2020

Extracting Data-Level Parallelism in High-Level Synthesis for Reconfigurable Architectures

Juan Andres Escobedo Contreras

University of Central Florida

EXTRACTING DATA-LEVEL PARALLELISM IN HIGH-LEVEL SYNTHESIS FOR RECONFIGURABLE ARCHITECTURES

by

JUAN ESCOBEDO
M.S. University of Central Florida, 2019
B.S. Central University of Venezuela, 2013

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2020

Major Professor: Mingjie Lin

ABSTRACT

High-Level Synthesis (HLS) tools are a set of algorithms that allow programmers to obtain implementable Hardware Description Language (HDL) code from specifications written high-level, sequential languages such as C, C++, or Java. HLS has allowed programmers to code in their preferred language while still obtaining all the benefits hardware acceleration has to offer without them needing to be intimately familiar with the hardware platform of the accelerator.

In this work we summarize and expand upon several of our approaches to improve the automatic memory banking capabilities of HLS tools targeting reconfigurable architectures, namely Field-Programmable Gate Arrays or FPGA's. We explored several approaches to automatically find the optimal partition factor and a usable banking scheme for stencil kernels including a tessellation based approach using multiple families of hyperplanes to do the partitioning which was able to find a better banking factor than current state-of-the-art methods and a graph theory methodology that allowed us to mathematically prove the optimality of our banking solutions. For non-stencil kernels we relaxed some of the conditions in our graph-based model to propose a best-effort solution to arbitrarily reduce memory access conflicts (simultaneous accesses to the same memory bank). We also proposed a non-linear transformation using prime factorization to convert a small subset of non-stencil kernels into stencil memory accesses, allowing us to use all previous work in memory partition to them.

Our approaches were able to obtain better results than commercial tools and state-of-the-art algorithms in terms of reduced resource utilization and increased frequency of operation. We were also able to obtain better partition factors for some stencil kernels and usable baking schemes for non-stencil kernels with better performance than any applicable existing algorithm.

To my mother and my wife. Without your unconditional support, infinite love, and heavenly patience none of this would have been possible.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Mingjie Lin. Your motivation, support, and guidance were instrumental in this achievement. Your unwavering faith in my skills and abilities kept me inspired to push forward even in the hardest of times.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Problem background

Processing power has been increasing at a faster rate than memory access speeds and available bandwidth [4, 5]. This means the memory subsystems become a bottleneck in most architectures where the powerful and data-hungry processing units have to wait to get the information they need to operate which wastes valuable compute time and energy.

One model used to easily visualize the limitations set by the available memory bandwidth and compute power is the Roofline model [6,7] which plots a metric of performance (usually Giga/Tera Floating Point Operations per second or GFLOPS/TFLOPS) vs. the Operational intensity, or the number of arithmetic operations that can be performed on a single byte of data.

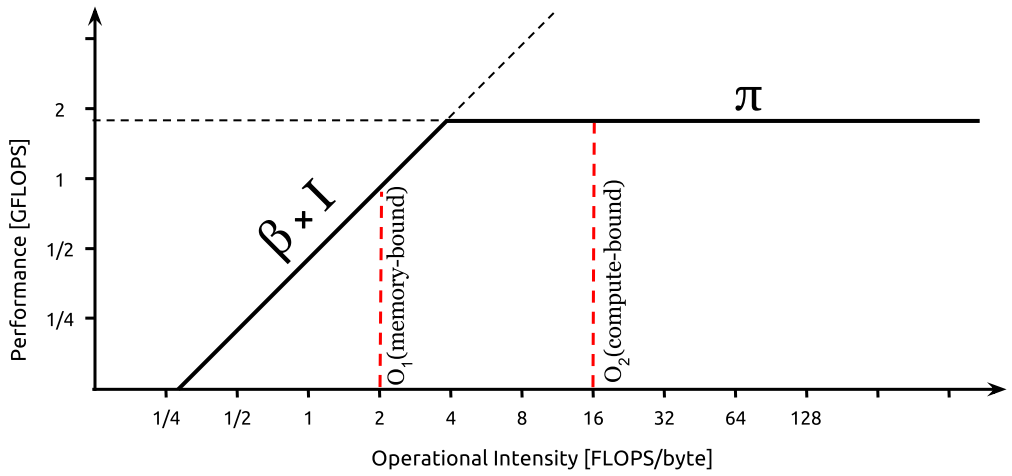


Figure 1.1: Performance upper bound using the Roofline model. Note the operation on the left is bound the available memory bandwidth while the operation on the right is only limited by the computing power available. By Giu.natale - Own work, CC BY-SA 4.0. https://commons.wikimedia.org/w/index. php?curid=49641351

In Figure 1.1 we see a plot using the Roofline model for 2 compute kernels. On the Y axis we find the performance metrics (GFLOPS in this case) while on the X axis we find the operational intensity, also known as arithmetic intensity, denoted as $\mathbf{I}$ and measured in FLOPS/byte. The operational intensity is a metric of how many arithmetic operations can we perform for each byte we fetch from memory. For kernels with low $\mathbf{I}$, such as $\mathbf{O_1}$, the peak performance will be bounded by the memory bandwidth β since we will require more data to complete the same amount of operations than a kernels on the right end of the spectrum. These kernels (such as $\mathbf{O_2}$) can perform many operations before requiring to access memory again, because of this, their performance is bound only by the maximum number of FLOPS the system can execute per second.

When a kernel's operating point is at the intersection of the performance bound set by the system and the memory bandwidth it finds itself at the optimal operation point since it is able to maximally

utilize the available memory bandwidth to achieve the maximum FLOPS of the system.

Memory bandwidth being a bottleneck has become increasingly more critical in modern computing. For example, in a multi-core chip, computing threads executing concurrently will inevitably contend with each other for the limited bandwidth of its main memory. Without advanced algorithms to deal with this issue, different threads will interfere with or block each other, thus severely degrading its overall computing performance. While this is a significant hurdle in traditional processor architectures, in reconfigurable computing the programmable logic chip provides an unique opportunity to customize application-specific digital circuits in order to achieve significantly higher computing performance. Even more interestingly, modern FPGA devices often contain tens of independent memory blocks capable of achieving a huge aggregated memory bandwidth.

To alleviate this problem, several techniques have been developed to improve concurrent access to the required data, of which memory partitioning and mapping algorithms have proven to be highly effective in a plethora of situation and thus have attracted the attention of the research community.

Fundamentally, the memory partitioning and mapping problem can be formulated as a memory access scheduling problem in both temporal and spatial dimensions. Ideally, memory partitioning and mapping schemes can not only maximize memory access performance but also incur lowest possible costs in logic usage and energy consumption. Given the significance of optimal memory partitioning and mapping, these have been extensive studies that aim at achieving fully-parallel memory accesses for a set of computing statements.

The goal of partitioning schemes is to allow parallel access to the data required by the computation every clock cycle, avoiding the need to serialize accesses to memory which might not have enough bandwidth to provide the data fast enough in order to keep the desired (or sometimes even required as it is the case of real-time applications) throughput.

Most modern processors and GPU's come with some partitioning and banking algorithms already built-in directly in the hardware, which allows for the execution of most pieces of code with acceptable performance. However, when performance requirements are increased and more specific or complicated memory partitioning and scheduling schemes are needed, it usually means dedicated hardware accelerators must be used. Programmers can find themselves in the need to design these accelerators at the Register-Transfer level (RTL) using Hardware Description Languages (HDL) such as Verilog and VHDL to control the most minute details of the functioning of the hardware and memory subsystem in order to ensure the desired levels of performance. The problem lies in the fact that designing efficient HDL code usually goes beyond the expertise of most seasoned programmers and is usually left to HDL experts with years of experience and intimate knowledge of the limitations and intricacies of the hardware platform where the accelerator is going to run on. This increases design times and costs since now an additional expert needs to be hired in order to complete the project and meet the specifications.

This is why High-Level Synthesis (HLS) has gained immense popularity in recent years. HLS are a set of software tools that compile high-level "soft" programs into efficient RTL "hard" specifications. This is of particular importance in today's computing world where FPGA devices become readily available in many heterogeneous computing systems such as Microsoft Azures servers. Among many well-known optimization techniques used in HLS, memory partitioning is probably one of the most studied and applied in order to improve performance and increase parallelism in synthesizing computing kernels. However, almost all of the previous HLS work strictly focuses on stencil-based computations, where the distance between memory accesses within each kernel iteration remains constant in its data domain. Unfortunately, stencil-based kernel computations are only but a subset of all the available code kernels widely used for scientific and general purpose applications. The case where the geometry of the memory accesses within each kernel iteration changes with time is known as non-stencil or sometimes referred to as irregular memory access.

Even with the help of sophisticated HLS tools efficient register-transfer level (RTL) ”hard” specifications still requires significant amount of ”tweaking” if the access performance of synthesized memory subsystems needs to be effectively optimized. As such, one central research topic in the high-level synthesis (HLS) of FPGA is how to automatically construct parallel memory access architectures and schemes that allow for simultaneous, conflict-free, accesses to all the data required for continuous executions.

Adding to the problem, most existing memory banking techniques, to the best of our knowledge, can not guarantee solution optimality, or a lower bound for partition factor, for all stencils. Therefore, several key questions remain to be answered: 1) Given a stencil based computing kernel, what constitutes an optimal memory banking scheme that minimizes the number of memory banks required for conflict-free accesses? 2) Furthermore, if such an optimal memory banking scheme exists, how can an FPGA designer automatically determine it? 3) Finally, does any stencil based kernel have the optimal banking scheme? We believe that all these questions possess especial interests as High Level Synthesis (HLS) gradually gains popularity among FPGA designers.

Furthermore, many scientific codes, unlike stencil-like computing kernel with static memory offsets, exhibit much more general and sophisticated memory access patterns, thus posing much greater challenges to achieving effective memory partitioning and mapping in order to facilitate parallel memory accesses. Intuitively, if the memory accesses in non-stencil kernel computing are completely random, then effectively extracting any kind of parallelism is unlikely. As such, one naturally wonders what happens if we limit our scope to a subset of non-stencils that obey special mathematical properties. Unfortunately, even for the irregular non-stencil kernel with affine memory accesses, the body of work is quite limited because its changing geometry of memory accesses during each kernel iteration makes it complicated to find some sort of pattern to exploit that is independent of the problem size. This problem is further aggravated that, in order to keep calculations simple enough to be implemented efficiently with hardware, most analysis focuses

on linear transformations and polyhedral analysis of the memory access that restricts the number of solutions. In short, to fully exploit memory-level parallelism in non-stencil kernel computing widely found in scientific applications, finding a versatile yet cost-effective method to synthesize application-specific hardware module, which not only is easy to implement but also assures solution optimality, from high level software code is imperative.

Motivation

With all this in mind, this section presents a relevant example where we explain in detail the advantages of optimal memory partition algorithms and how they can help increase data throughput while maintaining resource utilization to a minimum.

Video and image processing as well as feature extraction inside the convolutional layers in a Convolutional Neural Network or CNN involve a series of memory accesses and arithmetic operations such as multiplication and divisions executed in a loop better left for dedicated hardware to handle. This is because unless the data needed to be accessed by the loop can all fit in cache, which is rarely the case given the increasing resolution of the images begin processed, it requires costly accesses to lower levels of memory which have much higher latency and can take up to several orders of magnitude longer in terms of clock cycles to fetch that cache access. On the other hand, processors have very limited quantities of dedicated hardware used to handle more complicated arithmetic operations such as multiplications, divisions and Multiply-ACumulate (MAC) so only a limited amount of concurrent instructions can be scheduled at any given time.

In contrast, a dedicated hardware accelerator offers a custom memory subsystem that can be tailored for a specific kernel such that the number of costly off-chip memory accesses is minimized by taking advance of data reuse at the same time one can have as many processing elements, each

capable of executing all the required arithmetic operations, as needed to achieve certain level of performance.

In the aforementioned applications, one of the most common computational kernels is the Sobel edge detection. This kernel, belonging to the stencil family of memory access patterns, is convolved with the input image or feature map to generate an output image that removes the low-frequency information, leaving only the areas where there was a sharp change in the content of the input, i.e. edges. The traditional Sobel Edge detection kernel is composed by two 3×3 matrices [8] whose coefficients can be seen in equation 1.1.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A, G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A, \tag{1.1}$$

These two matrices G_x and G_y are convolved with the input data to calculate the gradient in the x and y direction respectively. The final output G, that combines the gradient information for the x and y axis, is computed by taking the magnitude of both gradients as per equation 1.2:

$$G = \sqrt{G_x^2 + G_y^2} \tag{1.2}$$

In Figure 1.2 we can see the original data to the left, in this case an image, and the output resulting of applying the Sobel operator to it. In this case the output is an image where the edges of the objects in the input image are highlighted.

Figure 1.2: Example of Edge Detection. Original image (left) and filtered image using a Sobel Edge Detection Kernel (right). By Shaddowffax - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index. php?curid=45561476

The way the convolution operator works on an image is exemplified in figure 1.3. In real applications we operate on each individual pixel of each color channel in the image, which naturally forms a grid. To keep the example simple enough instead we operate on a 4×5 grid superimposed on the grayscale image shown in (a). Operating on this grid now, we see the first iteration of the operation in Figure 1.3 (b) where we need to fetch the data elements in gray in order to calculate the gradient as shown in Equation 1.1 and 1.2 which will be stored in element (B,B) of the output image (right) shaded in dark gray. In the next iteration, shown in (c), the 3×3 window of interest is shifted one column to the right. Similarly, the result of this second computation is stored one element to the right from the previous one in location (B,C). Finally, (d) shows the final result of applying the kernel to the whole image.

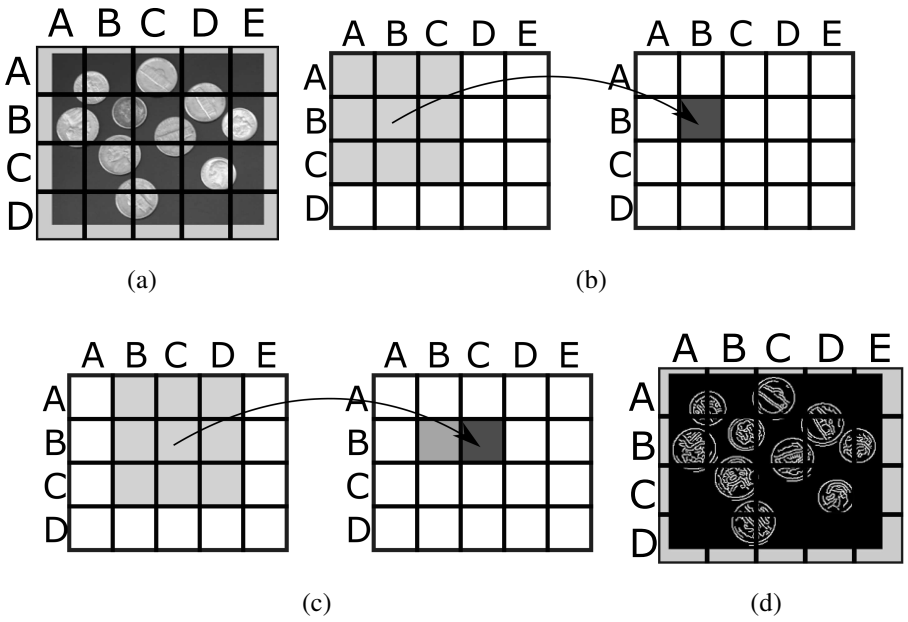


Figure 1.3: (a) Original grayscale image with superimposed 4 $\times$ 5 grid used as input of the Sobel edge detection kernel. (b) First iteration of the kernel. Light gray, data elements accessed from the input image (Left). Element in the output image where we store the gradient computation after applying the kernel (Right). (c) Second iteration of the kernel. Light gray, data elements accessed from the input image (Left). Element in the output image where we store the gradient computation after applying the kernel (Right). (d) Output image after applying the kernel to the whole image

Trying to execute this kind of kernels, where each iteration requires access several memory locations to complete a single operation, in a traditional architecture with a memory subsystem similar to the one seen in Figure 1.4 has the disadvantage that even if we have enough execute units to perform all the arithmetic operations we need in parallel, the monolithic memory architecture implies we need to serialize the accesses even to the on-chip memory to fetch all the data we need. In other words, each of the nine memory locations needed to be accessed for this example would need to be accessed from on-chip memory one by one. If we can access on-chip memory once per

clock cycle, this means it will take 9 clock ticks just to fetch the data we need to start operating. For larger kernels this problem is aggravated even more. Note this also means these kind of kernels are memory-bounded according to the roof-line model since performance is limited mainly by the available bandwidth and have relative low operational intensity.

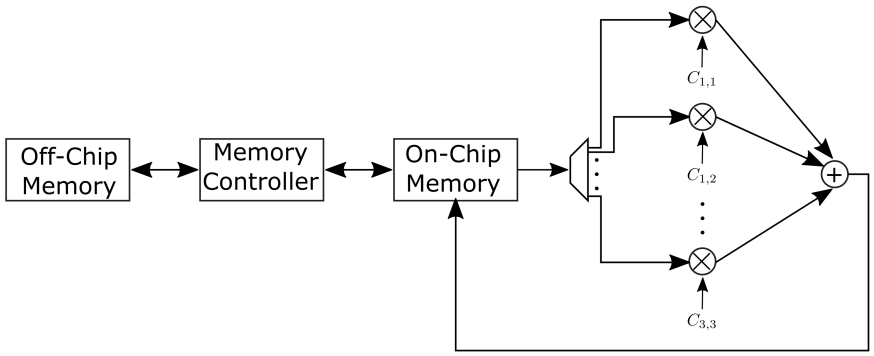
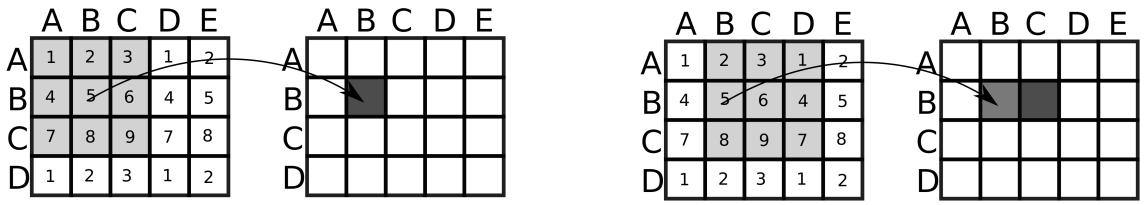


Figure 1.4: Example of a traditional memory subsystem

However, if we can find a way to divide the data into banks, meaning storing the information in independent memories that can all be accessed simultaneously, then we could move away from the monolithic and sequential data access architecture to a more efficient and parallel one. Figure 1.5 shows an example where we have divided the data domain, i.e. the original input data, into 9 different banks denoted by the number inside the grid.



(a) (b)

Figure 1.5: Partitioned data space. Each number corresponds to a bank. Elements with the same number are assigned to the same bank. First (Left) and second (Right) iterations of the Sobel Edge Detection Kernel.

Although the architecture of a dedicated hardware accelerator, and the associated custom memory subsystem and banking scheme, will vary from kernel to kernel, there is a plethora of methods designed to find a general solution to the banking problem. These techniques will be explored in Section 2. Using the algorithm defined and explained in [1] we find a valid and optimal banking scheme for the Sobel Edge Detection kernel that uses as many banks as memory accesses are required to complete one computation which is the definition of optimallity for this problem.

If each bank is independent from each other and can be accessed concurrently, then from Figure 1.5 (a) we observe now all elements inside the gray area on the input data (left) can be accessed simultaneously since they have all been assigned to different banks. If we still maintain the condition that all memory accesses can be performed in one clock cycle then now we only require one clock tick to fetch all the data we need to complete the current computation from the on-chip banks. The result of course is then stored at the same memory location of the output data since the actual execution order of the kernel and the required read/write locations of each iteration have not been modified in any way. A valid memory partitioning scheme is not valid for just one iteration, it is supposed to ensure conflict-free, fully parallel memory access during the entire execution of the

kernel. In Figure 1.5 (b) we see the new memory locations we need to access on the next iteration from the original data domain (left). Note that even though the region of interest has been shifted to the right by one column all the elements in the shaded area are still assigned to different banks which means that once again we can access them all in parallel without them interfering with one another and thus having to serialize the accesses. If we overlap a 3×3 rectangle on any region of the input data domain we will see all the elements will be assigned to different banks, ensuring the desired property of conflict-free parallel memory access.

Due to the nature of the convolution operator, the relative position of each access within the kernel (for example, the top left corner of the 3×3 rectangle in our Sobel example) will access all the banks after the code finishes execution. This means, all possible permutations of the banking will be accessed withing the region of interest. Because of this, extra routing logic is needed to have each execution unit access all banks. This extra routing logic might possibly reduce the clock speed by some amount but the speedup gained by parallelizing the memory access greatly surpass this reduction in execution speed. It is because of this reduction in clock period that the final speedup gained after applying a memory partitioning algorithm is not exactly equal to the number of memory accesses in each kernel. Continuing with our Sobel example, this means that even though we reduced the number of clock cycles it takes to fetch all the data elements from on-chip memory from 9 to just 1, the speedup gained is not $9\times$ but in real applications might be closer to $8\times$.

Figure 1.6 shows a simplified diagram of the memory subsystem used by a hardware accelerator that has been designed to use a banking scheme allow parallel access to the required data.

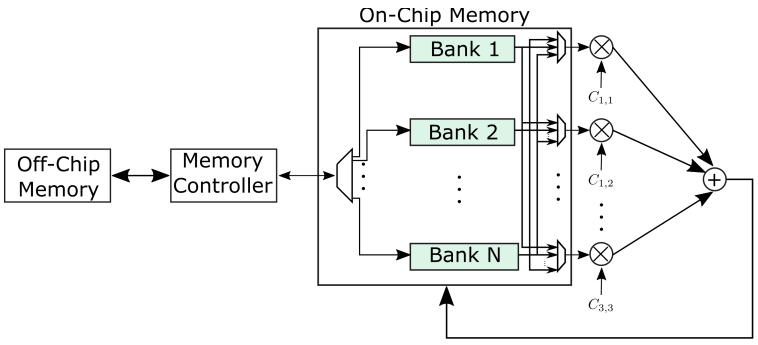


Figure 1.6: Example of a memory subsystem with several banks to allow conflict-free, parallel memory access.

The off-chip memory controller is in charge of loading the data required for the kernel either on its entirety or in sections that fit the on-chip memory and distributes it to the corresponding bank following the pre-computed partitioning scheme. Then, during runtime, a multiplexer routes the data from the correct memory bank to the processing unit that requested a particular memory location to be used on the current iteration.

Note with this architecture the memory controller that handles off-chip communication still has to do the same amount of work, bringing on chip and off loading the same amount of elements but now the on-chip memory access is done in parallel.

Contributions

The main objective of this dissertation is to compile our work in the area of automated memory partitioning and banking algorithms for HLS tools targeting reconfigurable architectures. We

include our novel tessellation-based memory partitioning and mapping, graph-based approach to optimally solve memory banking for any given shape of stencil as well as our graph-based methodology to arbitrarily reduce memory access conflicts on non-stencil code. Finally we present our work where we find an algorithmic solution to obtaining the optimal partition factor for a sub-set of non-stencil computations we define as quasi-stencil by means of applying a non-linear transformation that raises the dimensionality of the problem which in turns converts this Quasi-stencil code into a stencil, opening up a plethora of partitioning algorithms inclusion optimal ones.

In this dissertation we claim the following contributions:

- We expand on the literature review presented in our previous work to include all the most recent research up to the moment of compiling this dissertation. With this, we aim to paint a clearer and more detailed picture of the state-of-the-art techniques with regards to automated memory partitioning algorithms in HLS for both stencil and non-stencil code kernels by comparing them in-depth with our approaches in the appropriate cases. Mentioning the pros and cons of each.

- We compile our previous work in a coherent manner with the goal of presenting a cohesive and unified framework that showcases the evolution of our work and the increasingly better results obtained in terms of partition factor, resource utilization, clock period, among other performance metrics not only for stencil computations but also non-stencil code.

- We refine our testing and result collection methodology to obtain updated results, comparing our performance metrics against those of the applicable state-of-the-art methods in automatic memory banking for HLS targeting reconfigurable architectures.

Dissertation Outline

The rest of this dissertation is divided as follow:

- Review of current state-of-the-art methods and related literature are presented in Chapter 2.
- Chapter 3 contains our approach in tackling the optimal memory partition problem in stencil computations based on a purely geometric approach involving tessellation to find the smallest region of repeated banking. This mapping is then stored and used to do the partitioning. With this, we achieve better results in some kernels than current state-of-the-art methods. Our algorithms are capable of automatically finding the optimal multi-dimensional loop unrolling factor and finding the optimal partition factor when the original stencil does not tessellate the entire data domain. To find a proof on the optimality of our solutions, we start exploring the idea of modeling the problem a a graph in order to use the well established and powerful algorithms in graph theory to solve the partition problem.
- Our graph-based solution for stencil computations is presented in Chapter 4 where we find the smallest induced sub-graph of the entire conflict graph needed to be colored in order to find the optimal partition factor of the kernel. The optimality of this solution is mathematically proven by using an optimal graph coloring algorithm to find the partition. For non-stencil code, given the variance in the geometry we are unable to algorithmically find the equivalent induced sub-graph as in stencil computations.
- Chapter 5 contains our approach of trying to emulate the ability of implementing a banking scheme where we store a region of repeated banking into a small memory, we formulate our graph-folding strategy where the entire conflict graph is mapped to a small memory of a predefined size generating a wighted graph. The aforementioned graph is then colored using an algorithm depending on the end user needs and the solution is stored to do the mapping.

With this approach we can arbitrarily reduce the number of memory access conflicts at the expense of additional resource utilization, maximizing memory access parallelism.

- Our algorithm approach to finding the optimal partition factor for a sub-set of non-stencil code is explored in Chapter 6. Here, we find a sub-set of non-stencil code we have labeled as Quasi-Stencil which under a specific non-linear transformation, namely prime factorization, is mapped to a domain where the memory accesses now behave as stencils, which opens up the possibility of using the vast repertoire of stencil memory partitioning techniques, including optimal ones.

- Finally, Chapter 7 we summarize our findings and give our conclusions regarding our contributions to the stat-of-the-art memory partitioning algorithms as well as mention possible future research directions we desire to pursue including adding data reuse analysis to our methodologies and further refinements to our quasi-stencil methodology in order to handle more general cases.

