flow engine," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–6.

[5] O. Lindtjorn *et al.*, "Beyond traditional microprocessors for geoscience high-performance computing applications," *IEEE Micro*, vol. 31, no. 2, pp. 41–49, March 2011.

[6] G. C. T. Chow *et al.*, "A mixed precision monte carlo methodology for reconfigurable accelerator systems," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 57–66. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145705

[7] J. Arram *et al.*, "Hardware acceleration of genetic sequence alignment," in *Reconfigurable Computing: Architectures, Tools and Applications*, P. Brisk, J. G. de Figueiredo Coutinho, and P. C. Diniz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 13–24.

[8] C. Guo, H. Fu, and W. Luk, "A fully-pipelined expectation-maximization engine for gaussian mixture models," in *2012 International Conference on Field-Programmable Technology*, Dec 2012, pp. 182–189.

[9] Amazon, *Amazon F1 Instance*. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[10] Microsoft, *Inside the Microsoft FPGA-based configurable cloud*. [Online]. Available: https://azure.microsoft.com/en-gb/resources/videos/build-2017-inside-the-microsoft-fpga-based-configurable-cloud/

[11] ZDNet, *Intel FPGAs picked up by Dell EMC and Fujitsu*. [Online]. Available: https://www.zdnet.com/article/intel-fpgas-picked-up-by-dell-emc-and-fujitsu/

[12] Xilinx, *Xilinx ALVEO Adaptable Accelerator Cards for Data Center Workloads*. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo.html

[13] ——, *UltraScale+ FPGAs. Product Tables and Product Selection Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf

[14] ——, *Large FPGA Methodology Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/ug872_largefpga.pdf

[15] ——, *UltraScale Architecture Memory Resources. User Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf

[16] ——, *UltraScale Architecture Configurable Logic Block. User Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf

[17] W. K. C. Ho and S. J. E. Wilton, "Logical-to-physical memory mapping for fpgas with dual-port embedded arrays," in *Field Programmable Logic and Applications*, P. Lysaght, J. Irvine, and R. Hartenstein, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 111–123.

[18] R. Tessier *et al.*, "Power-efficient ram mapping algorithms for fpga embedded memory blocks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 278–290, Feb 2007.

[19] K. Roy and C. Sechen, "A timing driven N-way chip and multi-chip partitioner," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, Nov 1993, pp. 240–247.

[20] R. V. Cherabuddi and M. A. Bayoumi, "Automated system partitioning for synthesis of multi-chip modules," in *Proceedings of 4th Great Lakes Symposium on VLSI*, March 1994, pp. 15–20.

[21] F. Mao *et al.*, "Modular placement for interposer based multi-FPGA systems," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, May 2016, pp. 93–98.

[22] G. Zhong *et al.*, "Design space exploration of fpga-based accelerators with multi-level parallelism," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 1141–1146.

[23] Xilinx, *Vivado HLS: How can I infer UltraRAM in HLS?* [Online]. Available: https://www.xilinx.com/support/answers/71259.html

[24] D. Komatitsch *et al.*, *SPECFEM3D Cartesian v2.0.2 [software]. Available: https://geodynamics.org/cig/software/specfem3d/*, Computational Infrastructure for Geodynamics.

[25] P. Giannozzi *et al.*, "Advanced capabilities for materials modelling with quantum espresso," *Journal of Physics: Condensed Matter*, vol. 29, no. 46, p. 465901, 2017. [Online]. Available: http://stacks.iop.org/0953-8984/29/i=46/a=465901

[26] Y. Nakamura and H. Stüben, "BQCD - Berlin quantum chromodynamics program," vol. abs/1011.0199, 2014. [Online]. Available: https://arxiv.org/abs/1011.0199

[27] G. Madec, *NEMO Ocean engine. Note du Pôle de modélisation*. Institut Pierre-Simon Laplace (IPSL), 2008.