CHAPTER 2: LITERATURE REVIEW

Study of memory partition schemes dates back to the first days of computer science and memory banking. Attempts to automatically find the optimal banking scheme for simultaneous, conflict-free, accesses in stencil computations can be traced back to early work dealing with memory partition in parallel computers [9]. All these works shared the common objective to create a conflict-free memory access scheme for memory access maps in the form of rows, columns, diagonals or anti-diagonals operating on square matrices of size $N \times N$ with N banks. Work in [9] studied how to partition the memory space in the fewest number of banks for kernels that had stencil memory access in SIMD machines following a technique to generate the bank and address on the fly called skewing schemes that relied on hyperplanes and on the fact that if the stencil tessellates the plane, then there should exist a linear skewing scheme to do the partition. The limitation of this technique and all others using the same partition scheme with just one family of hyperplanes is that it does not ensure the solution will indeed be optimal for all stencils and does not provide an upper bound to the maximum partition factor it might need. Later work by [10] proved a conjecture by [9] that claimed there exist a valid linear skew, if there exists a valid periodic skewing scheme. The advantage of this skewing scheme being periodic is that it allows for an efficient and simple way to locate the data. This work also proved there is a polynomial time algorithm to determine if a stencil tessellates the memory space. In [11], they refined the theory behind the skewing schemes on the basis that previous mathematical definitions where imprecise. The most significant contribution of this paper is the fact that an upper bound to the maximum number of banks needed to generate a linear skewing scheme is stated, being the first prime greater or equal to N for an N-point stencil. Similar results were compiled and polished in [12].

More recently, the authors in [13] translated the idea of hyperplane partitioning to the HLS framework with their AMP method. This method maintained the same single family of hyperplanes for

bank calculation but added a memory padding technique that allowed for a much more simple way to calculate the address of a memory location in a particular bank at cost of some memory overhead. This method still lacks an optimality proof, namely the minimum number of banks and the actual parameters of the partitioning hyperplane are unknown, requiring an heuristic search to find them. Additionally, it does not always give the true optimal partition factor [10]. Further work on the area by [3] introduced the concept of memory address block, where two or more contiguous memory locations on the same row are assigned the same bank, each with a different address inside the bank, which allowed to reduce the number of the partition factor needed for some cases. They also refined the padding technique to reduce memory waste. Some of the limitations is that the blocks only extend in one of the dimensions of the memory space and still there is no formal proof of what is the minimum partition factor needed to do the mapping. In [14], the authors introduce a method that uses the geometric information from the stencil itself and can find a valid solution (partition factor and orientation of the hyperplane family) to do the mapping much faster than previous methods with much less arithmetic operations needed. Again, this solution does not always guarantee the same partition factor, but it ensures all banks are balanced (contain the same number of elements) and reduces memory overhead.

In addition to the hyperplane mapping strategy, there has also been a line of research based in lattices that tries to overcome the limitations of the aforementioned method. In [15] the authors propose the use of lattices to solve the problem of memory reuse. Although they are trying to solve a different problem, they set a solid mathematical background to do the analysis of memory access under the lattice framework. This work, based on the liveliness of a variable, tries to reduce the memory needed for the execution of algorithm. They make the connection between an integer lattice and a modular mapping of the indexes of the array, a mapping strategy of the same dimension as the data space. One of the limitations of the this method is that, since their goal is not to solve the memory partition problem, the partition factor considered optimal is the determinant of the used

lattice, which does not guarantee will be the optimal in terms of the smallest factor. The definition of conflict also differs from the one considered in the traditional partition problem. Instead of it being simultaneous accesses to the same bank in a loop execution, they consider a conflict when there are some indexes considered alive simultaneously under a given schedule. Work in [16] improves and expands on the idea and focuses explicitly in memory partition problem. They use Z-polyhedra model to do the analysis and follow the same concept of integer lattices and a modular mapping of the indexes of the array to do the mapping. The main idea is that they explore all lattices of the same rank as the memory space such that the determinant corresponds to the number of memory banks to use, and the optimization problem is formulated around the minimization of an objective function seeking to reduce the number of conflicts. In this case, we have a solution that is analogous to using as many families of hyperplanes as the dimensionality of the data space, this guarantees the optimal partition factor can indeed be found. Note that this also means that any solution obtained in the hyperplane partitioning method using a single family can also be found using this approach, making it only a subset of the possible solutions. They also tackle the problem of memory minimization, solving a minimization problem that ensures either asymptotically zero or a small, fixed, and arbitrary amount of memory waste. The main limitation of this work is the assumption that the user has prior knowledge of the (at least approximate) number of banks to be used (although an heuristic search to find the best determinant is considered) and not find the smallest factor from the start. Another limitation of this work is that it relies on and the assumes the code is already parallelized and the loops properly rearranged.

Our previous work in [17] reintroduces the concept of tessellation and modifies it specifically to fit the HLS framework targeting FPGAs. In it, we further developed and expanded on the concept of block introduced in [3], relating it with the concept of multi-dimensional loop unrolling, thus making the block a hyper-rectangle of the same dimension as the problem. We also made the connection between the concept of lattice (and the associated families of hyperplanes) and

tessellation, using this as their basis to claim that we can indeed find the optimal partition factor. The main idea of this work is the introduction of the Supertile, the smallest hyper-rectangle such that the mapping contained in it allows for conflict free access and can be used to cover the entire data domain just by repeating it side by side without overlapping. This solution, as well as the intra bank offset calculation methods, offers not only a simplified bank and address calculating logic, eliminating the need for DSP slices entirely, reducing overall resource utilization, and achieving higher clock speeds, but in many cases, a reduction in memory waste in comparison with state-of-the-art methods. The main disadvantage is the fact that still no upper bound to the partition factor is given.

More recently, graph-based approaches have received a lot of interest from the research community. In [18], the authors map addresses from the memory access trace to a small subset of addresses with the goal of finding a solution with N banks using a mask with $log_2(N)$ bits. Each of the masked address is considered a node in a graph and addresses that need to be accessed in the same iteration are joined by an edge. If the maximum clique is larger than N, then N needs to be increased and the whole process repeats until this condition is met. The bank assignment is then done using optimal graph coloring. The solution is then stored in a multi-level lookup table. The main advantage from this method, contrary to [16] and [17], is that it can in theory be used for any kind of kernel, stencil or not, with better results than [3] for non-stencils and no worse for stencils. The disadvantage is the time it takes to test all possible masks to find the best and there is still no guarantees on the upper limit needed to achieve an optimal solution. Although this method is general enough to be used in both stencil and non-stencil kernels, and even those memory access patterns that can not be written as affine accesses, the solution space explored leaves out certain solutions that at a minimum increase of hardware usage, provide a much better conflict rate, increasing throughput. If the memory access are spread all over the memory space, one would require a mask that includes the information for most of the bits, and the resulting graph would

prove increasingly difficult to color given the NP-complete nature of he problem. Another graph based approach is the one presented in our previous work [1] .Unlike previous approaches, in this work we propose a graph-based approach to find the optimal partition factor of stencil computations natively and with proof absed on a rigorous graph theory framework. Here we construct a special data structure called ESG, a special graph that, when colored optimally, gives the optimal partition factor for that given stencil. Coloring this ESG gives the advantage of the result being optimal without having to color the entire memory conflict graph of the problems which given the NP complete nature of the problems is unfeasible even for small problems. The main limitation of this method is, for bigger stencils, performing optimal graph coloring can be computationally expensive. Because to generate the ESG we require to convolve the stencil with itself, the fact of not having an stencil to work with discards this method to be implements for any kind of non-stencil code directly.

Memory partitioning for non-stencil code however presents a very limited body of study, specially for FPGA and the HLS community. This is why we proposed another graph-based framework that tries to tackle this problem in [2]. In this work we propose a best-effort approach that tries to average out the conflict graph to a manageable size by overlapping regions of it and color this graph to obtain a mapping scheme. Although this method is general enough to be used for any kind of kernel code, the obvious limitation of this method is that it does not offer conflict-free memory access for all cases and cannot provide the optimal partition factor for a given non-stencil code. Another limitation is finding the best tile size to do the "folding" is not explored and we provide just a best effort solution based on some pre-defined folding sizes. It is good to note however that when applied to a stencil graph, if the right folding size is chosen, one will find the same solution as the Supertile from [1]. This optimal folding size for stencils is evidently the size of the Supertile (or an integer multiple of it) since by definition the Supertile is the smallest region of repeated coloring that can be used to color the entire data domain while still ensuring a conflict-free, parallel memory

access scheme. In the stencil case, if the Supertile is used as the folding size, we can also ensure the conflicts will be entirely eliminated.

Very recently in [19] the authors propose an edge-centric method to improve the performance of non-stencil kernel execution. Their methodology generates a graph that models memory access precedence and dependency of stencil and on-stencil affine kernels. These graphs are divided into ranks, where the data in each rank dependents on the data in lower ranks to be computer. All data that does not have any dependency are assigned to Rank 0, and that serves a the base of the graph. Edges are then partitioned based on their destination and source rank on an N $\times$N matrix, where N is the number of ranks, and each rank is further partitioned into P banks to allow parallel update of ranks. Each block of edges going from Rank A to Rank B that is then partitioned into a P$\times$P block where each block is an edge going from Bank X of Rank A to Bank Y of Rank B. One of the limitations of this method is that if the number of banks sleeted per rank is not enough, it cannot ensure fully parallel update of the rank. Furthermore, since P banks are needed per Rank, if the graph is complex and with several ranks, the number of total memory banks will be increased, which can hurt performance given the increased complexity of the interconnect. A methodology to select the number of banks P per rank to obtain acceptable results is not provided.

While most of the previous work has mainly focused on ensuring parallel access for stencil kernels such as the Sobel edge detection and Denoise but very little work has been done to ensure parallel memory access in general code kernels. Our work in [2] is one of the few efforts to generalize memory partitioning schemes to on-stencil code. In [20] we propose a methodology to tackle a specific sub-kind of non-stencil computing kernels ensuring an optimal solution which offers truly conflict-free, parallel memory accesses. If the kernel satisfies the requirement of being what we define as a quasi-stencil, definition and requirements which are explained in detail in section 6, our methodology can effectively transform its original memory space into a new one through a nonlinear transformation based on prime factorization, where the original non-stencil kernel with

irregular memory accesses will behave like a conventional stencil with regular memory accesses, therefore allowing for the use of the existing methods of memory partitioning and data reuse. For several real-world non-stencils, our results have shown that all of then only require a small number of memory banks independent of their problem size. This is a significant improvement over state-of-the-art methods for non-stencil kernel computations because the reduction in partition size directly leads to a reduction of the interconnect complexity which in turn leads to better resource utilization, power consumption and increased performance. Our future work will focus on expanding the definition of quasi-stencil code to more real-world applications and exploring other effective nonlinear transformations. Ultimately, we aim at achieving optimal partition factors for an increasingly larger number of non-stencil kernel computations.

While beyond the scope of our work, it is worth mentioning some areas of research also try to maximize the use of off-chip bandwidth by improving the data reuse opportunities. This is, the times a computing kernels can reuse some of the data that has already been brought on-chip to perform a new computation by either changing the execution order ot the data layout to reduce the number of very costly off-chip data transfers.

An approach that tries to take advantage of the expressiveness of the polyhedral framework is explored in [21]. They use a scratchpad memory system and evaluate different arbitrary tile sizes by solving an optimization problem aiming to minimize bandwidth while adhering to a maximum buffer size while exploiting data reuse between iterations when possible. The main limitations of this method are that it relies on costly scratchpad memory systems as on-chip storage that have a high resource utilization due to its complexity. Although the authors claim the number of possible tile sizes to try is tractable, claiming in general it remains under 100, this cannot be ensured for all cases. Their model also suffers from the general limitation that the possibilities are enumerated based on a model which if not defined properly might miss a better solution. They also mention this technique cannot guarantee correctness of behavior for the case of imperfectly nested loops.

Similarly to the previous work, in [22] the authors face the problem of exploring the huge design space that is associated with reducing the amount of off-chip data transfers. This time, instead of intra-tile reuse optimization, the authors also explore a design that maximizes inter-tile reuse, which makes the solution space even larger. They achieve the reduction in communications mainly by exploring iteration reordering. Once a target volume of memory operations are achieved, extra optimizations seek to reduce the size of the buffers by means of additional transformations. The way this is done is by modeling the problem with two independent cost functions, one that tries to minimize inter-tile communication cost and another one that minimizes intra-tile communication costs by using reuse buffers. The massive difference between the bandwidth available for off-chip communication and on-chip resources is noted in the work in [23]. In this work the authors take advantage of he cyclo-static representation that can be used to model certain kind of code, namely perfectly nested, stencil code, to solve the problem of how to automatically determine the tile size and optimal buffer size. Although they achieve significant higher throughput over Symphony-C and Vivado HLS solutions even for a lower clock speed, they rely on data duplication to maintain throughput which is not the most efficient use of on-chip resources which is supported by their much higher resource utilization. In a similar manner, the work in [24] seeks to find optimal tile sizes that partition the data/iteration domain into chunks that can fit into size-constrained faster on-chip memory. While most work trying to solve this problem uses a partial solution enumeration approach, here the authors an analytical approach based in polyhedral parametric model to find the the size for optimized data reuse. Their parametric approach formulates a non-linear optimization problem that takes into account on-chip memory constraints and finds the tile size for minimizing communication overhead. To test the validity of their approach and optimization model, they apply their method to three very popular computing kernels: matrix multiplication, a full search motion estimation, and Convolution Neural Network. They compare their results with results for other methods that use random enumeration of tile sizes, a method where the tile sizes are selected through at random and then performance results are computed. On average, it takes this approach

half the time to find a solution that meets the requirements in comparison to the random approach while also achieving a much better communication cost for a given tile size.

CHAPTER 3: TESSELLATION BASED APPROACH FOR OPTIMAL MEMORY BANKING IN STENCILS

This chapter is based on our previously published work in Juan Escobedo and Mingjie Lin,Tessellation-based multi-block memory mapping scheme for high-level synthesis with FPGA, Proceedings of the 2016 International Conference on Field-Programmable Technology (FPT) ©2011 IEEE [25] and Juan Escobedo and Mingjie Lin, Tessellating memory space for parallel access, Proceedings of the 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC) ©2011 IEEE [17] that encompass our purely geometrical approach to find the optimal partition factor and usable banking scheme for stencil kernels.

We will begin this chapter with a motivational example that will be used throughout this chapter to demonstrate concepts and exemplify the contribution of our geometrical approach for stencil kernels.

Motivating Example

In image processing and computer vision, stencil-based computing kernel frequently occurs. Fig. 3.1 (a) lists a simple example of one such example found in an image denoising task. In virtual memory space, during each iteration, one pixel value will be replaced with a simple average of all its nearest neighboring pixels. With the index i and j taking different integer values within their bounds, five distinct memory locations will be either read or written. Clearly, in order to maximize the computing throughput of this code segment, one wish to access all these five memory locations in parallel for all iterations, therefore achieving an iteration interval of exactly one clock cycle. Intuitively, five separate single-port memory banks should suffice. Unfortunately, as shown in Fig. 3.1(c),

not any memory partitioning and mapping can accomplish conflict-free parallel memory access. Specifically, we have shown three access iterations whose memory locations are enclosed with a blue polygon. However, in each of these iterations, at least two memory location belong to the same memory bank, which causes undesirable memory stalls and reduces its overall computing throughput either by two times or three times.

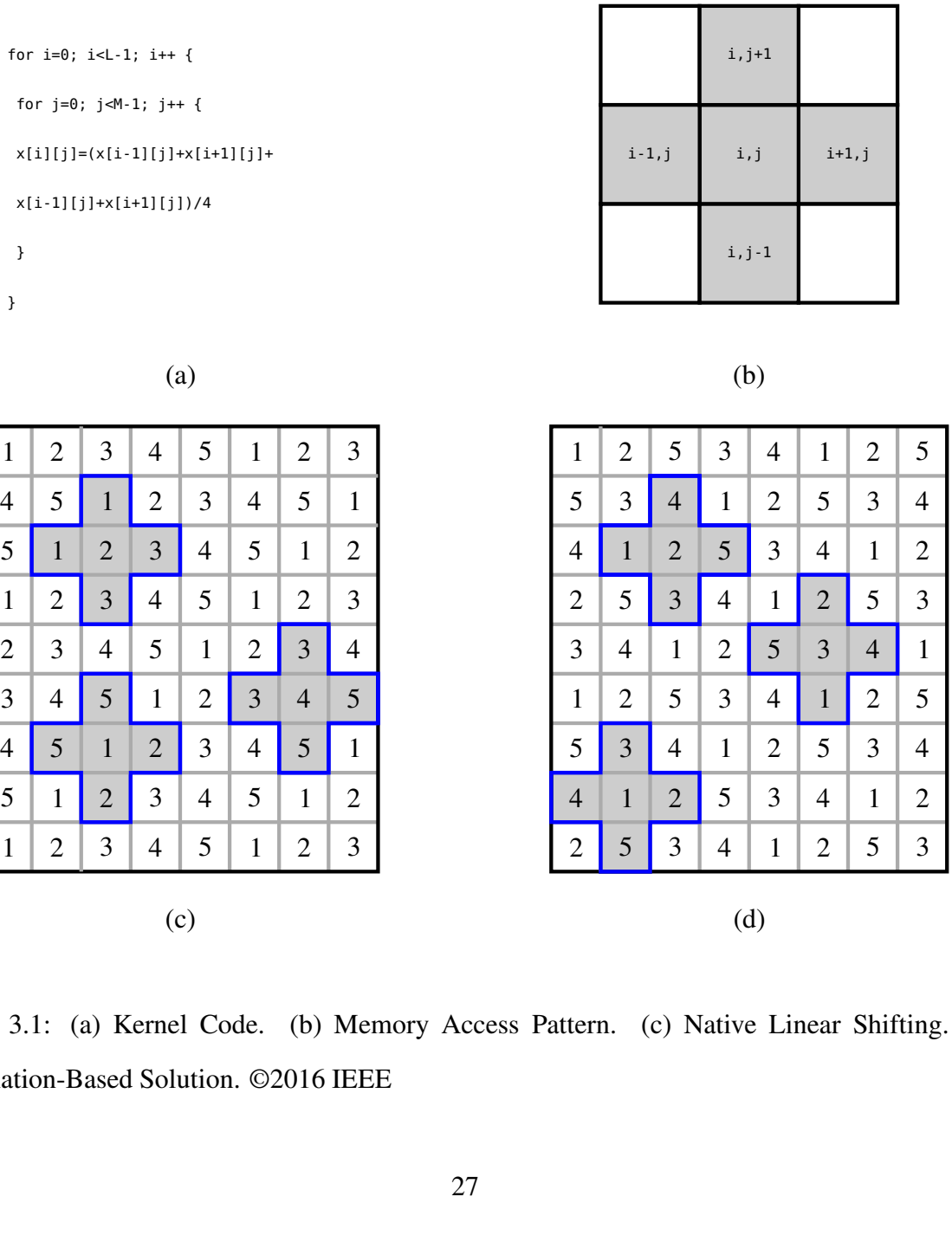


Figure 3.1: (a) Kernel Code. (b) Memory Access Pattern. (c) Native Linear Shifting. (d) Tessellation-Based Solution. ©2016 IEEE

On the other hand, for the same five memory banks, if we just change its memory mapping as shown in Fig. 3.1(d), on can readily verify that, for any iteration, these five memory accesses will only read/write to five distinct memory banks, thus creating 0 memory access conflict. This simple example motivates us to develop an effective and yet efficient methodology to partition and map any standard memory access pattern in order to achieve completely parallel memory access while consuming least possible amount hardware.

Problem formulation

In this section we present the definitions and problems of memory partitioning. Important variables to be used thorough the paper can be seen in 3.1.

Table 3.1: Table of symbols

Variable	Meaning
N	Partition factor
B	Partition block
B_k	Size of partition block in dimension k
m	Number of array references in the inner loop
l	Level of nest loop
$\mathcal{I}$	Iteration domain
$\mathcal{D}$	Data domain
d	Number of dimensions of the array
$\vec{i}$	Iteration vector
w_k	The k-th dimensional size of the memory
$\mathbb{Z}$	Integer set
S	Storage requirement
P	Access pattern (memory access map)
$\vec{A}_i$	i-th element of an access pattern

Definition 1 (Iteration Domain) *Given an l-level loop nest, the iteration domain $\mathcal{I}$ is formed by all the iteration vectors $\vec{i} = (i_0, i_1, \ldots, i_{l-1})^T$ within the loop bounds.*

Definition 2 (Data Domain) *Given an l-level loop nest, the data domain $\mathcal{D}$ is formed by all the vectors $\vec{x} = (x_0, x_1, \ldots, x_d)^T$ within the matrix bounds.*

Definition 3 (Affine Memory Reference) *We say that a memory reference in a loop is affine if $\mathcal{I}$ and $\mathcal{D}$ are affine spaces.*

$\mathcal{I}$ and $\mathcal{D}$ are affine if one can define any d-dimensional affine memory access $\vec{x} = (x_0, x_1, \ldots, x_{d-1})^T$ as a linear transformation of a one and only one l-dimensional iteration vector $\vec{i}$ in the form of:

$$\vec{x} = A_{d\times l} \cdot \vec{i} + \vec{C}$$

$$A_{d\times l} = \begin{bmatrix} a_{0,0} & \ldots & a_{0,l-1} \\ \vdots & \ddots & \vdots \\ a_{d-1,0} & \ldots & a_{d-1,l-1} \end{bmatrix}, \vec{C} = \begin{bmatrix} a_{0,l} \\ \vdots \\ a_{d-1,l} \end{bmatrix}$$

Where $A_{d\times l}$ is a coefficient matrix, $a_{k,j} \in \mathbb{Z}$ is the coefficient of the j-th iteration vector on the k-th dimension, and $\vec{C}$ is a a column vector with constants.

Definition 4 (Memory Access Map) *A pattern consists of m data points or accesses defined as P = $\{\vec{A_0}, \vec{A_1}, \ldots, \vec{A_{m-1}}\}$, where $\vec{A_i} = (A_i^0, A_i^1, \ldots, A_i^{d-1})^T$, $\vec{A_i} \in M$, $\vec{A_i} \in \mathbb{Z}$.*

If the position offset of P from the origin is $\vec{O} = (O_0, O_1, \ldots, O_{d-1})^T$ then the address of each elements of the accesses in P are now $P_{\vec{O}} = \{\vec{O} + \vec{A_0}, \vec{O} + \vec{A_1}, \ldots, \vec{O} + \vec{A_{m-1}}\}$.

Definition 5 (Memory partitioning) *A memory partition of an array can be described as a pair of mapping functions $(f(\vec{x}), g(\vec{x}))$ where $f(\vec{x})$ assigns a bank for the data element and $g(\vec{x})$ generates the corresponding intra-bank offset.*

A bank access conflict between to references $\vec{x}_j$ and $\vec{x}_k (0 \leq j < k < i)$ is represented as $\exists \vec{x} \in \mathcal{D}$ s.t.

$$f(\vec{x}_j) = f(\vec{x}_k)$$

This means the references intend to access the same bank in the same clock cycle. We use Problem 1 to formulate the bank mapping problem (for single-port memories).

Our memory partitioning consists of two mapping problems: bank mapping and intra-bank offset mapping.

Problem 1 (Bank minimization) *Given an l-level loop on the iteration domain $\mathcal{D}$ with m affine memory references $\vec{x_0}, \vec{x_1}, \ldots, \vec{x_{m-1}}$ on the same array, find a partition factor N such that:*

$$Minimize : N = max_{0 \leq n < m}\{f(\vec{x}_i)\} \tag{3.1}$$

$$s.t.\ \forall \vec{x}_j, \vec{x}_k \in \mathcal{D}, f(\vec{x}_j) \neq f(\vec{x}_k), 0 \leq j < k < m$$

Eqn. 3.1 defines the objective function of memory partitioning, ensuring no access conflict between any two references. After bank mapping, a data element in the original array should be allocated a new intra-bank location. For correctness, two different array elements will be either mapped onto different banks or the same bank with different intra-bank offsets. An intra-bank offset function is valid if and only if:

$$\forall \vec{x}_j, \vec{x}_k \in \mathcal{D}, \vec{x}_j \neq \vec{x}_k \rightarrow (f(\vec{x}_j), g(\vec{x}_j)) \neq (f(\vec{x}_k), g(\vec{x}_k))$$

Which means either

$$f(\vec{x}_j) \neq f(\vec{x}_k)\ or\ f(\vec{x}_j) = f(\vec{x}_k), g(\vec{x}_j) \neq g(\vec{x}_k)$$

Problem 2 (Storage minimization) *Given an l-level loop on the iteration domain $\mathcal{D}$ with m affine memory references $\vec{x_0}, \vec{x_1}, \ldots, \vec{x_{m-1}}$ on the same array, find a partition factor N, find an intra-bank offset mapping function g with minimum storage requirement S such that:*

$$\textit{Minimize}: \sum_{j=0}^{N-1} max_{\forall i s.t. f(\vec{x}_i)=j}(g(\vec{x}_i)) \tag{3.2}$$

$$\textit{s.t.}\ \forall \vec{x}_j, \vec{x}_k \in \mathcal{D}, \vec{x}_j \neq \vec{x}_k \rightarrow (f(\vec{x}_j), g(\vec{x}_j)) \neq (f(\vec{x}_k), g(\vec{x}_k))$$

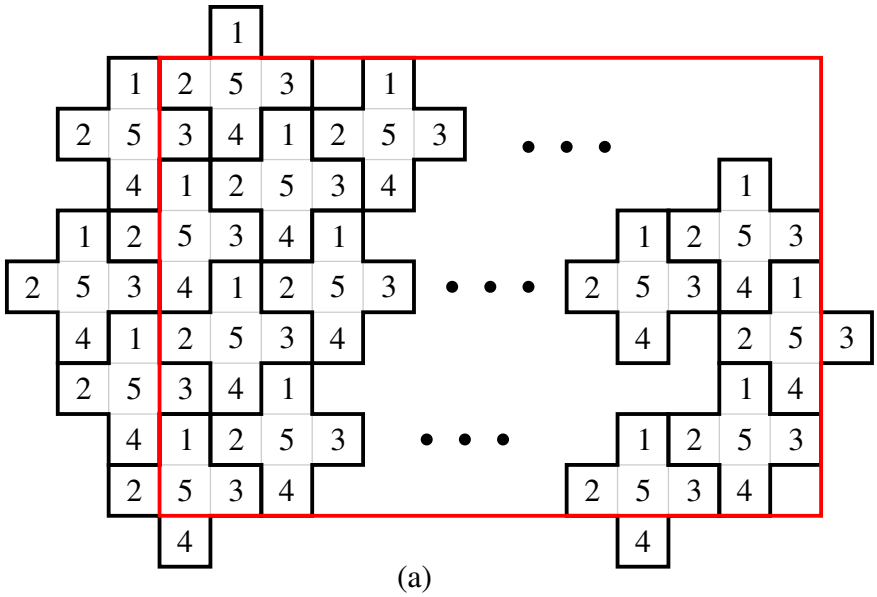
Eqn. 3.2 defines the objective function of partitioning with minimum storage overhead, ensuring a valid partition.

Motivational observation

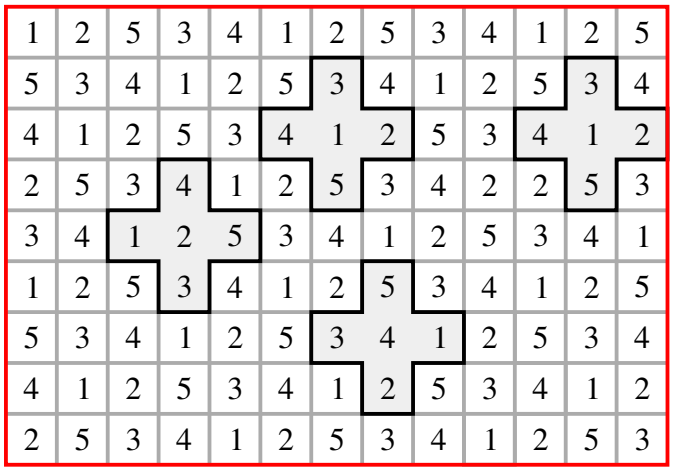
One central problem of maximizing memory access performance is how to partition and map all multidimensional memory references in a multidimensional loop nest to separate memory banks in order to enable loop pipelining with simultaneous memory accesses. Without the loss of generality, in this study, we assume both the loop initiation interval (II) and the physical memory port number of each memory block to be 1.

The memory partitioning scheme in this paper consists of two problems: how to uniquely map each individual memory access to a specific memory bank and how to determine the intra-bank offset for each of such memory accesses.

The basic idea of the proposed methodology is taken from [25], which is to exploit memory space tessellation. Mathematically, *tessellation*, or regular divisions of the plane, are arrangements of stencil shapes that completely cover the plane without overlapping and without leaving gaps. Typically, the shapes making up a tessellation are polygons or similar regular shapes, such as the square tiles often used on floors. Tessellations can be generalized to higher dimensions and a variety of geometries.



(a)

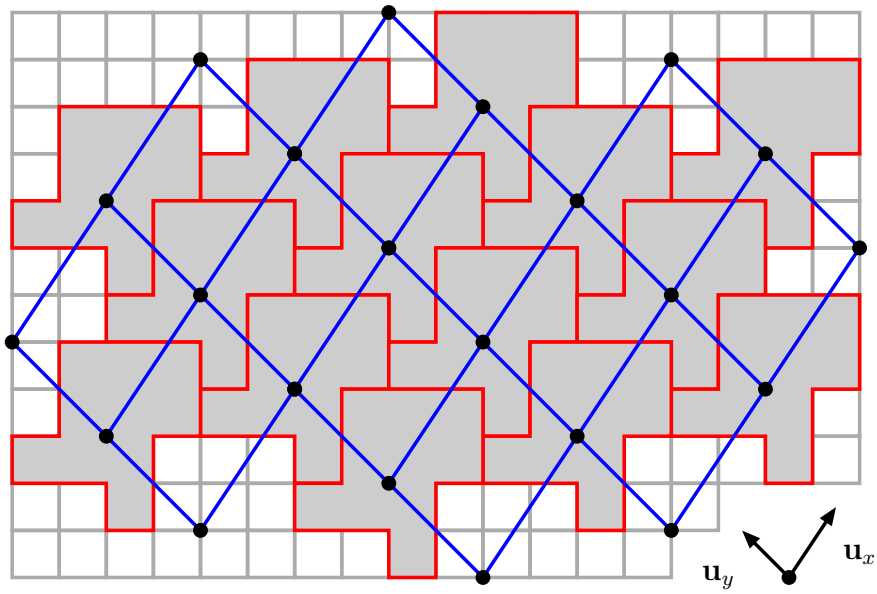


(b)

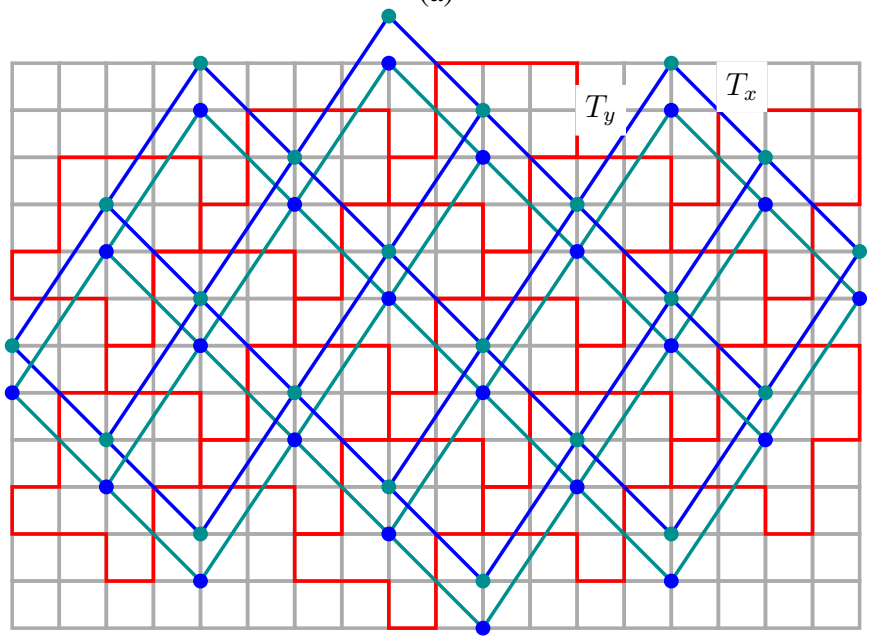
Figure 3.2: (a) Tessellation Example. (b) Resulting Memory Mapping. ©2016 IEEE

As depicted in Section. 3, for the given memory access pattern depicted in Fig. 3.1(a), our tessellation-based memory partitioning and mapping scheme can produce a conflict-free memory mapping shown in Fig. 3.1(d). Fig. 3.2(a) depicts the basic idea of tessellating a given memory access

space, which is bounded with a red rectangle. Each cross-like shape represents a tile. Note that, in each of these tiles, all memory locations are indexed with a unique integer, which denotes a different memory block. With this tessellation and indexing scheme, each memory location is mapped to a unique memory bank. To validate this resulted memory mapping, in Fig. 3.2(b), four different memory access stencils are shown, each of them accomplishes completely conflict-free memory accesses. In fact, moving this memory access stencil to any location within this memory space, no memory access conflict can be found.



(a)



(b)

Figure 3.3: (a) Tessellation Example. (b) Embedded Mesh within a Tessellation. ©2016 IEEE

The fact that a rotationless tessellation can produce a completely conflict-free memory mapping

can be rigorously proven [9], [10]. We now briefly outline this proof. For a given tessellation depicted in Fig. 3.3(a), connecting any corresponding memory location will result a mesh or lattice space shown in in Fig. 3.3(b). Now, imagine translating a sliding window with the same shape of the tessellating tile, it should be clear that no two memory locations will have the same memory block index, hence achieving conflict-free memory access.

Overall Methodology

In this section we describe how we automatically map an array in a loop nest into separate memory banks to enable parallelized memory access. We use a loop initiation interval (II) of loop pipelining to measure the throughput. It represents the clock cycles between the successive iterations and reflects the parallelism of on-chip memory access. For a fully pipelined loop with all accesses parallelized, II is 1. This is the performance target in this paper.

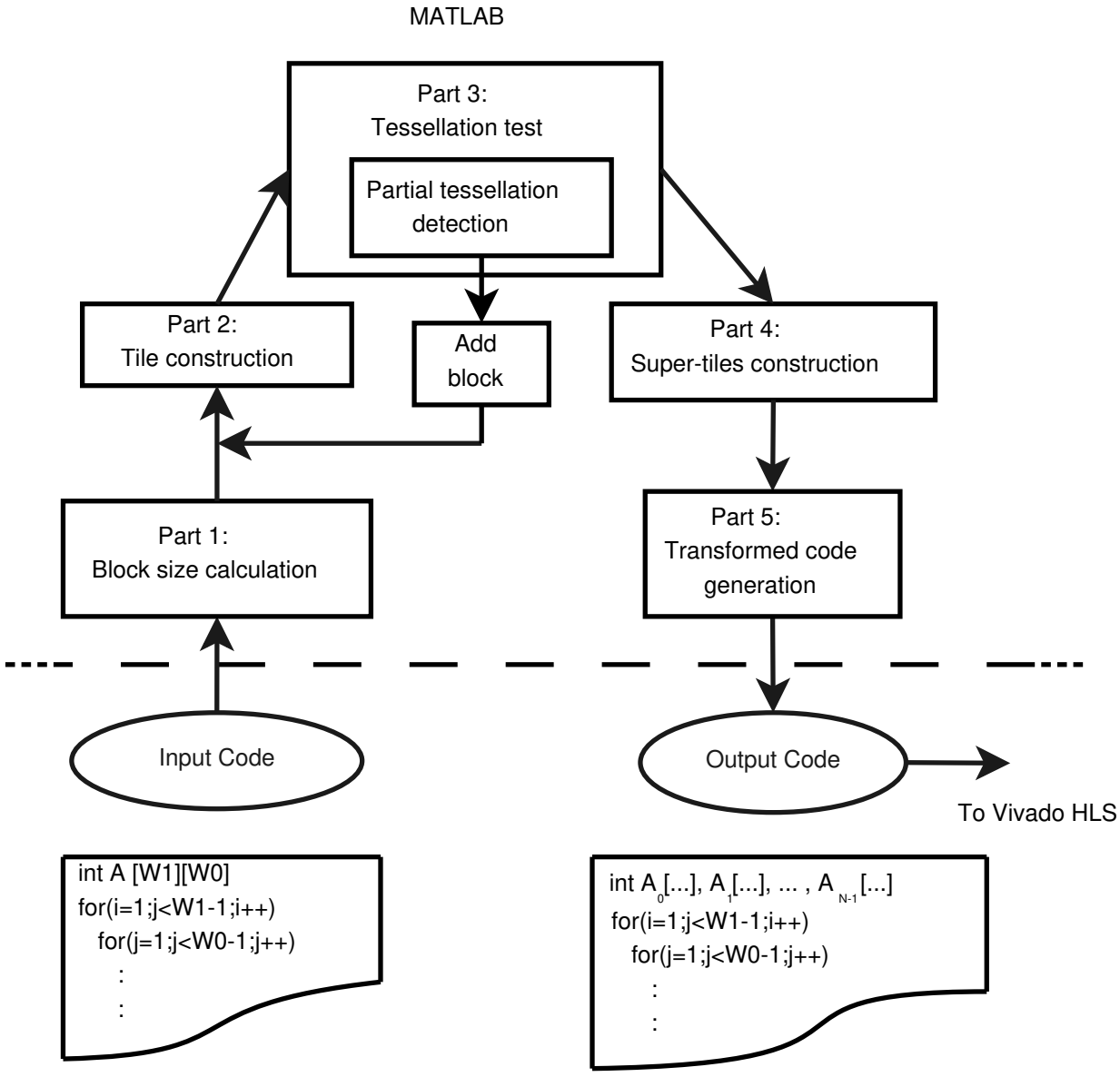


Figure 3.4: Design Flow

Fig. 3.4 depicts all of main steps of our tessellation-based memory partitioning and mapping

scheme. Specifically, given a piece of kernel code, we first extract its memory access pattern in a straight-forward way by treating them as affine references and obtaining the index and offset matrices. We then determine the block size and shape. With this, we create the first tile shape in the form of a horizontally contiguous polyomino. Subsequently, we proceed with a heuristic search, from a small pool of potential candidates, to find the vertices of the smallest parallelogram generated by connecting the same access points of 4 instances of the original polyomino that has no holes. If one is not found, the tile is modified by adding an additional "block" and the process is repeated until no holes are left inside the parallelepiped. This will generate a valid tessellation of the memory access space under consideration. Once we have a valid tessellation, we take advantage of the periodicity of the tessellable pattern to find an equivalent tessellating pattern in the form of a rectangle. This rectangle, so-called super-tile, will be used as a table to determine the bank assigned to a particular reference. In the following two subsections, we describe in more details all key steps necessary for our tessellation-based memory partitioning and mapping scheme.

Block size calculation

The first step in our method is to determine the block size and shape. This is vital to finding the tessellation that offers the solution with the least amount of banks.

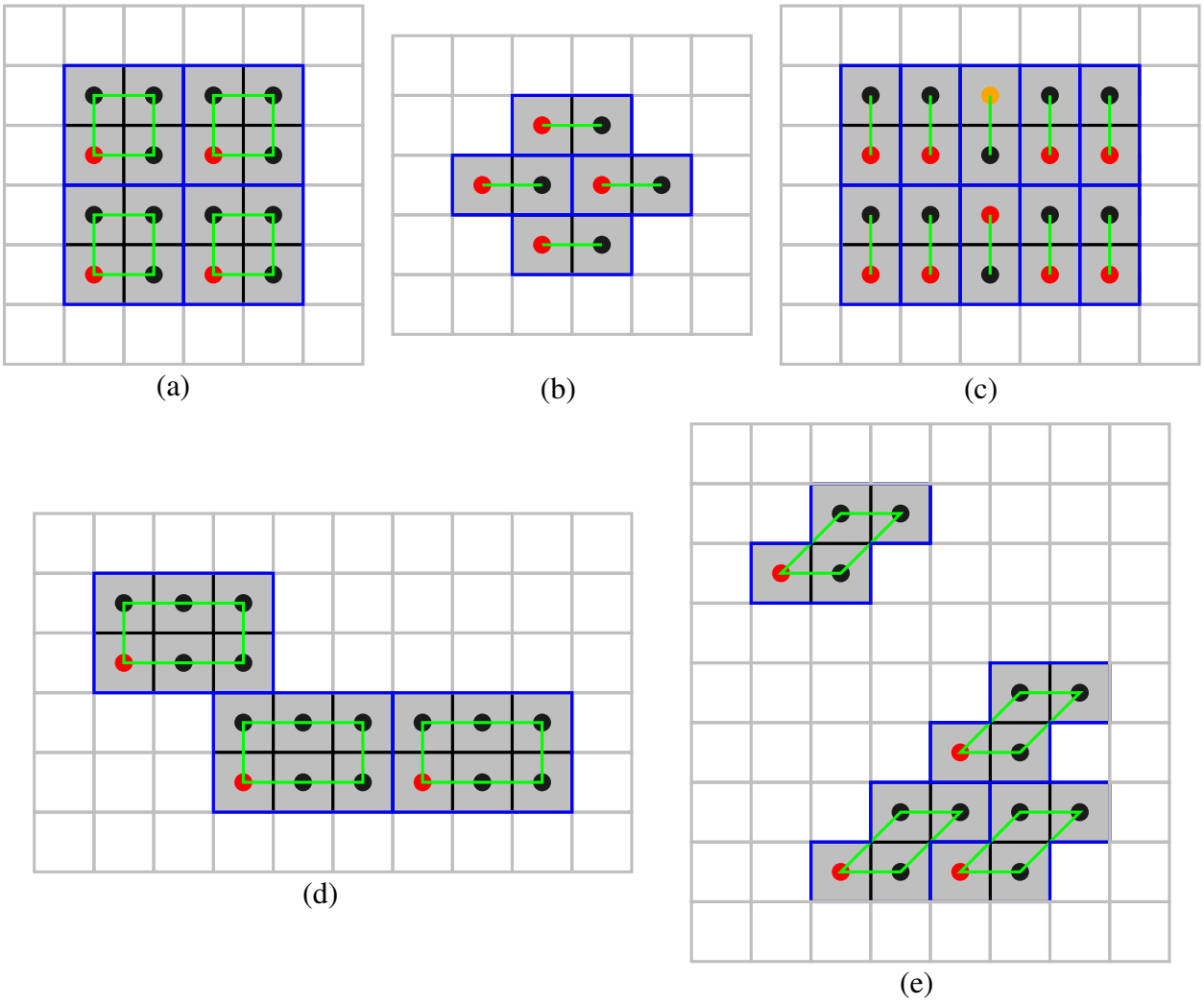


Figure 3.5: Different access patterns and their final tile shape. The blue outline represents the block. Dots are the center of a cell: original memory access are red, extra bank in orange. Green parallelepipeds represent loop unrolling/tilting.

In figure 3.5 (a) and (b) we can see the Bicubic and Denoise access patterns respectively. Both of this patterns would require 5 memory banks [26] without the use of blocks. The number of required banks is reduced to 4 [25] when the block is introduced. The rest are arbitrary patterns

which again, would require a greater number of memory banks if the original pattern were used directly.

Although previous work by [3] also makes use of this parameter, their analysts is restricted to 1-D blocks on d_0, and only sizes that are a power of 2. This because the addition of an extra division operation would hinder performance and keeping it as a power of 2, would result just in a shifting operation, which is free in terms of hardware resources in FPGAs.

The tessellation method takes away this limitation with its much simpler bank and offset calculations, freeing the block to be any size and shape necessary to achieve the optimal bank number.

The block size is intrinsically related to the concepts of loop unrolling and loop tilting, which are widely used in the compiler community to improve performance, particularly spatial and temporal locality. Loop unrolling is a technique commonly used to improve performance when hardware resources are available, allowing the parallel execution of instructions with no inter-dependence. While loop tilting or loop skewing is a technique used to keep dependences between iterations of the outer loop. This is done by changing the bounds of one, or more, of the loops to be an affine function of the others, this changes the areas of the data domain accessed by iterations of the innermost loop, thus removing any dependencies.

If one were to access the data domain using the original access pattern following the optimal tilting, and maximum unroll factor for all dimensions, the elements of the data domain accessed by any particular element would make the block which size and shape would give the best tessellation results.

Both techniques are needed to obtain the optimal results. Applying only loop unrolling to the arbitrary patter 3.5 (e) yields a suboptimal result. Needing to add many more blocks to the tile in order to perform a total tessellation of the data domain. While introducing a tilt, the tile meets all

the requirements for tessellation.

Figure 3.6 shows a simple method to extract the block size, using loop unrolling only. This is the algorithm used in section 3 for all the benchmarks. No loop tilting was implemented because for most stencil applications commonly used, there is no tilt needed in order to achieve the optimal solution.

Data: $P = \{A_0, A_1, \ldots, A_{m-1}\}, w = [w_0, w_1, \ldots, w_{k-1}]$
Result: Block size (B_{size}, B)
$B = [0, 0, \ldots, 0]_{1\times k}$;
$B_c = [0, 0, \ldots, 0]_{1\times k}$;
$B_{nc} = [0, 0, \ldots, 0]_{1\times k}$;
forall $A_j, A_m, m \neq j$ **do**
    for $0 \leq i < k$ **do**
        if $A_m - \{A_m^i\} == A_j - \{A_j^i\}$ **then**
            $\Delta_c \leftarrow |A_j - A_m|$;
        else
            $\Delta_{nc} \leftarrow |A_j - A_m|$;
        end
    end
end
for $0 \leq i < k$ **do**
    if *min* $(\Delta_{c_{i,:}} > 0) \neq \emptyset$ **then**
        $B_{c_{1,i}} \leftarrow \min(\Delta_{c_{i,:}} > 0)$;
    else
        $B_{c_{1,i}} \leftarrow w_i$
    end
    $Bnc_i \leftarrow \min(\Delta_{nc_{i,:}} > 0)$;
end
$B_k = \max(B_c)$ s.t. $w - Bc > 0$;
$B_n = Bnc_n, n \neq k$;
forall B *s.t.* $B_n == w_n$ **do**
    $B_n \leftarrow \max(A_j^n) - \min(A_j^n) \forall j$;
end

Figure 3.6: Block size calculation

Tile construction

Once the block size is obtained, the next step is to generate the first tile to test. Note that the first tile is not always just the overlap of a block on the original accesses.

Data: P, B

Result: Tile (T)

forall $A_i \in P$ **do**

  forall $B_i \in B$ **do**

    T $\leftarrow A_i + B_i$;

  end

end

for *min*(T_j^1) $\leq i \leq$ *max*(T_j^1) $\forall j$ **do**

  Range=max(T_j^0) $-$ min(T_j^0) s.t. $T_j^1 == i$ $\forall j$;

  for *min*(T_j^0) $\leq j \leq$ *min*(T_j^0) $+ \lceil Range/B^0 \rceil$ **do**

    Tile $\leftarrow [j\ i]$;

  end

end

Figure 3.7: Algorithm for Tile Formation.

What algorithm 3.7 is doing, for 2D case, is for every distinct coordinate value in d_1 of the access pattern, calculate the range where there are elements in d_0, and fit consecutive blocks until the interval is covered. This can be extended to N dimensions.

Tessellation

As stated above, the core idea behind our methodology is the concept of geometric tessellation, which mathematically covers a d-dimensional space with a fixed pattern or tile using only translated, non-overlapping copies of the same tile. To achieve a gap-less tessellation, finding a suitable and optimal tessellating regular shape, tile, presents the first challenge. In this study, we need to develop a general methodology to compute a tessellating tile shape for any memory access pattern. Before presenting our algorithms in detail, we state the following theorem:

Theorem 1 *A tile $T = \{T_0, T_1, \ldots, T_N\}$ comprised of N units blocks can tessellate a given plane completely if the corresponding unit blocks of 4 non-overlapping instances of a given tile can form the vertices of a parallelepiped with no holes inside.*

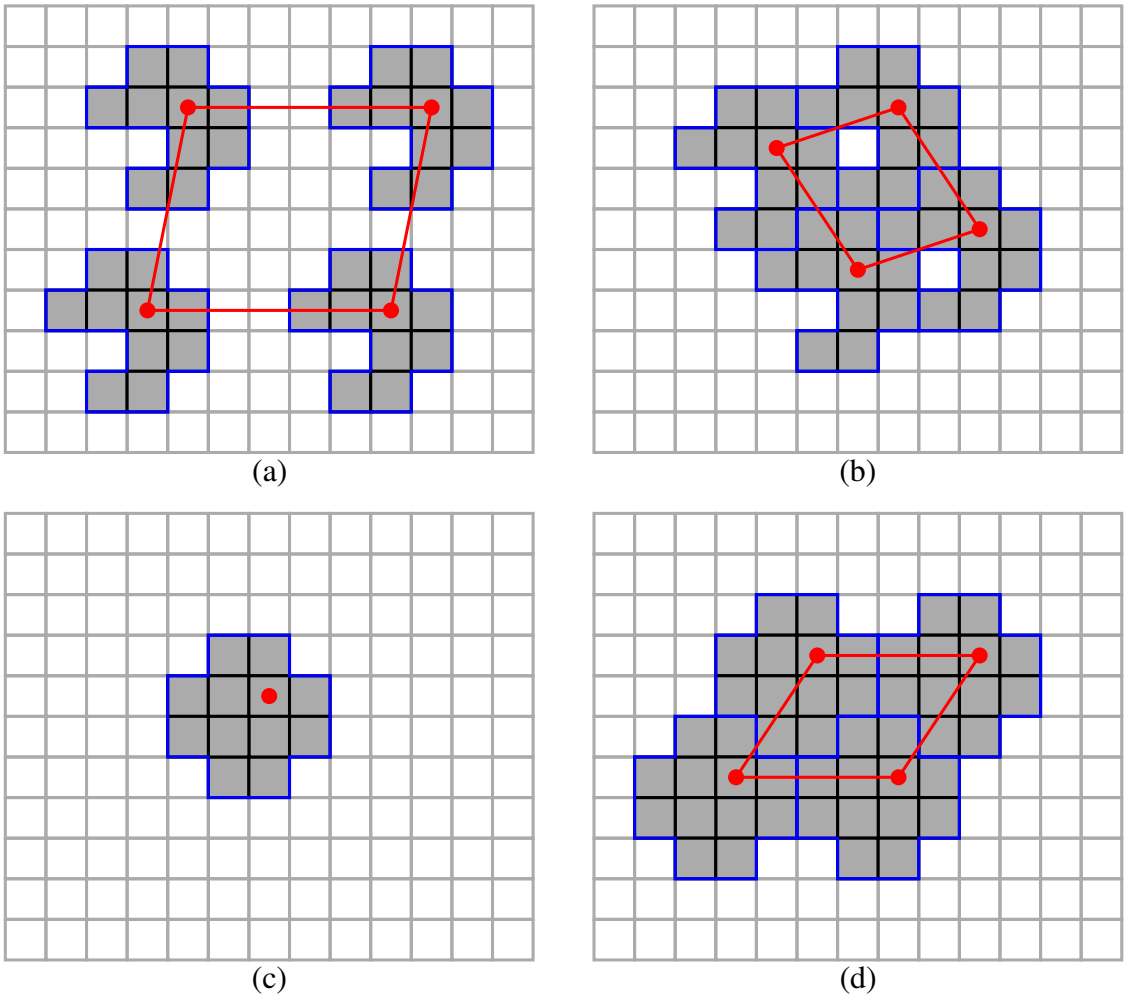


Figure 3.8: (a) 4 non-overlapping instances of the memory access map and embedded parallelogram. (b) 4 non-overlapping instances of the memory access map forming the smallest parallelogram. (c) Updated tile with extra block. (d) Final tessellation with new tile. ©2016 IEEE

For this to be true, the resulting hyper-parallelepiped must be the one enclosing the smallest area (for 2D), smallest volume (for 3-D), etc. If there are no "holes" or unpartitioned elements inside the hyper-parallelepiped, the partition problem is solved. If there are elements that don't belong to any block, then the "hole" is covered by an additional block. This block is attached to a particular

instance and added to all the other instances in the same location and the process is repeated with this new tile until the hyper-parallelepiped has no holes inside. It has been proven that a tessellation is partial if any two or more adjacent, non-overlapping instances of the tile form a hole, thus the need to fill this hole with an extra block to ensure total tessellation of the matrix. This process can be seen in Fig. 3.8, where we have a pattern that has a hole in the smallest hyper-parallelepiped (b), so we add an extra block, obtaining the pattern from figure (c). The partition factor N is then the number of blocks in a particular instance. as seen in Fig. 3.8.

We now present two additional theorems essential to our tessellation-based scheme. Proofs of this theorems can be found in [9].

Theorem 2 *If a tile T=*$\{T_0, T_1, \ldots, T_N\}$ *comprised of N blocks of size B tessellates the plane, each block comprising the tile can be assigned an arbitrary distinct identifier and it will be a valid, and the resulting tessellation is conflict-free partition for the entire plane.*

Theorem 3 *Given a tile T=*$\{T_0, T_1, \ldots, T_n\}$ *comprised of N blocks of size B, each block assigned an arbitrary distinct identifier, that tessellates a d-dimensional space, there exist a hyper-rectangle smaller than* $N * B_0 \times N * B_1 \times \cdots \times N * B_{d-1}$ *that is also a valid solution for the tessellation problem and maintains the same assignments of identifiers to each element of the space when it is used to tessellate it instead of the original tile.*

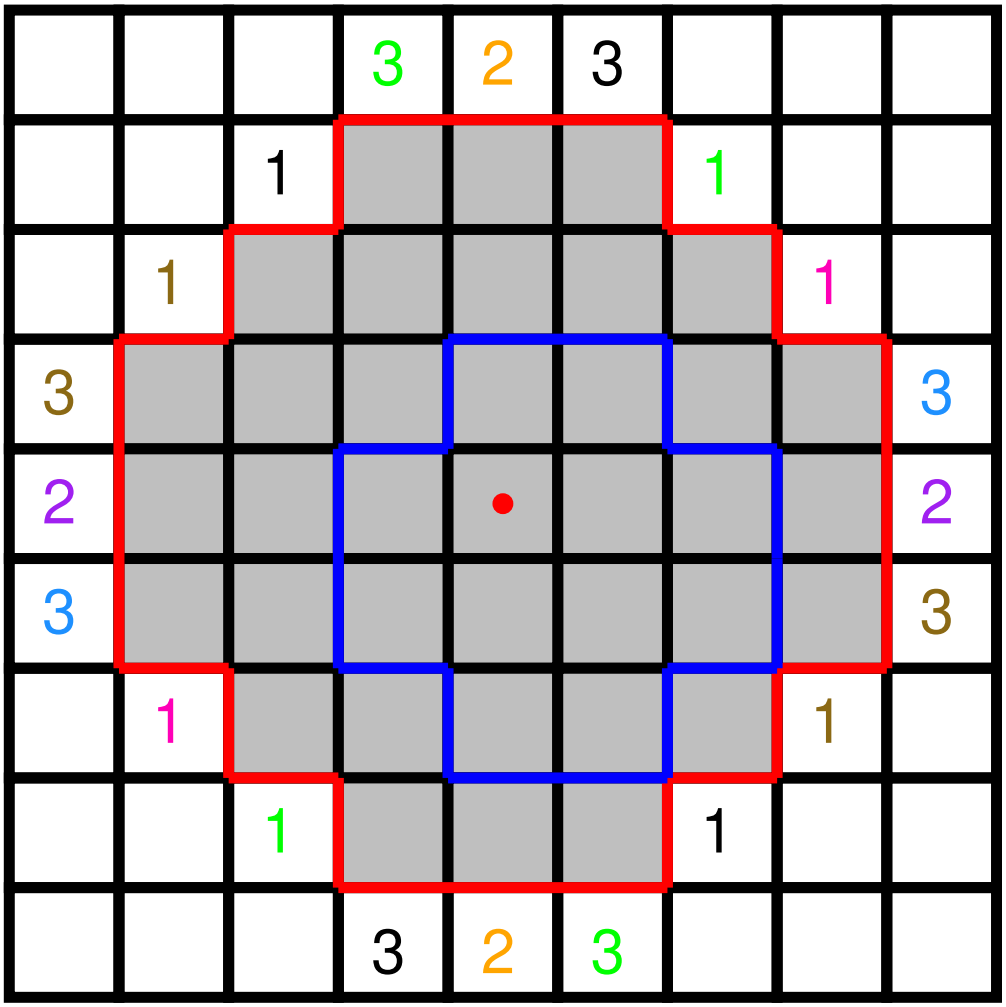


Figure 3.9: In blue, access outline. In red, exclusion zone outline. Matching numbers of same color are symmetrically opposite from center

A hyper-parallelepiped in a d-dimensional space can be defined by a set of d vectors, V_m. Once a solution for the tessellation using a tile is reached, we take note of the base vectors of the hyper-parallelepiped formed by the instances of the tiles. We use this vectors to find the smallest distance in each canonical dimensions where the problem repeats itself. We will use this hyper-rectangle to solve of bank mapping problem. This is what we call a super tile.

To find a set of valid base vectors, we first pick any point in the Data Domain and find the exclusion zone as shown in figure 3.9. The exclusion zone is created by overlapping the all possible tiles a particular point can belong to. This zone is the set of points that cannot have the same bank number (ignoring block size) to achieve conflict free parallel access. The original cell is then taken as the origin and cells on the periphery of the exclusion zone are then considered as possible ends for the base vectors, starting from the new origin. Candidates are explored based on the norm of the vector they would make. Smaller norm means the vector is considered first (1's in the figure) since it would make the smallest parallelepiped.

Once two vectors are found such that they make 4 non-overlapping instances of the tile, we then proceed to check if there are any holes inside the formed parallelepiped and calculate the area. We then select the best candidate based on which has the fewer number of holes and the smallest area. If the best candidate has a hole, an extra block is introduced, increasing the total number of banks by 1, and the process is repeated.

Super-tile construction

Once a solution without holes is found, we use the property of the tessellation that there exist a spatial period in which the problem repeats itself not only on the direction of the base vectors, bu also on the canonical dimensions.

To find said distance we calculate the LCM of all the vectors on each dimension. For the 2D case, since we only have 2 vectors, if we divide the LCM for a certain dimension by the coordinate in that dimension of a vector, we will get the smallest number of times we have to add itself to a certain multiple of the other base vector to cancel the complementary component. This multiple is also the value of the LCM divided by the corresponding component of the other vector.. Then it is a simple matter of knowing whether to add the base vector or its complement based on whether the

coordinates have the same sign or different. Note that since we are working on an integer lattice, all vectors have integer LCM and any linear combination will also be an integer vector.

This is better exemplified with an example. Refer to figure 3.3 where we have base vectors $\mathbf{u}_y = (-2, 2)$ and $\mathbf{u}_x = (2, 3)$. The LCM for dimension d_0 and d_1 are 2 and 6 respectively. To obtain the smallest vector in d_0 we can make from any linear combination of the base vectors, figure 3.10 (b), we divide the LCM in d_1, this is 6, by $\mathbf{u}_y(1) = 2$ and $\mathbf{u}_x(1) = 3$ obtaining 3 and 2 respectively. This means we need 3 $\mathbf{u}_y$ and 2 $\mathbf{u}_x$ to obtain our desired distance. Now, since $\mathbf{u}_y(1) = 2$ and $\mathbf{u}_x(1) = 3$ have the same sign, we just need to subtract 3 $\mathbf{u}_y$ from 2 $\mathbf{u}_x$ or vice versa, instead of adding them to obtain the result. The modulus of the resulting vector will be the size of the super-tile in d_0. Note that doing 3 $\mathbf{u}_y$ - 2 $\mathbf{u}_x$ vector with either positive or negative coordinates while -3 $\mathbf{u}_y$ + 2 $\mathbf{u}_x$ will yield the opposite sign, but since we care is about the modulus of said vector it does not affect the result.

In a similar manner, to obtain the smallest vector in d_1 we can make from any linear combination of the base vectors, figure 3.10 (c), we divide the LCM in d_0, this is 2, by $\mathbf{u}_y(0) = -2$ and $\mathbf{u}_x(0) = 2$ obtaining 1 both times. This means we need 1 $\mathbf{u}_y$ and 1 $\mathbf{u}_x$ to obtain our desired distance. Now, since $\mathbf{u}_y(0) = -2$ and $\mathbf{u}_x(0) = 2$ have the different sign, we can add both directly to obtain our result. Taking the modulus of the resulting vector as the size of the super-tile in d_1.

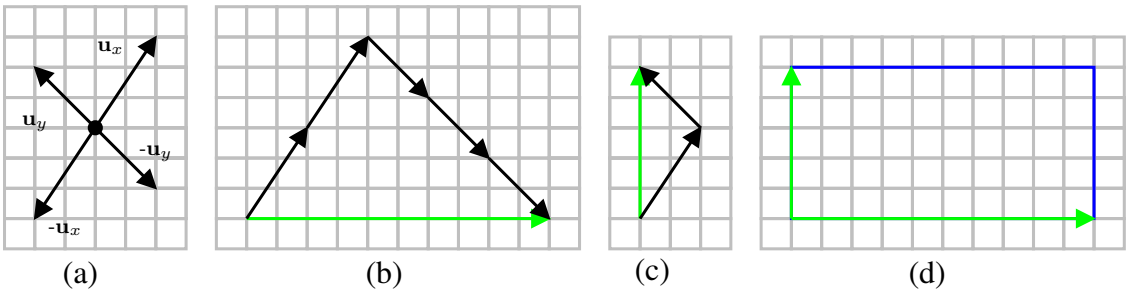


Figure 3.10: (a) base vectors and their complement. (b) Smallest horizontal vector made from a linear combination of base vectors. (c) Smallest vertical vector made from a linear combination of base vectors. (d) Super-tile

The previous can be summarized in the following equation:

$$\begin{aligned} ST = [(LCM^2) * |\frac{V_1^1}{V_1^2}| - sign(V_1^2 * V_2^2) * |\frac{V_2^1}{V_2^2}| \\ (LCM^1) * |\frac{V_1^2}{V_1^1}| - sign(V_1^1 * V_2^1) * |\frac{V_2^2}{V_2^1}|] \end{aligned} \tag{3.3}$$

Where the subscript represents the vector, and the superscript the dimension.

If any of the components is zero, the size of the super-tile of the corresponding dimension is just the non-zero component.

In theory, once the size of the super-tile is known, we can use the vectors of the parallelepiped as a guide, and add non-overlapping instances of the final tile in each direction until a volume equivalent to the hyper-rectangle calculated previously is filled. Assigning each corresponding block of all the instances the same bank number, we store the sequence of bank assignments for all

elements of the hyper-cube in memory.

In practice, starting from an arbitrary cell in the super-tile, we can overlap a block and from each of those cells, move following both base vectors and their conjugate allowing to wrap around, until we have reached all possible points. This set of cells are assigned to one bank. The process is repeated until all points in the super-tile are assigned to a bank.

The final tessellation result for the Memory Access Map in question can be seen in Fig. 3.11(a), which we name it as a super-tile.

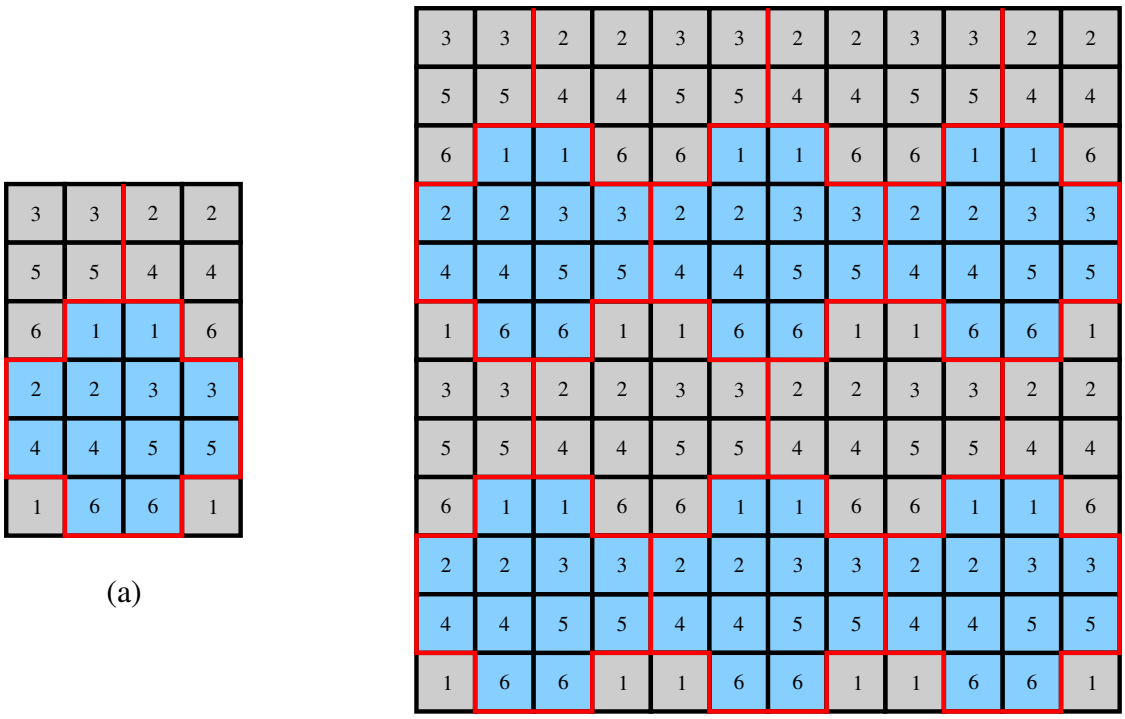


Figure 3.11: (a) Bank mapping memory Mem_B. (b) Tessellation for an $m \times n$ matrix using a 4×6 super-tile. ©2016 IEEE

Super-tiles depicted in Fig. 3.11(a) will be used to tessellate the matrix as seen in Fig. 3.11(b). It has the characteristic that is regular in all canonical dimensions. As such, to obtain memory bank mapping, all one has to do is to calculate the relative position of a desired data element in the matrix to its relative position inside the super-tile. This can be done by modulating each coordinate with the length of the cube in the corresponding dimension. The bank mapping for memory location X becomes an access to an d-dimensional memory in the form of:

$$\text{Bank} = \text{Mem}_B[X_0 \mod a_0, \ldots, X_{d-1} \mod a_{d-1}] \tag{3.4}$$

Given the regularity of the access of perfectly nested loops in stencil applications, the math can be simplified even further by using accumulators and counters instead of costly modulo operation to access all the dimensions of the super-tile every iteration.

Intra-Bank Offset

Due to the regularity of memory space tessellation with super-tiles, the problem of computing intra bank offsets becomes an extension of the above memory bank mapping problem. First, for the upper d-1 dimensions, we want to calculate number of elements belonging to a particular bank that are in a d-2 dimensional space. This is, for 3-D matrix, we want to calculate how many elements are of each bank first in a cube with base vectors $[a_2, 0, 0], [0, w_1, 0], [0, 0, w_0]$, then in a rectangle with base vectors $[0, 0, 0], [0, a_1, 0], [0, 0, w_0]$, and finally in a tessellating super-tile. The maximum of these values for each dimension are then stored in memory. Once this is done, the intra bank offset becomes a function of the intra super-tile offset and, 2 accumulators per access for the 2D case. One to store the number of elements before it in the same row of super-tiles, and

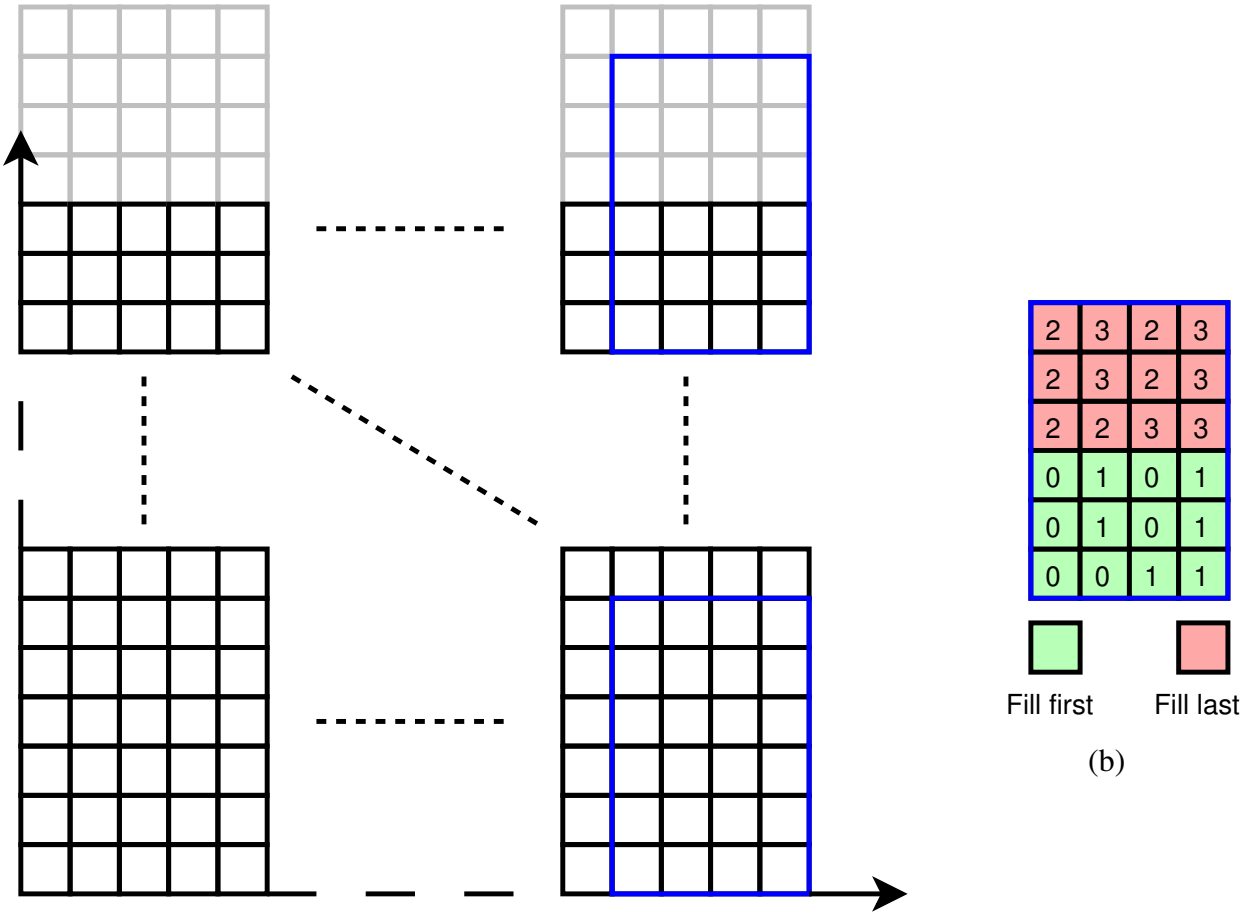
another with the number of elements in the previous rows of super-tile.

$$\text{Offset}_{\text{Acc}_k} = \text{Mem}_O[X_0 \mod a_0, \ldots, X_{d-1} \mod a_{d-1}] + \text{Acc}_\text{H} + \text{Acc}_\text{V} \tag{3.5}$$

Similarly as with the bank access, one can use accumulators and counters instead of costly modulo operation to access all the dimensions of the super-tile every iteration.

It is worth nothing that the internal offset of the super-tile stored in Mem_O can be carefully arranged to reduce memory waste. By keeping the lower offsets in the area of the super-tile that will always be withing the bounds of the matrix under consideration, and assigning the higher ones to the areas that represent the smaller area for the other dimensions, we can effectively reduce the total memory waste. The highest offsets being located in the region of the super-tile less accessed.

In figure 3.12 we see the size of the matrix in one dimension is an integer multiple of the size of the super-tile in dimension d_0, then we will only incur in wasted memory in the last iteration of the complementary loop. With this in mind, we can keep the lowest offsets in the area of the super-tile that are always going to be withing the bounds, thus in the last iteration, such indexes will not be accessed and we have reduced memory waste to 0. The order of the offsets in each of the zones can be arbitrary and does not affect memory waste in any form.



(a)

(b)

Figure 3.12: (a)Tessellation for an m x n matrix using a 4 x 6 super-tile. (b)Mem_O offsets. ©2017 IEEE

The final amount of memory overhead can be calculated by:

$$\text{Overhead} = \sum_{i=0}^{d-1}((\lceil \frac{w_i}{STi} \rceil - w_i) \times \prod_{k!=i} w_k) \tag{3.6}$$

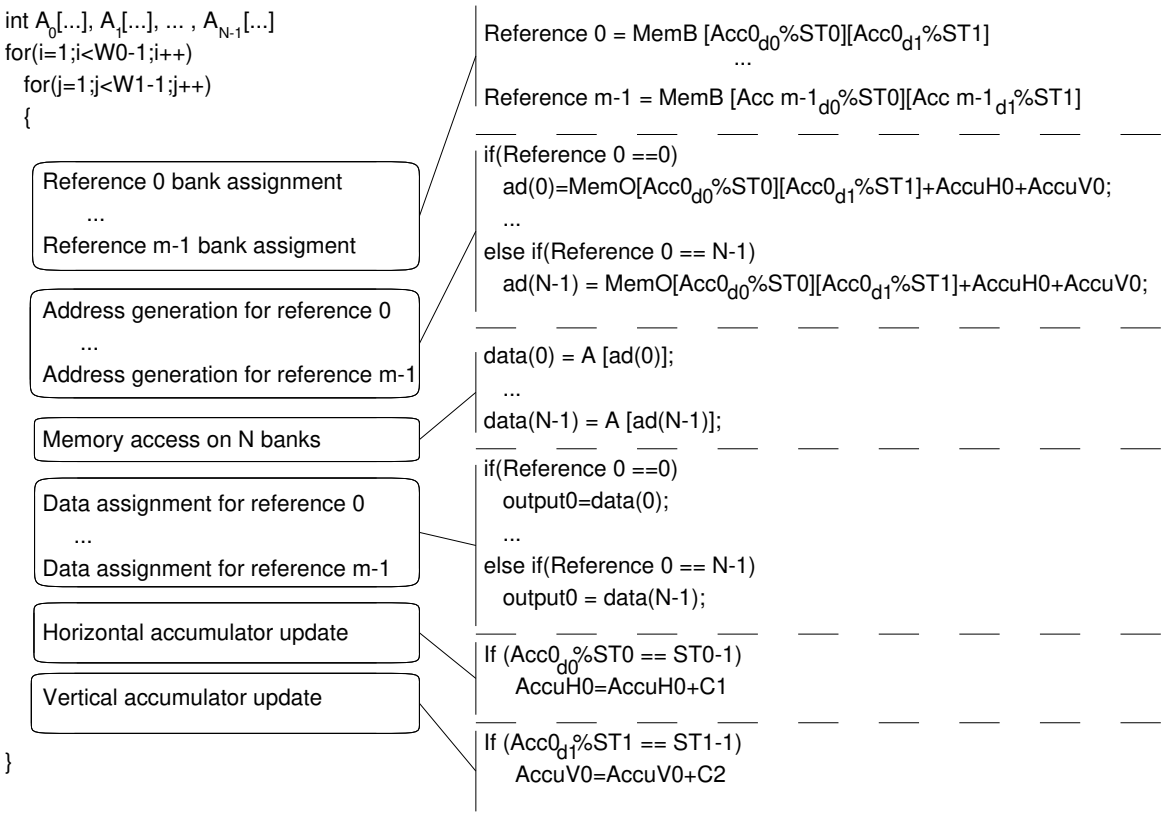


Figure 3.13: Template of transformed code. ©2016 IEEE

To validate the performance benefits of our tessellation-based memory mapping scheme, we follow the workflow of figure 3.4. The first process consists of inputting the memory access patterns of all test benchmarks into a Matlab script which computes the bank assignment and relative offset inside a super-tile for all memory locations. A Matlab script takes the information about the bank and offset super-tile and automatically generates new transformed code in C. This transformed C code is then used as an input to the Vivado HLS 2016.2 from Xilinx, which generates the HDL files in Verilog. The software also automatically generates a Vivado HLx 2016.2 project with the Verilog code already included. This project is synthesized and implemented. This software suite

is also the same tool used to report post implementation resource usage and power estimation. To illustrate, we have listed a transformed code snippet in Fig. 3.13.

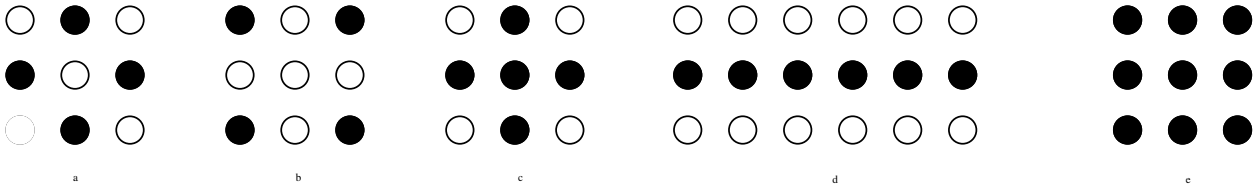


Figure 3.14: Test kernels used: (a) Denoise. (b) Bicubic. (c) Deconv. (d) Motion-LH. (e) Sobel. ©2016 IEEE

Five loop kernels with different access patterns are selected from a wide range of realistic applications, such as medical image processing and H.264 motion compensation, as shown in Fig. 3.14. In our experiments, we mainly focus on the effects brought by different access patterns. The detailed experimental results are shown in Table 3.2 and Table 3.3. To compare, we also implemented the GMP method [3] for the same set of benchmarks. To do a fair comparison the original C code has the same structure for both sets of benchmarks, only calling different functions when testing GMP and Tessellation. In all experimental runs, we turned on the loop pipelining setting in Vivado and set the target throughout with the iteration interval (II) to be 1, which requires all of the memory accesses in the same iteration to be in one clock cycle. For the hardware usage and energy consumption, we chose the target device to be the XC7K160tffg676-3 Kintex-7 FPGA for both Vivado HLS 2016.2 and Vivado Hlx 2016.2, and a bank size of 512 elements each in order to use one full RAMB18E1 block with a data width of 32 bits plus 4 bits of parity in single port mode. The results obtained can be seen in Table 3.2

Table 3.2: Resource utilization and clock period comparison

		CP(ns)	DSPs	FFs	LUTs	Pow.(mW)	Pipeline(#)
Denoise	GMP	2.1	0	254	438	754	8
	Tess(auto)	2	0	367	646	790	11
	Tess(man)	1.9	0	284	440	474	7
	Improv. (%)	4.76 / 9.52	0 / 0	-44.48/ -11.81	-47.48 / ≈0	4.77/ 37.13	
Bicubic	GMP	2.1	0	229	391	738	8
	Tess(auto)	2	0	368	651	789	11
	Tess(man)	1.9	0	276	437	491	7
	Improv. (%)	4.76 / 9.52	0 / 0	-60.7/ -20.52	-66.5/ -11.76	6.91/ 33.45	
Deconv	GMP	2.5	5	1320	1796	710	26
	Tess(auto)	2.7	0	488	1010	570	11
	Tess(man)	2	0	370	633	795	7
	Improv. (%)	-8 20	100 / 100	63 / 71.97	43.76 / 64.76	-19.71/ -11.97	
MotionH	GMP	2.5	6	1783	2867	920	25
	Tess(auto)	2	0	513	1136	1029	12
	Tess(man)	2	0	426	725	951	7
	Improv. (%)	20 / 20	100 / 100	71.23 / 76.11	60.38 / 74.710	-11.84/ -3.7	
Sobel	GMP	2.5	9	3213	5792	1347	24
	Tess(auto)	2.1	0	1082	2337	1595	13
	Tess(man)	2.2	0	606	1416	1340	7
	Improv. (%)	16 / 12	100 / 100	66.32 / 81.14	59.65 / 75.55	-18.41 / ≈0	
Average (%)		7.5 / 14.21	60/ 60	19 / 39.38	10 / 40.65	-12.3/ 10.98	

Using the memory waste formula for GMP [3] and equation 3.6 for the tessellation-based method,

we calculate and compare the overhead of the two methods. The results obtained are shown on table 3.3.

Table 3.3: Memory waste comparison

		Memory overhead (elements)				
		SD 640x480	HD 1280x720	FHD 1920x1080	WQXGA 2560x1600	4K 3840x2160
Denoise	GMP	0	0	0	0	0
	Ours	0	0	0	0	0
	Improv.	0%	0%	0%	0%	0%
Bicubic	GMP	0	0	0	0	0
	Ours	0	0	0	0	0
	Improv.	0%	0%	0%	0%	0%
Deconv	GMP	0	0	0	0	0
	Ours	0	0	0	0	0
	Improv.	0%	0%	0%	0%	0%
MotionH	GMP	960	2880	0	3200	0
	Ours	960	2880	0	32000	0
	Improv.	0%	0%	0%	0%	0%
Sobel	GMP	3840	5040	6480	8000	6480
	Ours	960	720	0	8320	0
	Improv.	75%	85.7%	100%	-4%	100%
Average		15%	17.1%	20%	-0.2%	20%

From table 3.2 we see the tessellation-based method, when implemented directly from the HLS code in general presents positive results, having a reduction of clock period by 7.5 % on average, and reducing the usage DSPs completely, and LUTs, and FFs by 10 and 19% respectively . Only having a small increment of power when compared out implementation of the GMP method [3]. This due to the simpler logic needed to calculate not only the Bank number but also the offset of all the accesses by using the super-tile and accumulators instead of several complex arithmetic and modulo operations.

The results from the Deconv kernel, where we actually get an increase in clock period from the automated results, prompted a deeper analysis of the generated HDL code. We see that the results from GMP are deeply pipelined, sometimes almost doubling the pipeline stages generated for the tessellation based method. With this in mind, we implemented the tessellation-based method in Verilog manually, following the scheme from figure 3.15 and adding one stage between the M:1 and N:1 muxes and two more, just before and after the memories for a total of 7 stages.

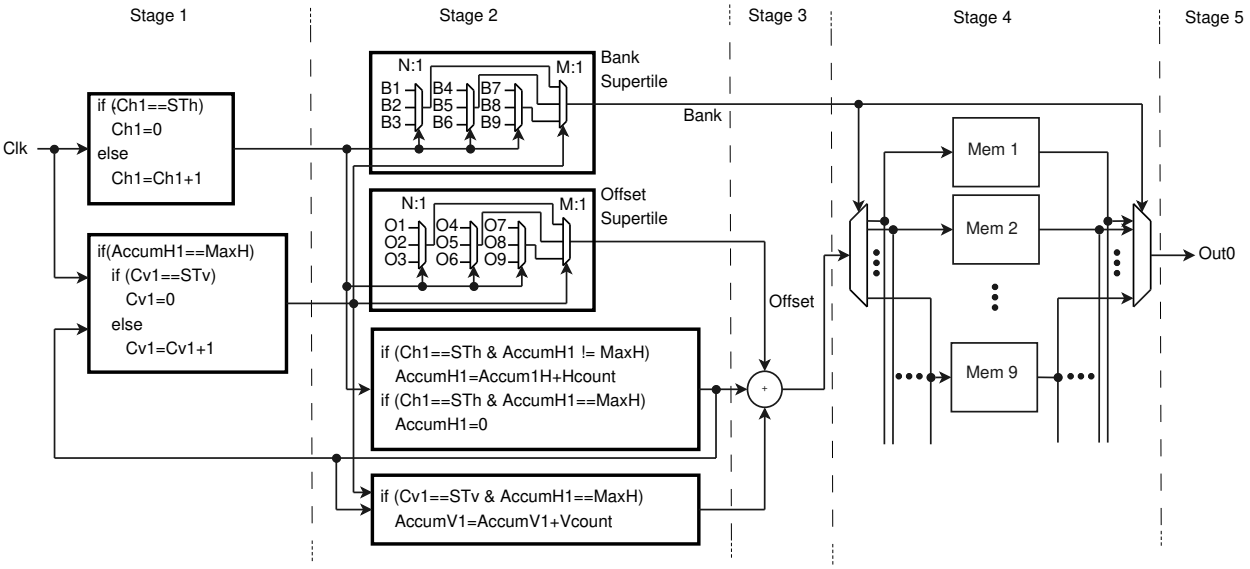


Figure 3.15: Diagram for Tessellation-based method

Comparing the results from the hand-made Verilog code vs. the automated one, we still get a reduction of the clock period but now doubling the average improvement, as well as increased reduction of all comparison parameters.

CHAPTER 4: GRAPH BASED APPROACH FOR OPTIMAL MEMORY BANKING IN STENCILS

This chapter is based on our previously published work in Juan Escobedo and Mingjie Lin, Graph-Theoretically Optimal Memory Banking for Stencil-Based Computing Kernels, Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays DOI 10.1145/3174243.3174251 [1] and encompasses our graph based approach to find the optimal partition factor and usable banking scheme for stencil kernels.

We will begin this chapter with a motivational example that will be used throughout this chapter to demonstrate concepts and exemplify the contribution of our graph-based approach for stencil kernels.

Motivational example

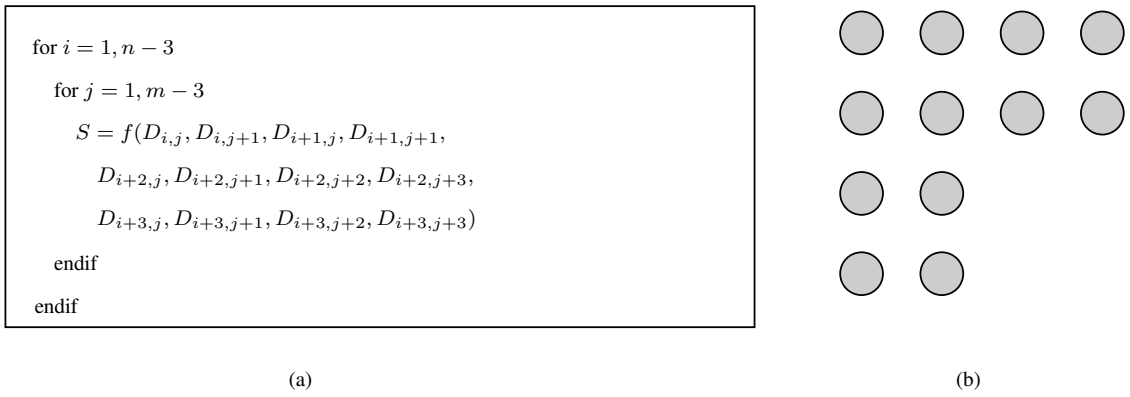


Figure 4.1: (a) Code snippet for a motivational example. (b) 12-point example stencil. [1] DOI 10.1145/3174243.3174251

The code snippet listed in Fig. 4.1(a) contains a computing kernel that produces the stencil depicted in Fig. 4.1(b). The same stencil form has also been considered in [9]. Through directly following the hyper-plane-based memory banking method from [3], we can obtain a partition factor of 14 with a block size of 1, i.e., totally 14 independent memory banks are required to ensure conflict-free memory accessing. The question we ask is: What is the minimally required number of memory banks that full parallelizes all memory accesses while still having a practical memory address mapping scheme? With our graph-coloring-based methodology, we now believe that, 12 memory banks are not only sufficient to guarantee conflict-free memory accesses but also can be formally proven to be optimal.

Our key idea is to use optimal graph coloring to determine the optimal partition factor. Since the problem can be arbitrarily large, doing an optimal coloring of the entire conflict graph is infeasible, even for relatively small problem sizes given the NP-complete nature of the algorithm. To solve this, we attempt to instead optimally only a small induced subgraph of the original entire memory access conflict graph, and subsequently "stitching" this obtained much smaller obtained solution to form the complete coloring solution of the entire conflict graph. In this paper, we generate what we call *the Extended Stencil Graph*. This graph is the smallest graph we need to color to obtain both the optimal partition factor and an usable coloring. The details of our algorithm and the definitions of various graph-related concepts that are essential to our method can found in Section 4. Specifically in Section 4, using the recently discovered theorems in graph theory, we formally prove that the solutions obtained by our algorithm will be optimal.

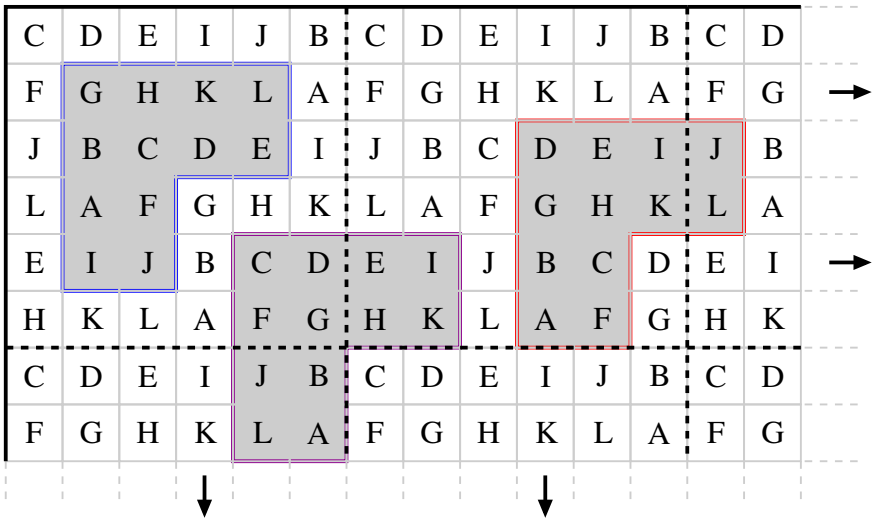


Figure 4.2: Coloring of the data space. Black lines indicate where the pattern repeats. Light gray areas are instances of the stencil. [1] DOI 10.1145/3174243.3174251

To illustrate, our final obtained solution is depicted in Fig. 4.2, where each cell denotes a memory location and its alphabetical label represents its allocated memory bank index. For clarity, we omitted the edges in the figure. The geometry of this graph is the equivalent of considering each memory location in the memory space a node and taking a single node (any node) in it as a reference along all nodes that also belong to any stencils of which the pivot node is part of, keeping their relative positions. As we can see in figure 4.2, this coloring forms a pattern that can be repeated in any direction, and can be used to cover a memory space of any size using only a fixed, and more importantly: finite, coloring scheme. As we can see from Fig. 4.2, after memory banking is finished, if we move stencil instances around (denoted with color blue, purple, and red), in each instance, there is no memory conflict,

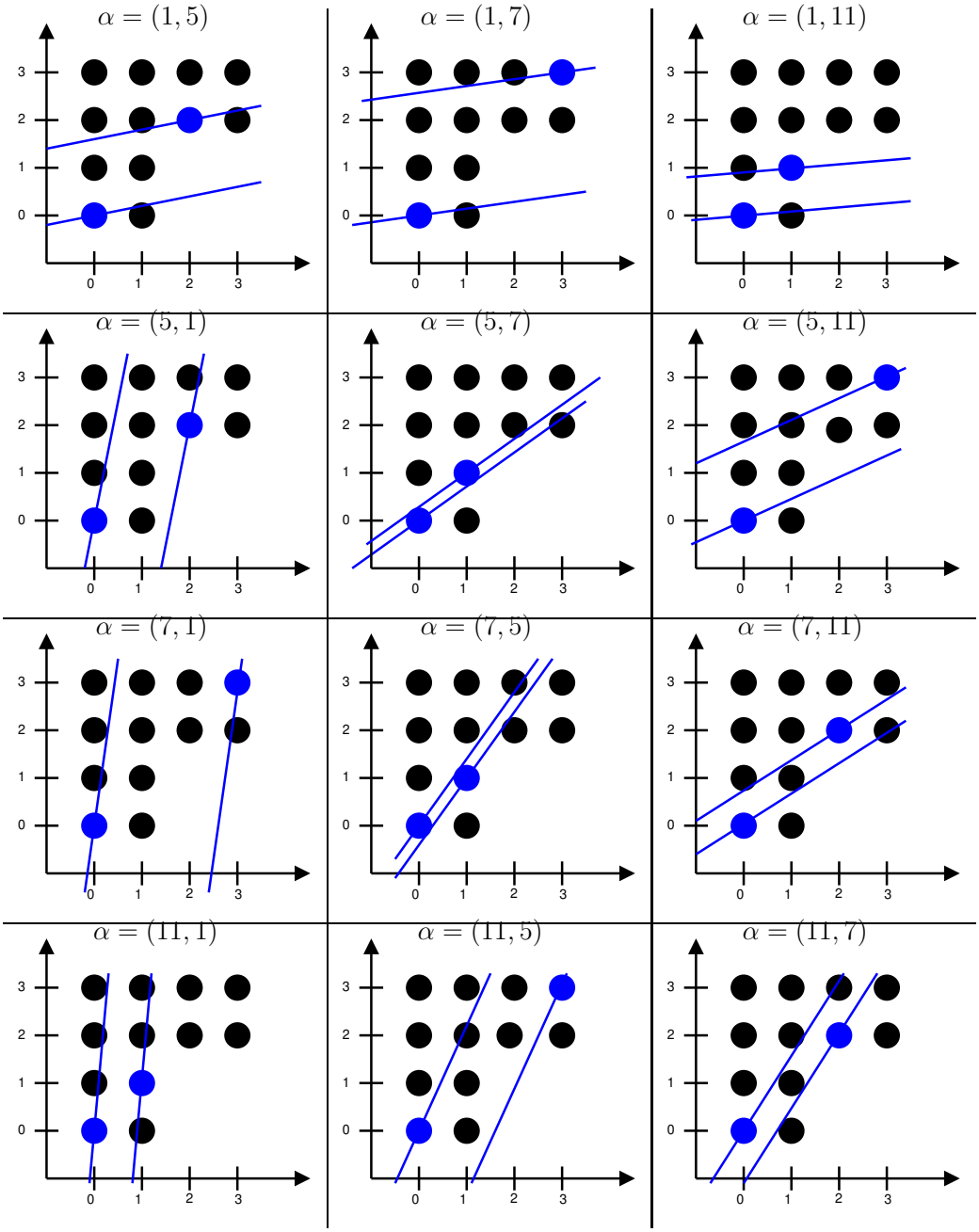


Figure 4.3: Sample hyper-plane families showing at least one conflict. [1] DOI 10.1145 3174243.3174251

We now intuitively explain why the hyper-plane-based approach alone may not produce the optimal

memory banking in this case. In Fig. 4.3, we have enumerated multiple possible hyper-plane families. We found that, no matter what hyper-plane family we use, we always have at least one conflict. This is also supported by the claim made in [9], where the authors studied this particular stencil and found no linear skewing scheme in mod 12 that ensures conflict free partition of the memory space. Also in [16], Cilardo *et. al.* re-stated that having a single hyper-plane family might lead to missing certain solutions with a better partition factor. If we consider the memory banking problem abstractly, all memory locations can be treated as multi-dimensional data points scattered over a multi-dimensional space. All previous methods, including hyper-plane-based and lattice-based [16], attempt to find the most suitable multi-dimensional planes to separate these data points in order to avoid conflicts (or minimize them for a given bank number in the case of [16]). But, these multi-dimensional planes are confined to be affine in mod n, therefore these approaches may miss better solutions which are somewhat non-linear in nature.

Problem Formulation and Overall Solving Strategy

The central task of high-level synthesis is to transform a C-like software code segment into efficient and high-performance hardware circuit description. Typically, within a given code segment, the largest percentage of computation will be concentrated on iteratively executing a small-size computing kernel, which often access multiple data items stored in one or several data arrays as shown in Fig. 4.1. Mathematically, during each iteration, all the memory accesses within such a kernel K can be defined as a set of m data points $P = \{\vec{A_0}, \vec{A_1}, \ldots, \vec{A_{m-1}}\}$, where each data point $\vec{A_.}$ is stored in a d-dimensional array structure. The main objective of memory banking is to distribute all array elements into multiple independent memory banks such that fully parallel memory accesses can be enabled. In other words, during any iteration i, all accessed memory points in P will be read from totally independent memory banks with zero memory reading conflicts. Mathematically,

the memory banking problem of an n-dimensional array can be defined as finding a pair of mapping functions $f(\vec{x})$ and $g(\vec{x})$, where $f(\vec{x})$ assigns a distinct memory bank and $g(\vec{x})$ generates its corresponding intra-bank offset for a given data element, respectively. Clearly, an access conflict between two memory references $\vec{x}_j$ and $\vec{x}_k$ occurs if $f(\vec{x}_j) = f(\vec{x}_k)$, which means accessing the same memory bank during the same iteration. Again, we assume single-port memory banks here. As such, the memory banking problem under our consideration consists of two mapping problems: memory bank mapping and intra-bank offset mapping.

Formally:

Problem 3 (Bank minimization) *Given an l-level loop on the iteration domain $\mathcal{D}$ with m affine memory references $\vec{x_0}, \vec{x_1}, \ldots, \vec{x_{m-1}}$ on the same array, find a partition factor N such that:*

$$\textit{Minimize} : N = \textit{max}_{0 \leq n < m}\{f(\vec{x}_i)\} \tag{4.1}$$

$$\textit{s.t. } \forall \vec{x}_j, \vec{x}_k \in \mathcal{D}, f(\vec{x}_j) \neq f(\vec{x}_k), 0 \leq j < k < m$$

Eqn. 4.1 defines the objective function of memory partitioning, ensuring no access conflict between any two references. After bank mapping, a data element in the original array should be allocated a new intra-bank location. For correctness, two different array elements will be either mapped onto different banks or the same bank with different intra-bank offsets. An intra-bank offset function is valid if and only if:

$$\forall \vec{x}_j, \vec{x}_k \in \mathcal{D}, \vec{x}_j \neq \vec{x}_k \rightarrow (f(\vec{x}_j), g(\vec{x}_j)) \neq (f(\vec{x}_k), g(\vec{x}_k))$$

which means either

$$f(\vec{x}_j) \neq f(\vec{x}_k) \; or \; f(\vec{x}_j) = f(\vec{x}_k), g(\vec{x}_j) \neq g(\vec{x}_k)$$

Problem 4 (Storage minimization) *Given an l-level loop on the iteration domain $\mathcal{D}$ with m affine memory references $\vec{x_0}, \vec{x_1}, \ldots, \vec{x_{m-1}}$ on the same array, find a partition factor N, find an intra-bank offset mapping function g with minimum storage requirement S such that:*

$$Minimize : \sum_{j=0}^{N-1} max_{\forall i s.t. f(\vec{x}_i)=j}(g(\vec{x}_i)) \tag{4.2}$$

$$s.t. \; \forall \vec{x}_j, \vec{x}_k \in \mathcal{D}, \vec{x}_j \neq \vec{x}_k \rightarrow (f(\vec{x}_j), g(\vec{x}_j)) \neq (f(\vec{x}_k), g(\vec{x}_k))$$

Eqn. 4.2 defines the objective function of partitioning with minimum storage overhead, ensuring a valid partition.

Theoretically, one can optimally solve the memory banking problem by optimally coloring the complete memory access conflict graph. Here, a memory access conflict graph is generated by considering all accessed memory locations as nodes and adding edges between nodes that need to be accessed together, forming a clique, during a particular iteration for all iterations in the iteration domain. In other words, whenever two array elements M_i and M_j are accessed during the same iteration, we consider them are conflicting and need to allocated into two different memory banks. Here, we assume all memory banks to be single-ported for simplicity. After solving the optimal single-port memory banking problem, it can be readily shown that multiple-port memory banking can be solved by defective coloring scheme with the same strategy.

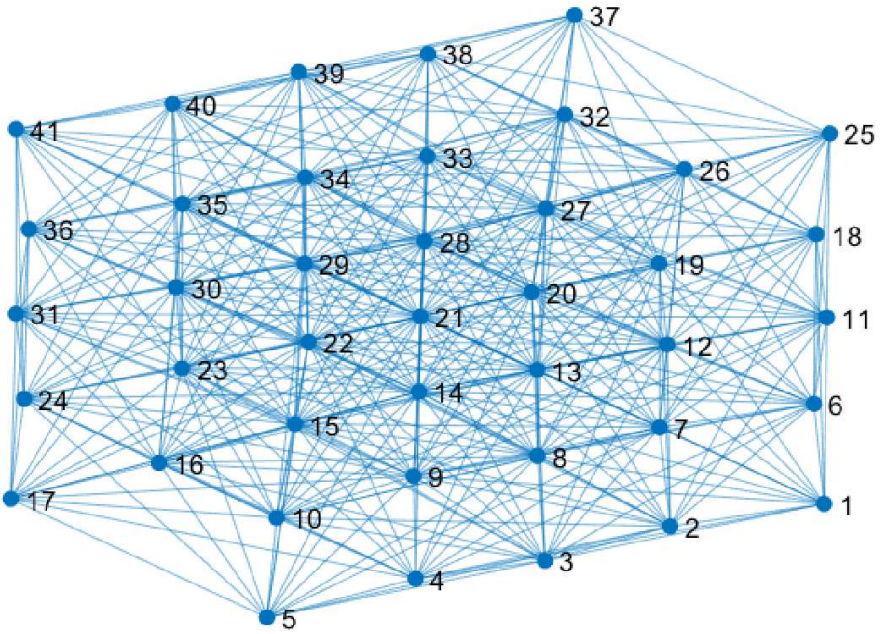


Figure 4.4: A small portion of the entire memory access conflict graph generated by the stencil in Fig. 4.1. [1] DOI 10.1145/3174243.3174251

The biggest advantage of such graph-coloring-based approach is its generality, i.e., this method imposes no limitation on the regularity of loop structure or the affiness of memory accesses. Unfortunately, in practice, this graph-based methodology is infeasible due to two critical issues. First, for any realistic code segment of loop, the size the memory access graph quickly increases with its associated array size, therefore simple too big for any existing graph coloring algorithm to handle optimally. For example, for the 12-point stencil S shown in Fig. 4.1(b), even a tiny portion of its complete memory access conflict graph becomes quite complex as shown in Fig. 4.4, thus infeasible to optimally color. Second, even if we can optimally solve this graph coloring problem, the resulted memory address mapping will simply be too large, therefore completely impractical to implement with hardware.

In this paper, we first limit our research scope to only the stencil-based kernel code. As such, all memory accesses we consider will be affine, which means that the array index of each d-dimensional memory access $\vec{x} = (x_0, x_1, \cdots, x_{d-1})^T$ we consider is a linear transformation of a l-dimensional iteration vector $\vec{i}$ in the form of:

$$\vec{x} = A_{d\times l} \cdot \vec{i} + \vec{C}$$

$$A_{d\times l} = \begin{bmatrix} a_{0,0} & \cdots & a_{0,l-1} \\ \vdots & \ddots & \vdots \\ a_{d-1,0} & \cdots & a_{d-1,l-1} \end{bmatrix}, \vec{C} = \begin{bmatrix} a_{0,l} \\ \vdots \\ a_{d-1,l} \end{bmatrix}$$

where $A_{d\times l}$ is a coefficient matrix, $a_{k,j} \in \mathbb{Z}$ is the coefficient of the j-th iteration vector on the k-th dimension, and $\vec{C}$ is a a column vector with constants. Moreover, within each kernel K, the relative displacements between all memory locations will be invariant throughout all iterations.

By focusing on only stencil-based computing kernel, we show in the following that the repeatability of a complete memory access conflict graph can be exploited to produce a much smaller-sized kernel expansion graph. As such, the optimally coloring the complete conflict graph can be greatly reduced to optimally coloring a much smaller graph, which typically is only within two times of the kernel size. Furthermore, the reduced graph coloring method also will only require a much smaller memory address mapping, thus completely practical for hardware implementation. Maybe most importantly, to the best of our knowledge, our graph-theoretic approach is the first work that offers the provable optimality of its memory banking solutions.

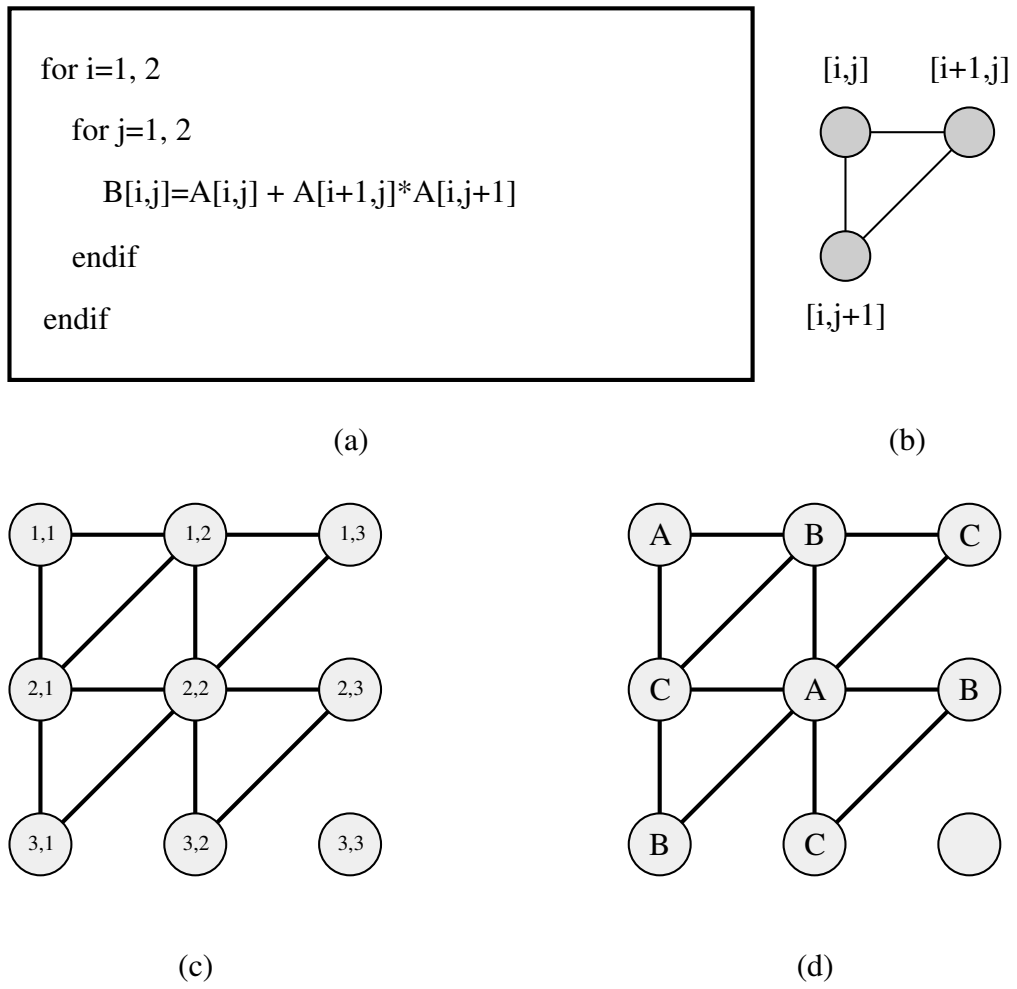


Figure 4.5: (a) Kernel code snippet. (b) Stencil S. (c) Extended stencil graph ($ESG(S)$). (d) Optimal coloring $ESG(S)$. A, B, and C denote different colors. [1] DOI 10.1145/3174243.3174251

Before describing our algorithm in detail, we mathematically define several important concepts in our methodology. Let $\mathcal{G} = (V, E)$ denote the memory access conflict graph, coloring $\mathcal{G}$ is the process of assigning each vertex $v \in V$ with a distinct color, such that no two adjacent vertices connected by an edge $e \in E$, have the same color. In mathematics, if a graph $\mathcal{G}$ can be colored with k colors, the graph is termed as k-colorable, and the smallest k for which the graph $\mathcal{G}$ is

k-colorable is defined as the chromatic number of $\mathcal{G}$, denoted as$\chi(\mathcal{G})$. For example, in Fig. 4.5, we present a tiny example of stencil-based kernel. Given the code snippet in Fig. 4.5(a), a 3-point stencil depicted in Fig. 4.5(b) can be readily extracted. During each loop iteration, three memory locations will be accessed in parallel. As such, we construct a clique with three conflict edges. If we iterate through all iterations, we will obtain the total memory access conflict graph G_0 in Fig. 4.5(c). If we proceed with optimally coloring this conflict graph, it can easily computed that only three colors needed to color this conflict graph, therefore G_0 is 3-colorable and its chromatic number is 3. Despite of the elegance of solving the small problem in Fig. 4.5, unfortunately, it is well-known that even determining if a graph $\mathcal{G}$ is k-colorable for $k \geq 3$ proves to be NP-complete [27]. This makes solving a realistic large-scale memory banking problem infeasible.

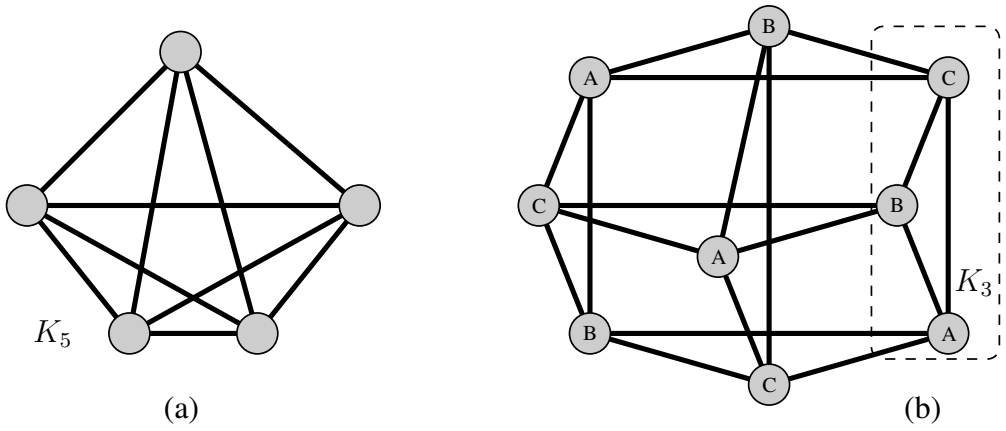


Figure 4.6: (a) A 5-node clique K_5. (b) A perfect graph of 9 nodes. [1] DOI 10.1145/ 3174243.3174251

In addition, two graph theory concepts are essential to the development of our memory banking algorithm. First, in graph theory, a clique is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent; that is, its induced subgraph is complete. Cliques are one of the basic concepts of graph theory and are used in many other mathematical

problems and constructions on graphs. One example of 5-node clique K_5 is shown in Fig. 4.6(a). It can be noted that, within our memory conflict graph, any instance of one stencil will clearly form a clique because all nodes inside this particular stencil instance will conflict with each other, i.e., they can not be allocated into the same memory bank in order to avoid any access conflict. Second, a perfect graph is a graph in which the chromatic number of every induced subgraph equals the size of the largest clique of that subgraph. One example of 9-node perfect graph is depicted in Fig. 4.6 (b) and shows a lot similarity with a typical extended stencil graph in topology. One important consequence of a perfect graph is that its optimal coloring can be solved in polynomial time. As shown in Fig. 4.6(b), all 9 nodes in the perfect graph $\mathcal{G}$ are colored optimally with exactly three colors A, B, and C, hence $\chi\{\mathcal{G}\} = 3$. More importantly, when a graph proves to be perfect, we can be sure its chromatic number will be equal to its largest clique number. Note that in Fig. 4.6 (b), $\mathcal{G}$ has six equal-sized cliques, each of which consists of 3 nodes.

One crucial observation in this paper is that, the stencil-induced memory access conflict graph is formed by combining many cliques corresponding to all loop iterations. The key to the success of our memory banking algorithm is to exploit the special graph-theoretical property of our target memory access conflict graph while avoiding the NP-hardness of optimally coloring a large-sized complete memory access conflict graph.

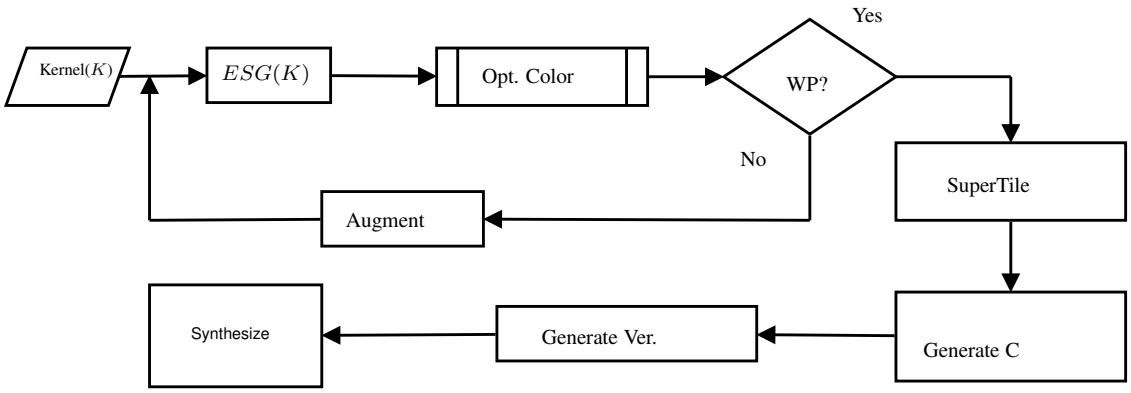


Figure 4.7: Flow diagram of our algorithm. [1] DOI 10.1145/3174243.3174251

Fig. 4.7 presents the overall flow diagram of our graph-based algorithm. Given a stencil-based computing kernel S, we first construct its Extended Stencil Graph or $ESG(S)$. We then optimally color this $ESG(S)$ using the Matlab toolbox from [28] and obtain its chromatic number $\chi(ESG(S))$. If $\chi(ESG(S))$ equals to the number of nodes in the given stencil S, i.e., the extended stencil graph $ESG(S)$ is perfect, we complete our memory banking scheme by allocating each array element to a distinct memory bank denoted by a distinct color. In Section 4, we will prove that not only this coloring scheme is optimal but also the resulted coloring results can be repeated to cover the whole memory array space. Otherwise, if $\chi(ESG(S))$ is larger than the stencil size, i.e. $ESG(S)$ is not perfect, we will show the the entire conflict graph can be colored with that many colors, but we make no claims on the existence of an usable mapping function we will start to modify the given stencil by adding more nodes. During each iteration, we treat the modified stencil shape as a new one and repeat the above steps. This iterative process will stop if the resulting kernel graph becomes perfect. More details about how we iteratively modify the stencil, why this algorithm is guaranteed to terminate, and why the resulting solution will be optimal will be further discussed in Section 4.

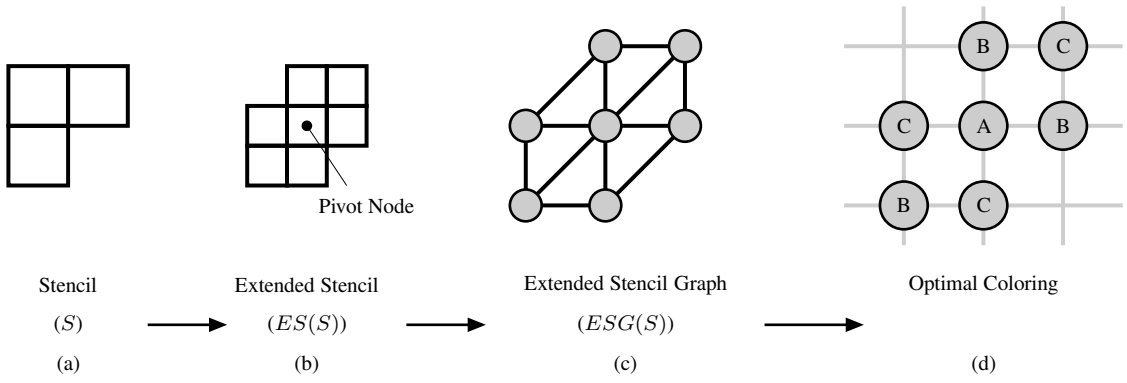


Figure 4.8: (a) An 3-point stencil example S. (b) Its extended stencil $ES(S)$. (c) Its extended stencil graph $ESG(S)$. (d) Optimal coloring of $ESG(S)$. [1] DOI 10.1145/3174243.3174251

Clearly, the key component in our algorithm is to construct the expanded stencil graph ESG of a given kernel-induced stencil. This construction consists of two steps. First, we construct an expanded stencil by overlapping multiple stencil instances that intersects or touches a fixed point. For example, exactly three instances of the given stencil depicted in Fig. 4.8(a) forms an expanded version of stencil in Fig. 4.8(b). Subsequently, by selecting all nodes in this expanded stencil, we can readily induce a subgraph from the overall memory access conflict graph. In graph theory, an induced subgraph of a graph is another graph, formed from a subset of the vertices of the graph and all of the edges connecting pairs of vertices in that subset. By including all necessary conflict edges, we then construct a graph, which we termed as the extended stencil graph ($ESG(S)$) in Fig. 4.8(c). Finally, we perform the optimal coloring on $ESG(S)$ in order to obtain the coloring of each node. In the next section, we will formally prove that (1) the chromatic number of an extended stencil graph equals the the chromatic number of a total memory access conflict graph and (2) the obtained coloring can be repeatedly utilized to optimally color all memory space. Note that both results are significant because the algorithm listed in Fig. 4.7 not only can be proven to be optimal but also, because typically the size of expanded stencil graph is much small than the

complete memory conflict graph, the optimally memory banking problem can be efficiently solved.

Proof of Algorithmatic Optimality and Hardware Implementation Efficiency

Our optimal memory banking methodology largely depends on effectively manipulating memory conflict graph through exploiting its special properties. In this section, we formally prove two key theorems that ensure the optimality of our memory banking solutions and the practical guarantee of hardware efficiency. Specifically, we will prove that, for any given kernel stencil, our methodology will always find the optimal memory banking scheme with the smallest possible number of memory banks. Furthermore, we will prove that, for any given kernel stencil, we only need to optimally color an extended stencil graph, only a small portion of the overall memory access conflict graph, and the complete graph coloring problem can be solved by repeatedly "stamping" this small coloring throughout the overall conflict graph. This ensures that the required memory banking indexes and offsets can be computed efficiently.

Minimum Memory Bank Number

When constructed as in Section 4, a given extended stencil graph ESG has three important properties. First, any given ESG has a pivot node in the center, which a number of stencil instances surround. Moreover, by definition, the number of stencil instances equals the number of nodes in a given stencil. Second, an expanded stencil includes all possible scenarios how two stencil instances intersect. This is important because this loosely but intuitively explains why optimally coloring an expanded stencil graph, an induced subgraph, infers optimal coloring the entire memory access conflict graph. Third, graphically, an expanded stencil graph combines multiple subgraphs, each of which is formed by an individual stencil. In fact, the induced conflict subgraph generated by

each stencil is a graph clique. Moreover, because each stencil contains the same number of nodes, all cliques found in an extended stencil graph are equal in size. This property turns out to be critical to our optimality proof.

Following the steps in Fig. 4.8, it is clear that a complete memory access conflict graph can be readily partitioned as a number of extended stencil graphs. This is expanded and proved in section 4. Furthermore, these extended stencil graphs are connected with each other through stencil-induced conflict graph cliques. Without loss of generality, let us consider two individual extended stencil graphs ESG_a and ESG_b that joined at a stencil-induced clique K_n, where n is the clique size. According to the theorem in [29], in general, for graphs G_1 and G_2, $\chi(\frac{G_1+G_2}{K_n}) = \max\{\chi(G_1), \chi(G_2)\}$, where operation $\chi(\cdot)$ denotes finds chromatic number and $\frac{G_1+G_2}{K_n}$ denotes a joined graph of G_1 and G_2 at a complete graph K_n. Fortunately, in our case, a clique generated by a n-node stencil is trivially a complete graph K_n. Additionally, two individual extended stencil graphs ESG_a and ESG_b under our consideration is isomorphic, thus $\chi(ESG_a = \chi(ESG_b) = c$. As such, we conclude that $\chi(\frac{ESG_a+ESG_b}{K_n}) = \max\{\chi(ESG_a), \chi(ESG_a)\} = c$. In other words, expanding one extended stencil graph through joining at a stencil-induced clique will preserve the chromatic number of the original extended stencil graph. Therefore, by continuously expanding a starting extended stencil graph, we can cover the whole memory space and reconstruct the entire memory access conflict graph. Because we know that the starting chromatic number is optimally obtained by coloring a extended stencil graph, the inferred chromatic for the entire conflict graph must also be optimal.

Graph Repeatability

Previous section proves that, given a stencil S, the chromatic number of its extended stencil graph $\chi(ESG(S))$ equals to the optimal partition factor of the entire memory access conflict graph, i.e.,

the minimum number of independent memory banks are needed to ensure conflict-free memory accesses throughout all loop iterations. However, this result doesn't by itself provide a valid and hardware-efficient memory address mapping. In the following, we prove by construction that the optimal coloring of an extended stencil graph repeats itself across the entire memory domain, thus providing a very efficient memory address mapping scheme if the considered extend stencil graph is perfect. Otherwise, we need to augment the original stencil by adding more nodes until the induced extended stencil graph becomes perfect. We will prove that such augmentation will terminate and always have an optimal solution.

As mentioned in Section 4, an extended stencil graph is formed by overlapping multiple stencil instances. By definition, each extended stencil graph ESG has a center point p. In addition, the conflict graph of each stencil constitutes a clique and the coloring of each node contained in this clique is unique. We now show that, a valid optimal coloring of a given ESG can be readily expanded in all directions with repeated patterns. Clearly, if this is true, we only need to store a small-size coloring in order to infer every node's color assignment.

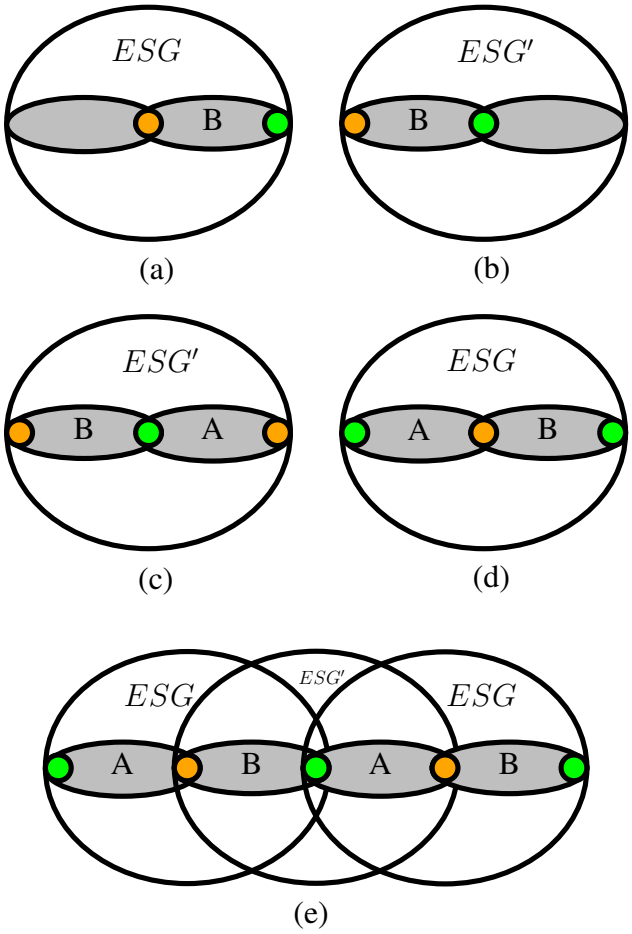


Figure 4.9: (a) Instance of the ESG with two cliques colored.(b) Instance of the gluing ESG (ESG') with the coloring reversed.(c) continuous sequence ESG-ESG'ESG- with repeated coloring. [1] DOI 10.1145/3174243.3174251

In Fig. 4.9, we consider one extended stencil graph ESG. Pick one of its cliques and call its particular coloring B. By definition, B assigns a distinct color to all the nodes in this clique. In particular, its previously defined "pivot" node is colored in orange, which will be shared by the other cliques, generating a partial coloring. Note that, even after fixing its "pivot" node with a specific color, any given n-size clique will still has $(n-1)!$ possible valid colorings. Without loss of

generality, let us select exactly one of the other $n-1$ nodes from this fully colored clique with B and define this to be a "link" node colored in green. Now consider another extended stencil graph ESG' and again choose two cliques in it. Note that these two cliques may not be the same ones considered previously. One of these cliques can be chosen such that it has the same coloring B, but its "link" node is now the "pivot" node of this particular ESG as seen in Fig.4.9(b). If the ESG is perfect, then all the cliques must have all the colors, so there must exist another clique in ESG' such that it contains the "pivot" node, keeping the corresponding green color in that particular location, and can keep the partial coloring from the clique in ESG plus (color that particular node in orange). Let's call this coloring of a clique "A". This is one of the (n-1)! previously mentioned valid colorings. This arrangement can be seen in 4.9 (c). We can then use coloring "A" to color the original partially colored clique from ESG since it preserved that partial coloring, fig. 4.9 (d). From this it follows that we can "glue" ESG with ESG' by an induced subgraph that includes the maximum clique colored "B". Let's call the resulting graph $<ESG$+$ESG'>$. Since both graphs are perfect with the same chromatic number, and we glue them by an induced subgraph that includes a maximum clique then $<ESG$+$ESG'>$ is also perfect and retains the same chromatic number. We can now repeat the procedure but now gluing $<ESG$+$ESG'>$ to ESG by an induced subgraph that includes the maximum clique colored "A", generation $<ESG$+ESG'+$ESG>$, which again retains perfectness and the chromatic number for the aforementioned reasons. We can continue doing this indefinitely. Since we only need one intermediate gluing graph ESG' before we can reuse the coloring of ESG, then it is obvious that the coloring has a maximum distance until it repeats. In the worst case, this distance is the width of the ESG in that particular direction.

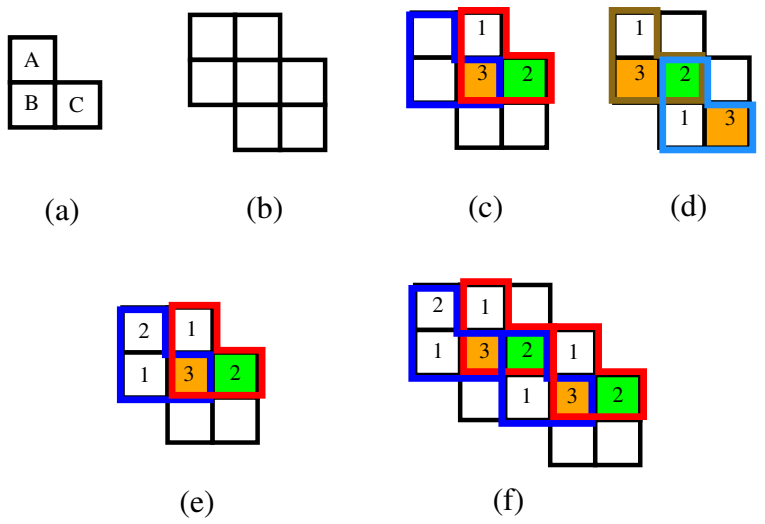


Figure 4.10: (a) An example stencil. (b) Extended stencil graph of the sample stencil. (c) ESG with valid coloring for one clique. (d) Glue extension graph ESG' with two cliques colored. (e) ESG with valid colorings for two cliques (f) Glued chain $ESG - ESG' - ESG$ for the sample stencil. [1] DOI 10.1145/3174243.3174251

This idea can be seen in more detail with a real stencil in figure 4.10.Here (a) is a three point stencil with nodes A,B, and C. (b) Shows the ESG of the sample stencil. In (c) we see a partial coloring of the ESG with labels 1,2 and 3. Only the red clique is fully colored. The blue clique only has node C colored. The center node of the ESG is the node shaded in orange, is assigned label 3, and the "Link" node in green, is assigned label 2. Now, in (d) we see another instance of the ESG. The only clique where the coloring from Red will make the "Link" node to be the "pivot" node is the clique in brown. Since all cliques must have all coloring labels due to the chromatic number of the graph being equal to the number of nodes in the stencil, then it follows that there must exist a clique such that it has node C labeled 3 . Note that we have two options ((N-1)! with n=3), one with label 2 in position B, and another with the label 2 in position A, light blue. Only placing label

2 in position A allows for a valid coloring of the entire ESG, generating the coloring from the light blue clique. We can use this coloring to complete the partial coloring of the blue clique, since the ESG includes all interactions of a stencil with another (e). Finally in (f) we can see how we can indefinitely glue the graphs (by means of ESG') while keeping the same chromatic number and with a repeating coloring. On top of that, since this graph is perfect, then there exists a polynomial time algorithm to color it.

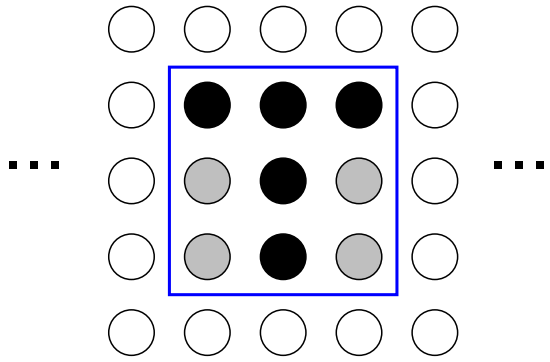


Figure 4.11: Stencil (black) and the circumscribing square (blue). Nodes in gray are potential candidates to be added. [1] DOI 10.1145/3174243.3174251

If, for a given stencil S, its extended stencil graph $ESG(S)$ is not perfect, i.e., $\chi(ESG(S)) > w(S)$, where $w(S)$ denotes the size of the stencil S. Unfortunately, the complete coloring of the entire memory access conflict graph can not be obtained by repeating the optimal coloring of the extended stencil graph. In this case, we have to iteratively add nodes to augment the original stencil until the resulting ESG of this modified stencil becomes perfect. To do this, we consider the smallest $m \times m$ circumscribing rectangle of the original stencil, perform an iterative search, and consider all C_n^k possible combinations of added nodes from inside the square, as seen in figure 4.11, where n is the number of nodes in the square but not in the stencil and for all k from 1 to n in incremental order to find a solution with the best chromatic number (and associated number of

added nodes) that produces a perfect extended graph. Since we have a finite candidate solutions, this algorithm will terminate. However, why this search will result the optimal solution, i.e, the minimum number of memory banks to guarantee conflict-free coloring and periodic mapping, remains to be proven.

Data: Stencil(P)

Result: Periodic coloring(PC)

CS $\leftarrow$ smallest m$\times$m square such that P $\subset$ CS ;

TN $\leftarrow$ CS - P;

for i $\in$ 1...$|TN|$ **do**

  forall EN s.t. EN $\subseteq$ TN and $|EN|$==i **do**

    EG $\leftarrow$ EG(P+EN);

    if x(EG)==$|P|$+i **then**

      Terminate and return Optimal Coloring(EG);

    end

  end

end

Figure 4.12: Node addition algorithm. [1] DOI 10.1145/3174243.3174251

Fig. 4.12 presents the algorithm that performs the above iterative search. Note that if we add more nodes (and the corresponding edges) the original graph, with the original stencil, becomes a common induced subgraph. This means that if a valid periodic coloring for the new graph is found, then this solution will also solve the desired periodicity coloring scheme desired for the original one. We now prove that this algorithm results in the optimal partition factor. The following proof consists of two steps.

First we prove that this search algorithm terminates. Because the circumscribing square contains finite number of nodes, we only have a finite number of possible candidate nodes sets to consider. Additionally, the final square augmented stencil is guaranteed to produce perfect extended stencil graph. This is because that any $m \times m$ square is always going to generate a perfect ESG since the Rook's graph has m^2 and can be directly mapped to an order m^2 Latin Square where every instance of the $m \times m$ stencil is surrounded by 3 or more same sized neighbors to form the aforementioned Latin square. One example of this condition is seen in the Sudoku game where 3×3 stencils form a larger 9×9 grid where each row and column has all the 9 symbols without repetition, but more importantly, each of the non-overlapping 3×3 squares has a permutation the 9 symbols, making it also a Gerechte square. Thus we know that since the geometry and maximum amount of nodes (and the largest number of colors) needed to obtain a periodic coloring for a given arbitrary stencil is the smallest circumscribing square that contains the stencil, then we can ensure that we will find at least one solution which is the all the nodes in the circumscribing square itself. In other words, we will always find a solution in the circumscribing square. This will also provide the upper limit to the number of colors needed to generate a periodic coloring.

Second, we argue optimality because of the incremental nature of the search in the solution space. We first try all sets of added nodes with cardinality 1 and increase it until we find an answer that generates a perfect Extended graph. We need to show that no better solution can be obtained by selecting nodes outside of it. We proceed to demonstrate by contradiction, using the properties of the Extended graph, that no such node exist that selecting it, will provide a better solution in terms of chromatic number. Assume the best solution found S_i has i added nodes from the circumscribing square with chromatic number k. Now assume that there exists a node(or set of nodes) outside the circumscribing square such that considering it would yield a periodic coloring with $i - 1$ added nodes and $k - 1$ colors. Given the vertex-transitive property of Rook's graphs, and the fact that this property is retained when the reduced graph is perfect, one can switch the relative position of

the selected node(s) with nodes inside the circumscribing square and obtain an isomorphic graph using nodes entirely inside said region. Thus, since the search is done incrementally, considering all combinations of increasingly large sets of nodes inside the square, if such solution with $i-1$ nodes and $k-1$ colors exist, it must have been considered when exploring the previous sets considering only points inside the circumscribing square.

Data Reuse

Accessing off-chip data is a very costly operation. While reading on-chip data can take just a few cycles with minimum latency, off-chip communication comes at the cost of very high latency and possibly hundreds of wasted clock cycles. Luckily, stencils have the peculiarity that in most cases part of the data accessed on one iteration will be needed in the next. This allows for a very straightforward data reuse scheme. To add data reuse ot our graph-based approach we first need to define two concepts widely used when performing data reuse analysis.

The first concept is reuse distance. Reuse distance is a metric of temporal locality that expresses in how many iterations the data accessed by one of the memory accesses in a stencil will be reused by another, if any.The second one is the data reuse graph, which is used to easily encode the reuse distances. The data reuse graph or DRG is a directed acyclic graph or DAG, meaning a graph where the edges connecting the vertices have a defined starting and ending point and it is impossible to follow the edges to from a path that returns you to the original starting vertex. The DRG is a weighed DAG where the weights represent the reuse distance between the memory accesses. If we use the code and associated stencil from figure 4.5 (b), we will obtain the DRG from figure 4.13.

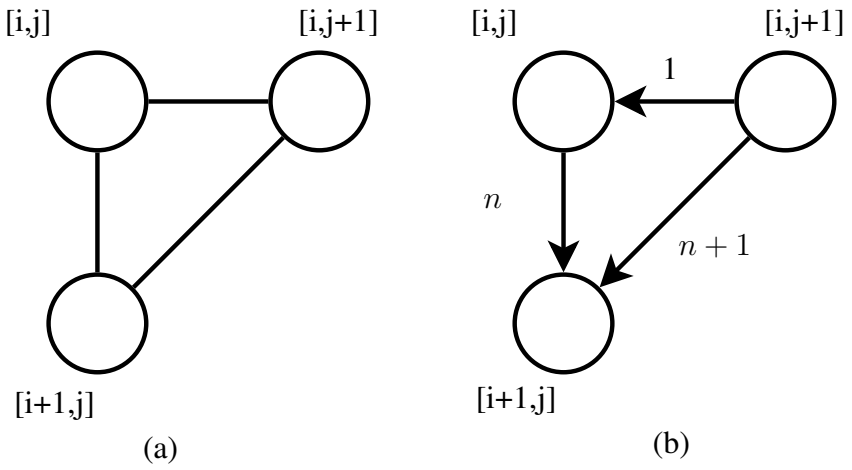


Figure 4.13: (a) Stencil clique. (b) DRG for the stencil where n is the problem size in the inner most loop.

One of the key properties of the Data Reuse Graph is the fact that it is edge transitive. This means that for any given graph G there exists a transformation such that, given any two edges e_1 and e_2 of G, there is an automorphism of G that maps e_1 to e_2, where automorphism is a form of symmetry in which the graph is mapped onto itself while preserving the edge - vertex connectivity [30]. In other words, under certain transformation, we can remove some edges while still maintaining reachability from one node to the other. This operation is called transitive edge reduction and will be the key to maintain the properties of the original ESG when working with the simplified stencil while allowing for a reduction of the clique size and thus partition factor.

We see an example of this process in figure 4.14. In (a) the 5-point stencil under considerations. Analyzing the geometry of this stencil and given the loop bounds, one can construct the DRG seen in (b). After performing the transitive edge reduction, we obtain the graph from 4.14(c). Note that any node reachable from any other one is still reachable and the path length (sum of the weight of the edges in the path) is the same as the optimal path in the original DRG.

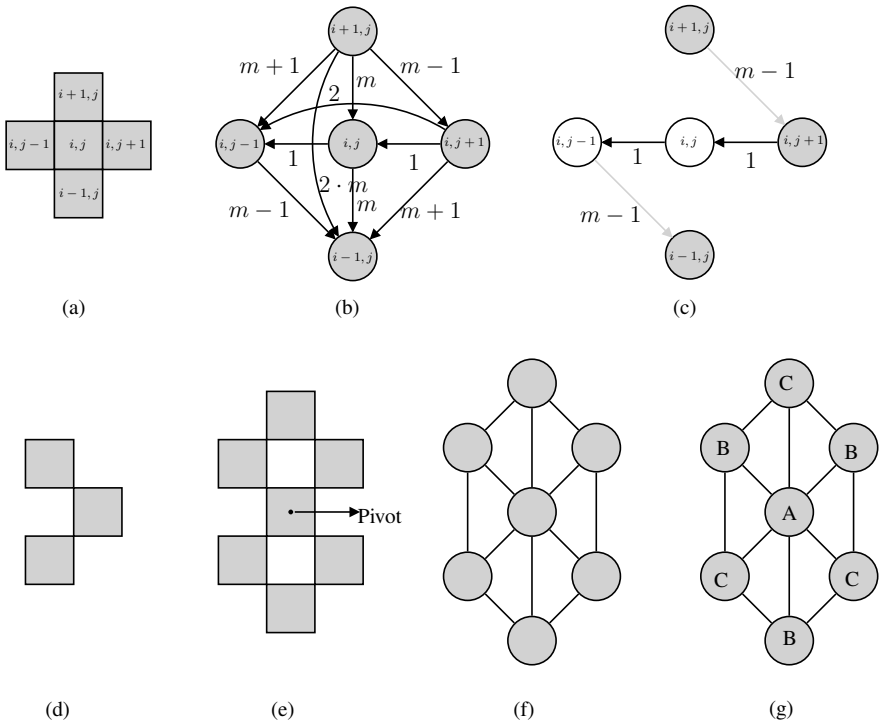


Figure 4.14: (a) Original stencil. (b) DRG. (c) DRG after edge reduction. In black are the edges independent of the the problem size. (d) Simplified stencil. Nodes without an incoming edge. (e) ES for the reduced stencil. (f) Corresponding ESG. (g) Colored ESG. Note we now only need 3 colors instead of the original 5.

Fig.4.15 presents the overall flow diagram of our graph-based algorithm when including data reuse. Given a stencil-based computing kernel S, we first perform a reuse analysis by generating the DRG of the stencil and running a transitive edge reduction algorithm on it. The remaining edges will be potential candidates to be used as reuse buffers. For simplicity, we use FIFO buffers that might even ve able to be implement with registers instead of using any additional hardware similar to the idea presented in [31] and [32]. Because we want to produce a memory system that has a size independent of the problem size, we only consider those edges which weight is not dependent on

the problem size. The weight or reuse distance is determined by the way the loops translate the stencil instance along the data domain. Once we have selected the edges that will be used as reuse buffers, we proceed to eliminate any nodes at then endpoint of any of those edges since they will reuse data from other nodes they do not have to be considered for the parallel banking scheme. The remaining nodes and all the edges between them will be used in the same as shown in previous sections to construct and color the Extended Stencil Graph or $ESG(S)$ of this simplified stencil. A detailed example of this process can be seen in figure 4.14.

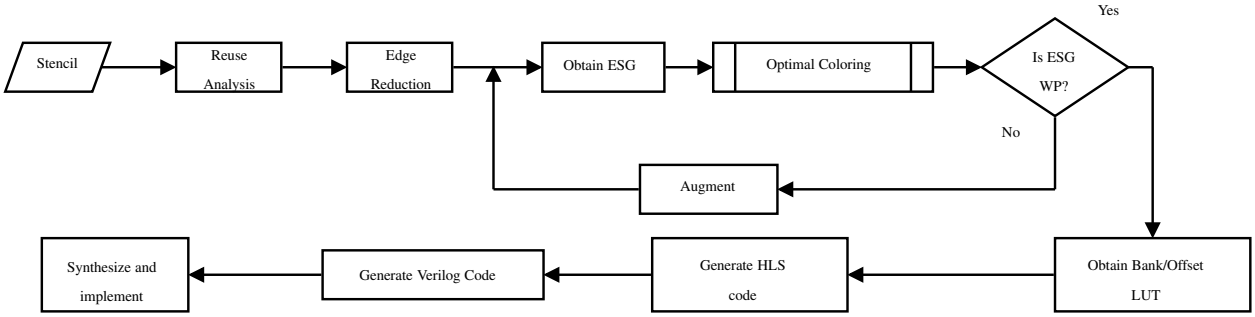


Figure 4.15: Flow diagram of our algorithm.

To illustrate the advantages of including data reuse and how it improves the partition factor we will consider the 12-point stencil seen in figure 4.16 where we can also see the final 4-bank solution obtained when incorporating data reuse. Two instances of the stencil, outlined in blue and purple are shown. Elements in dark gray are accessed every iterations while elements in dark gray are obtained from the reuse data scheme.

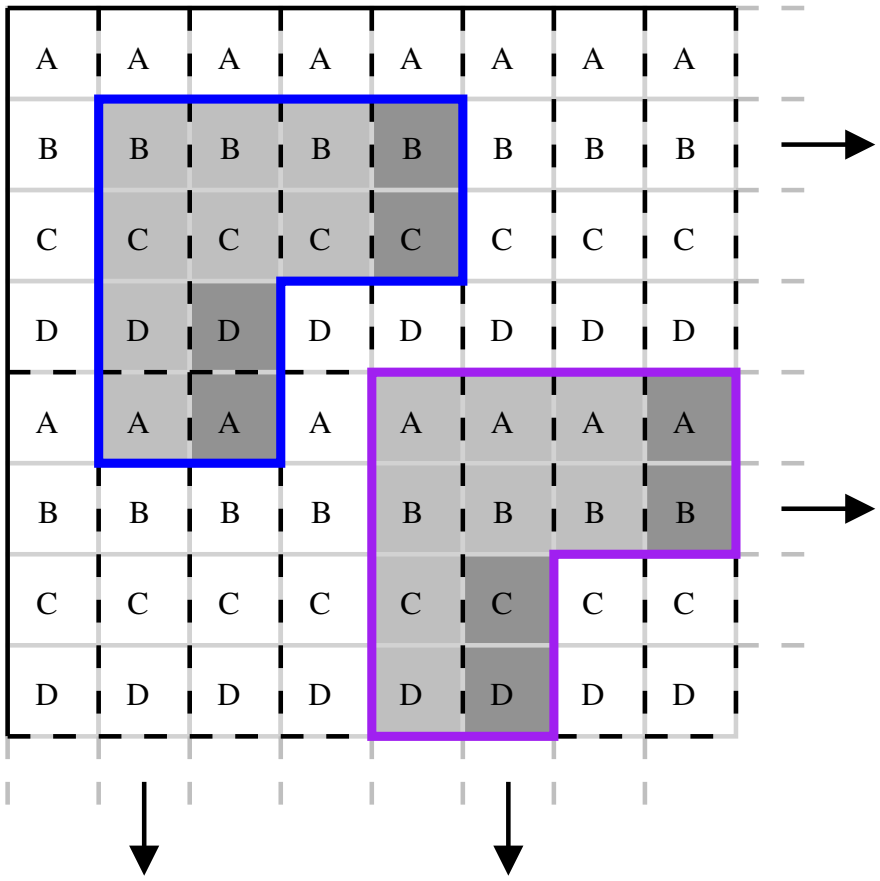
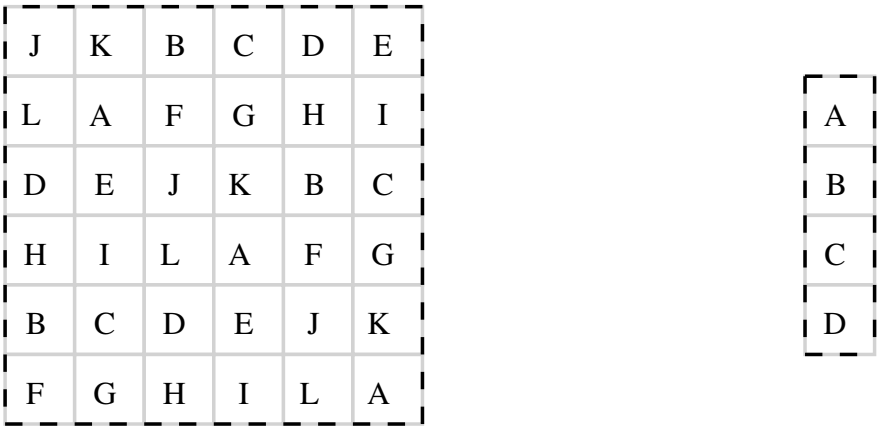


Figure 4.16: Coloring of the data space. Dashed black lines indicate where the pattern repeats. Gray areas are instances of the stencil. Light gray are memory accesses taken care of by the data reuse scheme. Dark gray need to be accessed in parallel each clock cycle

We can see that now we can achieve conflict-free parallel memory access with only 4 banks instead of the original 12. And not only that, but also the geometry of the repeating coloring has been reduced considerably from what originally was aa 6 x 6 square with 36 elements seen in Figure 4.17 (a), to a much smaller 1 x 4 rectangle as seen in Figure 4.17 (b). A reduction of 6x the number of elements. This will mean much smaller and simpler additional circuitry and possibly lower memory overhead.



(a) (b)

Figure 4.17: (a) Original repeating coloring for the 12-point stencil with 36 elements in a 6x6 square and 12 banks. (b) Repeating coloring for the 12-point stencil with data reuse using only 4 elements in a 1x4 rectangle and 4 banks

Modeling Multi-port memories

The method of obtaining the ESG of an stencil in order to obtain the optimal partition factor and a usable coloring scheme can be extended to use optimal defective coloring in order to model multi-port memories. This allows for a solution with a smaller partition factor to ensure parallel access, reducing the routing logic needed which in turn improves resource utilization and clock period given the smaller computational logic.

Defective coloring allows for up to d adjacent nodes to be assigned the same color and still consider it to be a valid coloring, where d is a parameter called defect factor of the coloring. It is easy to see how can one model multiport memories using defective coloring. At any given moment, only

up to d access to a certain memory bank are performed. If the memory bank has d ports then it can handle all requests without having to serialize the requests.

Using defective coloring to color for any given graph is equivalent to reducing the degree of each vertex by up to a factor of d. This is where the reduction of the chromatic happens. By removing edges, we are not only probably reducing the maximum degree of all nodes, but also possibly reducing the size of the maximum clique. Both of which influence the chromatic number.

Given the added complexity of implementing an optimal defective coloring scheme over a traditional optimal coloring algorithm and to maintain coherency between both approaches (optimal coloring and defective coloring), we instead do some preprocessing to the original stencil before proceeding as normal. This preprocessing consists in eliminating up to d edges from the original stencil graph, effectively reducing the size of the maximum clique (all stencils originally start as complete graph so removing an edge reduces the size of the maximum clique). It is in this step that we encode the defectiveness of the coloring. Once this is done we proceed to obtain the ESG of the modified stencil as usual. Unlike with the original ESG, now it is possible that not all nodes are connected to the pivot node. We use this to our advantage and consider the induced subgraph (ISG) of the ESG that contains only the nodes connected to the pivot node (this to maintain the requirement that the pivot node is connected to all other nodes, which in turn translates to the pivot node being the only node with a particular color). We proceed to optimally color this ISG. Because the nodes that weren't considered are not connected to the pivot node, and the pivot node is the only node with a particular color, it is safe to color said nodes the same color as the pivot. The analysis that follows is exactly the same as with the original stencil. If the chromatic number of the graph matches the maximum clique, then we have proven that it is possible to color the entire data domain using that coloring scheme.

In figure 4.18 we can see an example of this process. In (a) we consider an L shaped 3-element

stencil. (b) shows the modified stencil using a defect factor of 1. Eliminating up to d edges from all nodes in a random manner. (c) Shows the new ESG of the modified stencil constructed in the regular fashion. This modified ESG contains all the possible interleaving of the modified stencil, which has the defect factor encoded directly into it, with any other. (d) Shows the ISG considered where some nodes that are not connected to the picot node are not included in the coloring. Once this is done, (e) shows how we complete the coloring assigning the color of the pivot node to the originally excluded nodes. It is easy to see that the maximum clique of the ESG in (e) is 2, and that the chromatic number is also 2, so it meets the requirement for it to be used to cover the entire data space. (f) Shows how such coloring can be extended to cover the entire data space. At any point, selecting 3 nodes in the shape of the stencil will access any bank up to d times, ensuring parallel access.

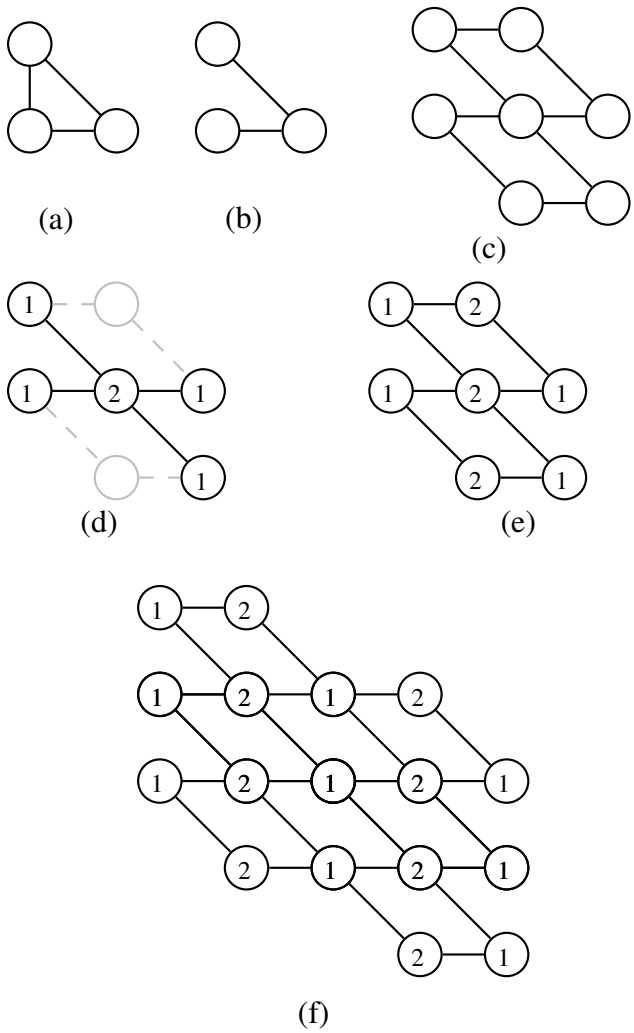


Figure 4.18: (a) Original 3-element stencil forming a complete graph. (b) Modified stencil removing up to d edges from each node, with d=1. (c) ESG of the modified stencil. (d) Colored induced subgraph of the ISG considering only the nodes connected to the picot point. (e) Colored full ESG. (f) Linkage of several ESG to cover the entire data domain with the defective coloring

To validate the performance benefits of our graph-based memory banking scheme, we start with inputting the memory access patterns of all test benchmarks into a Matlab script which computes the bank assignment and relative offset inside a super-tile for all memory locations. A Matlab script takes the information about the bank and offset super-tile and automatically generates new transformed code in C. This transformed C code is then used as an input to the Vivado HLS 2016.2 from Xilinx, which generates the HDL files in Verilog. The software also automatically generates a Vivado HLx 2016.2 project with the Verilog code already included. This project is synthesized and implemented. This software suite is also the same tool used to report post implementation resource usage and power estimation. To illustrate, we have listed a transformed code snippet in Fig. 4.19.

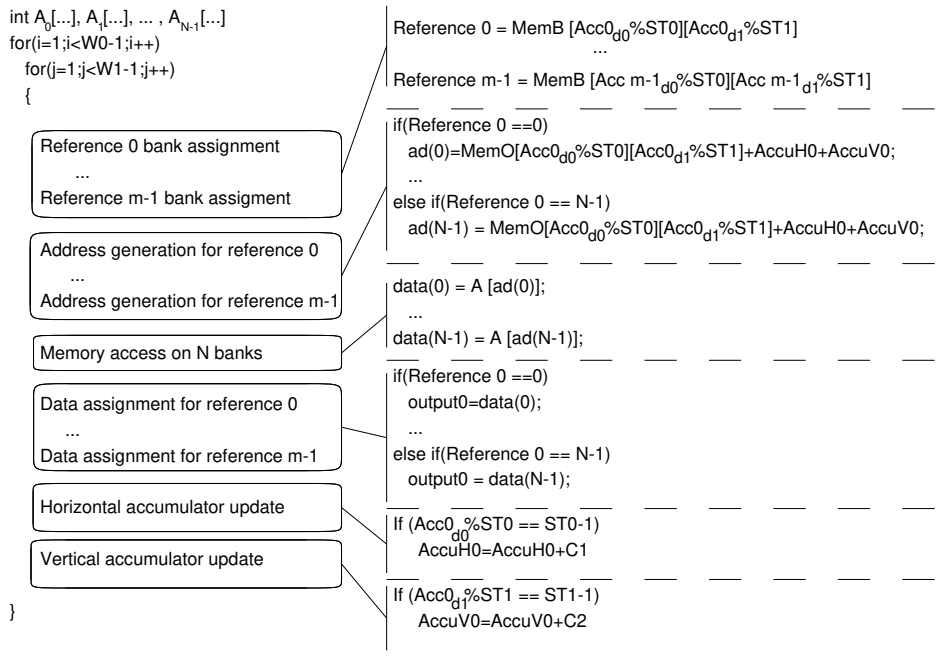


Figure 4.19: Template of transformed code. [1] DOI 10.1145/3174243.3174251

Six loop kernels with different access patterns are selected from a wide range of realistic applications, such as medical image processing and H.264 motion compensation. In our experiments, we mainly focus on the effects brought by different access patterns. The detailed experimental results are shown in Table 4.1 and Table 4.2. To compare, we also implemented the GMP method [3] for the same set of benchmarks and incorporated the results from [18] for the common ones (Bicubic, Deconv,Motion H, and Sobel). To do a fair comparison the original C code has the same structure for both sets of benchmarks, only calling different functions when testing GMP and our method. In all experimental runs, we turned on the loop pipelining setting in Vivado and set the target throughout with the iteration interval (II) to be 1, which requires all of the memory accesses in the same iteration to be in one clock cycle. For the hardware usage and energy consumption, we chose the target device to be the XC7K160tffg676-3 Kintex-7 FPGA for both Vivado HLS 2016.2 and Vivado Hlx 2016.2, and a bank size of 512 elements each in order to use one full RAMB18E1 block with a data width of 32 bits plus 4 bits of parity in single port mode. The results obtained can be seen in Table 4.1

Table 4.1: Resource utilization and clock period comparison. [1] DOI 10.1145/ 3174243. 3174251

		Bank #	CP(ns)	DSPs	FFs	LUTs	Pow.(mW)	Pipeline(#)
Denoise	GMP	4	2.1	0	254	438	754	8
	Ours	4	1.9	0	284	440	474	7
	Improv. (%)	0	9.52	0	-11.81	≈0	37.13	12.5
Bicubic	GMP	4	2.1	0	229	391	738	8
	Trace	4	3.66	0	212	184	N/R	N/R
	Ours	4	1.9	0	276	437	491	7
	Improv.(%)	0	9.52	0	-30.18	-137.5	33.45	12.5
Deconv	GMP	5	2.5	5	1320	1796	710	26
	Trace	5	3.37	10	383	541	N/R	N/R
	Ours	5	2	0	370	633	795	7
	Improv.(%)	0	20	100	3.39	-17	-11.97	73
MotionH	GMP	6	2.5	6	1783	2867	920	25
	Trace (Motion L)	6	3.31	6	392	425	N/R	N/R
	Ours	6	2	0	426	725	951	7
	Improv.(%)	0	20	100	-8.67	-70.59	-3.7	72
Sobel	GMP	9	2.5	9	3213	5792	1347	24
	Ours	9	2.2	0	606	1416	1340	7
	Improv.(%)	0	12	100	15.72	-33.71	≈0	70.8
12-Point	GMP	14	2.8	12	5108	9116	1687	34
	Ours	12	2.4	0	806	2159	1895	7
	Improv. (%)	14.3	14.3	100	84.22	76.3	-12.3	79
Average(%)		2.38	14.21	66.67	-8.77	-30.42	7.1	53.3

Due to the regularity of the repeating pattern, the problem of computing intra bank offsets becomes an extension of the above memory bank mapping problem where we have a rectangle with a repeating pattern, this time intra bank offsets. First, for the upper d-1 dimensions, we want to calculate number of elements belonging to a particular bank that are in a d-2 dimensional space. This is, for 3-D matrix, we want to calculate how many elements are of each bank first in a cube with base vectors $[a_2, 0, 0], [0, w_1, 0], [0, 0, w_0]$, then in a rectangle with base vectors $[0, 0, 0], [0, a_1, 0], [0, 0, w_0]$, and finally in one of the repeating regions. The maximum of these values for each dimension are then stored in memory. Once this is done, the intra bank offset becomes a function of the intra region offset and, 2 accumulators per access for the 2D case. One to store the number of elements before it in the same row of super-tiles, and another with the number of elements in the previous rows of super-tile. $\text{Offset}_{\text{Acc}_k} = \text{Mem}_O[X_0 \mod a_0, \ldots, X_{d-1} \mod a_{d-1}] + \text{Acc}_\text{H} + \text{Acc}_\text{V}$ Similarly as with the bank access, one can use accumulators and counters instead of costly modulo operation to access all the dimensions of the rectangle every iteration.

It is worth noting that the internal offset of the rectangle stored in Mem_O can be carefully arranged to reduce memory waste. By keeping the lower offsets in the area of the region that will always be within the bounds of the matrix under consideration, and assigning the higher ones to the areas that represent the smaller area for the other dimensions, we can effectively reduce the total memory waste. The highest offsets being located in the region that is less frequently accessed.

In figure 4.20 we see the size of the matrix in one dimension is an integer multiple of the size of the super-tile in dimension d_0, then we will only incur in wasted memory in the last iteration of the complementary loop. With this in mind, we can keep the lowest offsets in the area of the super-tile that are always going to be within the bounds, thus in the last iteration, such indexes will not be accessed and we have reduced memory waste to 0. The order of the offsets in each of the zones can be arbitrary and does not affect memory waste in any form.

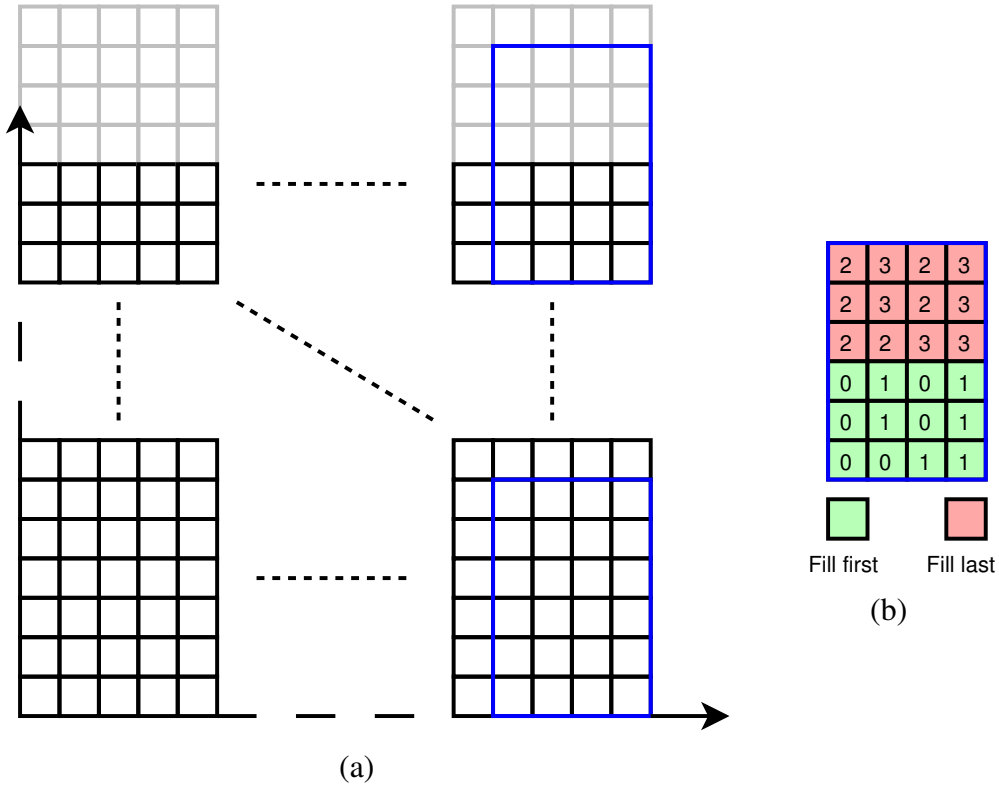


Figure 4.20: (a)Arrangement of the repeating intra bank offset rectangle of for an m x n matrix using a 4 x 6 rectangle. (b)Mem_O offsets. [1] DOI 10.1145/ 3174243.3174251

The final amount of memory overhead can be calculated by:

$$\text{Overhead} = \sum_{i=0}^{d-1}((\lceil \frac{w_i}{STi} \rceil - w_i) \times \prod_{k!=i} w_k) \tag{4.3}$$

The main results, obtained from synthesizing both mapping functions and their respective address calculation logic ([3] and ours) and comparing it with the reported results from [18], can be seen in table 4.1. Here we see that our method can not only achieve better partition factor for

certain stencils, as seen in the 12-point stencil, and no worse in general but also achieve reduced resource utilization and clock period. The reduction of the clock period across the board comes from the capabilities of the FPGA fabric to synthesize a small distributed multidimensional array with fine grain access to all its elements with low hardware usage. The elimination of many of the division and multiplication operations from [3] and [18], using instead a quick access to the aforementioned memory, and only a few modulo operations, some multiplication, and additions (which are also needed in [3] and [18]) account for the 100% reduction in DSP usage and decrease hardware recourse utilization in general. For small stencils, such as Denoise, we see that the implementing the mapping and address calculation function from [3] is actually more efficient in terms of hardware utilization, but our method scales much better as the stencil size grows. But compared with [18] we have an increase in hardware utilization for both LUT's and FF.

Table 4.2: Memory waste comparison. [1] DOI 10.1145/3174243.3174251

		Memory overhead (elements)				
		SD	HD	FHD	WQXGA	4K
		640x480	1280x720	1920x1080	2560x1600	3840x2160
Denoise	GMP	0	0	0	0	0
	Ours	0	0	0	0	0
	Improv.	0%	0%	0%	0%	0%
Bicubic	GMP	0	0	0	0	0
	Ours	0	0	0	0	0
	Improv.	0%	0%	0%	0%	0%
Deconv	GMP	0	0	0	0	0
	Ours	0	0	0	0	0
	Improv.	0%	0%	0%	0%	0%
MotionH	GMP	960	2880	0	3200	0
	Ours	960	2880	0	3200	0
	Improv.	0%	0%	0%	0%	0%
Sobel	GMP	3840	5040	6480	8000	6480
	Ours	960	720	0	8324	0
	Improv.	75%	85.7%	100%	-4%	100%
12-Point	GMP	1920	5759	12960	3180	21591
	Ours	960	0	0	8324	0
	Improv.	50%	100%	100%	-2261.7%	100%
Average		20.8%	30.95%	33.33%	-34%	33.33%

Another important result is the comparison between the memory overhead resulting from the padding method in [3] and our address calculation method. In both cases, memory overhead is required to generate a regular and simple intra-bank address calculation function. The improved memory overhead in comparison to [3] comes from the a ability to generate arbitrary offsets inside the repeating region. This in turn allows for a better control of how much space is wasted each row for different problem sizes while the method in [3] always needs to complete a sequence of N x B elements where N is the partition factor obtained by their method and B is the block size. There are

some cases however that the k-dimensionality of our mapping function generates memory overhead in all dimensions while the method in [3] being one dimensional always generates overhead in just one. In this cases, the padding in [3] can be lower than ours, but this is not generally the case.

CHAPTER 5: GRAPH BASED APPROACH FOR MEMORY BANKING IN NON-STENCILS

This chapter is based on our previously published work in Juan Escobedo and Mingjie Lin, xtracting Data Parallelism in Non-Stencil Kernel Computing by Optimally Coloring Folded Memory Conflict Graph, Proceedings of the 55th Annual Design Automation Conference DOI 10.1145/3195970.3196088 [2] and encompasses our graph based approach to find a reduced conflict memory banking solution for non-stencil kernels.

We will begin this chapter with a motivational example that will be used throughout this chapter to demonstrate concepts and exemplify the contribution of this graph-based approach for non-stencil code.

Motivational example

Take the modified forward Gauss-Jordan Elimination (GE) kernel from figure 5.1 (a). As we can see from figure 5.1 (b), (c), and(d), the relative distance of the memory elements to be accessed changes with time. This in contrast to the stencil-type kernels like Sobel or Denoise where all the relative distances remain constant. Although methods such as the one from [3] could in theory tackle this kind of problems, we know from [1] that as the problem size grows, the number of required banks for parallel memory access increases as well. For arbitrarily large problems, this yields an unfeasible number of partitions. The question is how can we obtain a mapping with a realistic partition factor that gives the best results in terms of conflict reduction

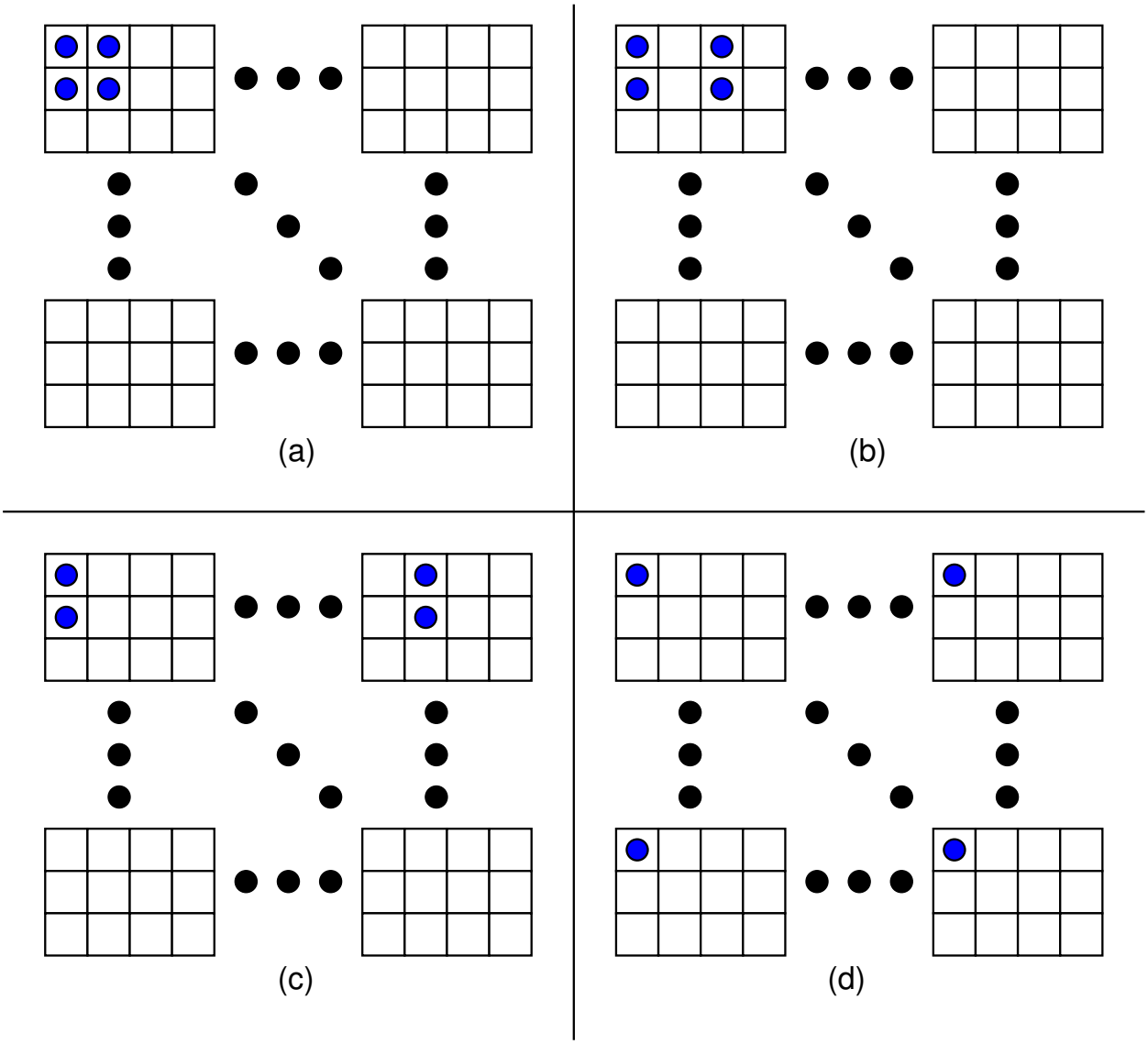


Figure 5.1: (a) Modified forward Gauss-Jordan elimination kernel. (b) Accessed elements on iteration (i,j,k) = (0,1,1). (c) Accessed elements on iteration (i,j,k)=(0,1,2). (d) Accessed elements on iteration (i,j,k) = (0,2,2). [2] DOI 10.1145/ 3195970.3196088

The idea is then to take a subproblem of size n, equivalent to the supertile from [17], and map the whole graph to a region of the desired size. Figure 5.2 (a) shows 3 iterations of the GE kernel,

each in a different color. Now, considering a subproblem of size 2 x 2. All we have to do is take the coordinates of each node in the grid modded by this factors, and carry the edges to this new domain. This is, all nodes with coordinates in the form (a·2,b·2) for $a, b \in \mathbb{Z}$ will be mapped to point (0,0), points in the form (a·n,b·n+1) to point (0,1) and so on and so forth. Any edges are carried to the new map, including the cases when two nodes are mapped to the same one. In this case we add a self edge to the respective node. These self edges will represent conflicts that cannot be resolved by adding more banks but only by changing the geometry of the subproblem size. This procedure is equivalent to overlapping all the subregions, as seen in figure 5.2 (b). Doing this process for the 3 iterations of figure 5.2 (a) yields the conflict weighted graph of figure 5.2 (c).

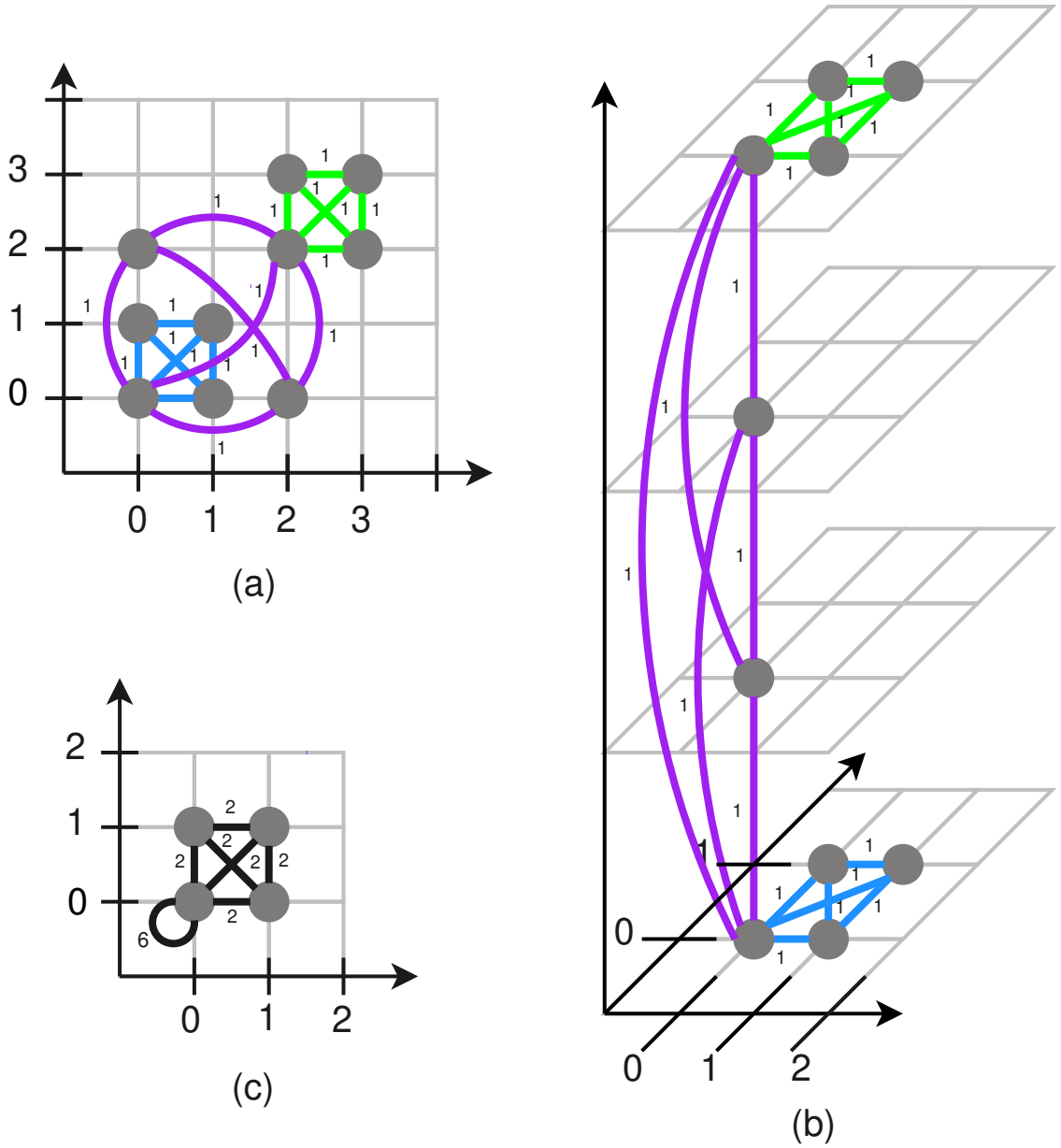


Figure 5.2: (a) Conflict graph for iterations (i,j,k)=(0,1,1) in blue, (i,j,k)=(0,2,2) in purple, and (i,j,k)=(2,3,3) in green. (b) Overlapping of 2x2 regions of the 4x4 problem. (c) Final weighted conflict graph for the iterations under consideration. [2] DOI 10.1145/3195970.3196088

Once the problem has been mapped to the new subproblem, we color each node pairwise based on the weight of the edge in descending order, trying to assign distinct colors to the nodes with the heaviest edges first (which represents the highest amount of conflicts). Depending on the maximum number of desired banks, we will be able to remove some of the conflicts. For the same 4x4 problem, one would need 7 colors to have true conflict free accesses. Using 2, 3, and 4 colors on the 2x2 mapped problem, one would get 32, 26, and 20 conflicts respectively. This corresponds to a reduction of 33.33%, 45.83%, and 58.3% of the total conflicts.

Considering the original 4x4 problem of figure 5.1, considering a supertile of size 2x2, we get the conflict graph from figure 5.3

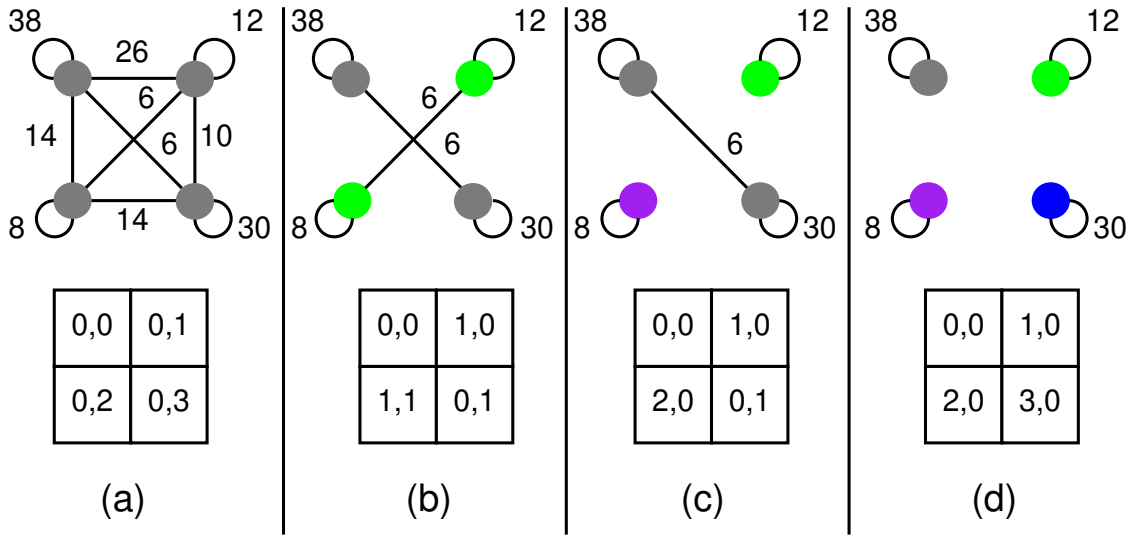


Figure 5.3: Coloring of the resulting conflict graph (top) and corresponding intra-supertile [bank, address] pair (bottom) for 1 bank (a), 2 banks (b), 3 banks (c), and 4 banks (d). [2] DOI 10.1145/3195970.3196088

Repeating this for a problem of size 256x256 (for which a greedy graph coloring algorithm as the one presented in [33] gives a required number of colors of 259 for conflict free access), still with

a supertile of 2x2 and again using 2,3, and 4 colors, we get 191520, 169664, and 147808 conflicts respectively. This corresponds to a reduction of 25.1%, 33.46%, and 41.83% of the total number of conflicts as seen in figure 5.4

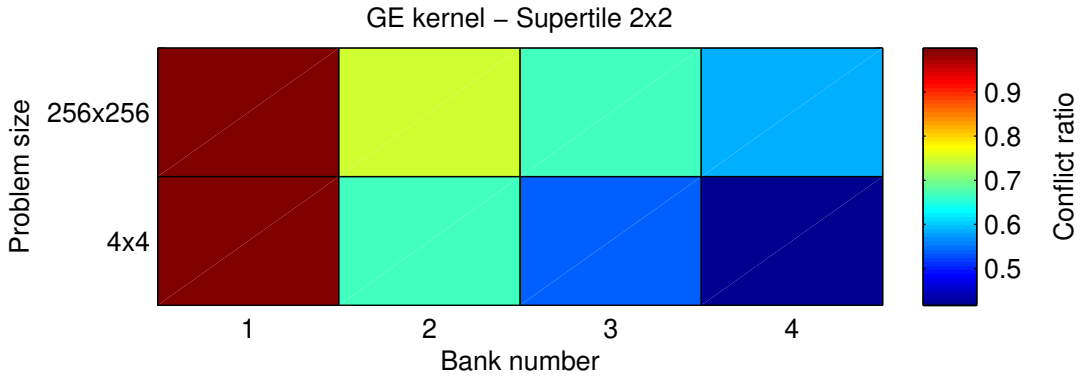


Figure 5.4: Conflict ratio for a supertile of size 2x2 and a problem of size of dimensions 256x256 (top) and 4x4 (bottom) for 1,2,3, and 4 banks. [2] DOI 10.1145/ 3195970.3196088

Problem Definition

Definition 6 (Iteration Domain) *Given an l-level loop nest, the iteration domain* $\mathbf{I}$ *is formed by all the iteration vectors* $\vec{i} = (i_0, i_1, \ldots, i_{l-1})^T$ *within the loop bounds.*

Definition 7 (Data Domain) *Given an l-level loop nest, the data domain* $\mathbf{D}$ *is formed by all the vectors* $\vec{x} = (x_0, x_1, \ldots, x_d)^T$ *within the matrix bounds.*

Definition 8 (Affine Memory Reference) *We say that a memory reference in a loop is affine if* $\mathbf{I}$ *and* $\mathbf{D}$ *are affine spaces.* $\mathbf{I}$ *and* $\mathbf{D}$ *are affine if one can define any d-dimensional affine memory access* $\vec{x} = (x_0, x_1, \ldots, x_{d-1})^T$ *as a linear transformation of a one and only one l-dimensional iteration vector* $\vec{i}$ *in the form of:*

$$\vec{x} = A_{d\times l} \cdot \vec{i} + \vec{C}$$

$$A_{d\times l} = \begin{bmatrix} a_{0,0} & \dots & a_{0,l-1} \\ \vdots & \ddots & \vdots \\ a_{d-1,0} & \dots & a_{d-1,l-1} \end{bmatrix}, \quad \vec{C} = \begin{bmatrix} a_{0,l} \\ \vdots \\ a_{d-1,l} \end{bmatrix}$$

Where $A_{d\times l}$ is a coefficient matrix, $a_{k,j} \in \mathbb{Z}$ is the coefficient of the j-th iteration vector coordinate on the k-th dimension, and $\vec{C}$ is a a column vector with constants.

Definition 9 (Memory Access Pattern) *A pattern consists of m data points or accesses defined as* $\mathbf{P} = \{\vec{A_0}, \vec{A_1}, \dots, \vec{A_{m-1}}\}$*, where* $\vec{A_j} = A_j \cdot \vec{i} + \vec{C_j}, \quad \vec{i} \in \mathbf{I}$.

Definition 10 (Memory partitioning) *A memory partition of an n-dimensional array can be described as a pair of mapping functions* $(f(\vec{x}), g(\vec{x}))$ *where* $f(\vec{x})$ *assigns a bank for the data element and* $g(\vec{x})$ *generates the corresponding intra-bank offset.*

A bank access conflict between to references $\vec{x}_j$ *and* $\vec{x}_k$ *is represented as* $\exists\, \vec{x} \in \mathbf{D}$ *s.t.*

$$f(\vec{x}_j) = f(\vec{x}_k),\ \vec{x}_j \neq \vec{x}_k$$

This means the references intend to access the same bank in the same clock cycle. We use Problem 1 to formulate the bank mapping problem (for single-port memories).

Our memory partitioning consists of two mapping problems: bank mapping and intra-bank offset mapping.

Problem 5 (Conflict minimization) *Given an l-level loop on the iteration domain* $\mathbf{I}$*, an access pattern* $\mathbf{P}$ *on the data domain* $\mathbf{D}$*, and a partition factor N, find a color mapping strategy f for the conflict graph such that:*

$$Minimize : -\sum_{i=1}^{|I|} | f(\mathbf{P}_i) | \tag{5.1}$$

Eqn. 5.1 defines the objective function of conflict minimization: minimizing the number of times elements in $\mathbf{P}_i$ are assigned to the same bank. After bank mapping, a data element in the original array should be allocated a new intra-bank location.

For correctness, two different array elements will be either mapped onto different banks or the same bank with different intra-bank offsets. An intra-bank offset function is valid if and only if:

$$\forall \vec{x}_j, \vec{x}_k \in \mathbf{D}, \vec{x}_j \neq \vec{x}_k \rightarrow (f(\vec{x}_j), g(\vec{x}_j)) \neq (f(\vec{x}_k), g(\vec{x}_k))$$

Which means either

$$f(\vec{x}_j) \neq f(\vec{x}_k) \text{ or } f(\vec{x}_j) = f(\vec{x}_k), g(\vec{x}_j) \neq g(\vec{x}_k)$$

Problem 6 (Storage minimization) *Given an l-level loop on the iteration domain $\mathbf{I}$, an access pattern $\mathbf{P}$ on the data domain $\mathbf{D}$, and a partition factor N, find an intra-bank offset mapping function g with minimum storage requirement S such that:*

$$Minimize : S = \sum_{j=0}^{N-1} max\ g(\vec{x}_i) \tag{5.2}$$

$$s.t.\ \forall\ i\ s.t.\ f(\vec{x}_i) = j\ and\ \forall \vec{x}_j, \vec{x}_k \in \mathbf{D}, \vec{x}_j \neq \vec{x}_k \rightarrow (f(\vec{x}_j), g(\vec{x}_j)) \neq (f(\vec{x}_k), g(\vec{x}_k))$$

Eqn. 5.2 defines the objective function of partitioning with minimum storage overhead, ensuring a valid partition.

Definition 11 (Conflict Graph) *A conflict graph is generated by considering all the points in the data domain* $\mathbf{D}$ *as nodes and adding edges between the data points accessed by the elements of all patterns* $\mathbf{P}_i$, $\forall\, i \in \mathbf{I}$

Definition 12 (Proper graph coloring) *Graph coloring of a graph* $\mathcal{G}$*=(*$\mathbf{V}$*,*$\mathbf{E}$*) is the process where the vertices v* $\in\mathbf{V}$*, are assigned distinct colors such that no two adjacent vertices, those connected by an edge e* $\in\mathbf{E}$*, have the same color.*

If a graph $\mathcal{G}$ *can be colored with k colors, it is said the graph is k-colorable. Determining if a graph* $\mathcal{G}$ *is k-colorable for k*$\geq$*3 has been proven to be an NP-complete problem [27].*

Definition 13 (Chromatic number) *The chromatic number of a graph, denoted* $\chi(\mathcal{G})$*, is the smallest k for which the graph is k-colorable.*

Definition 14 (Soft coloring) *Soft coloring [34] is a simplification of the proper coloring problem. Soft coloring coloring of a graph*$\mathcal{G}$*=(*$\mathbf{V}$*,*$\mathbf{E}$*) with* $\chi(\mathcal{G})$*=k and non-negative edge weights, is the coloring of said graph using m colors such that the number edges connecting same-color nodes is minimized. Note that m can be less, equal, or grater to k.*

Definition 15 (Conflict ratio) *The conflict ratio* γ *for a given conflict graph with particular coloring scheme is the ratio of the conflicts for said mapping scheme, which can be calculated as the sum of the cardinality of each memory access pattern set* $\mathbf{P}_i$ *minus the of the cardinality of the set* $\mathbf{C}_i$ *resulting from applying the mapping function f to the set* $\mathbf{P}_i$*, and the maximum amount of conflicts, which is the sum of the cardinality of each* $\mathbf{P}_i$ *minus 1 for all for all i* $\in \mathbf{I}$*. This is:*

$$\gamma = \frac{\sum_{i=1}^{|I|} |\mathbf{P}_i| - |\, f(\mathbf{P}_i)\,|}{\sum_{i=1}^{|I|} (|\mathbf{P}_i| - 1)} \tag{5.3}$$

Conflict minimization

The idea of fully parallel access for non-stencil applications was explored in [2]. The main issue was that as the problem size grows, the number of banks needed to ensure conflict free access grows at a rate that eclipses any benefits gained from the parallel access given the complexity of the interconnect network.

To aggravate the situation, storing the whole mapping function becomes impractical for arbitrarily large problems. Theoretically needing all or even more than the on chip storage/resources to do the mapping.

Dividing the solution and only loading the necessary sections of the mapping presents complications of its own. Partial reconfiguration in its current state is not viable due to very long reconfiguration times. Making the area of the solution used for each partial reconfiguration, reducing the number of total reconfigurations, does not solve the problem because the number of configuration bits themselves increase. Adding some overhead to the data brought on chip only presents benefits if the total number of bits needed to represent the bank and the address of each element is less than k times the payload size which cannot be ensured for arbitrarily large problems, with k being the number of access needed per iteration.

We propose to instead try to produce a fully conflict free solution, minimize the total number of conflicts for a given maximum partition number and problem area.

Conflict graph generation

The first step in the algorithm consists in generation a conflict graph, initially considering all valid data points in the data domain as nodes in an empty graph. For simplicity one can simply flatten the array and label the nodes in sequential order. Once this is done, all the elements of each $\mathbf{P}_i$ are connected via an edge. This is done for all i$\in \mathbf{I}$. The resulting conflict graph, as the one seen in figure 5.5, encodes all the information necessary to achieve conflict-free parallel access, or as presented in later sections, methods to arbitrarily reduce the total amount of conflicts.

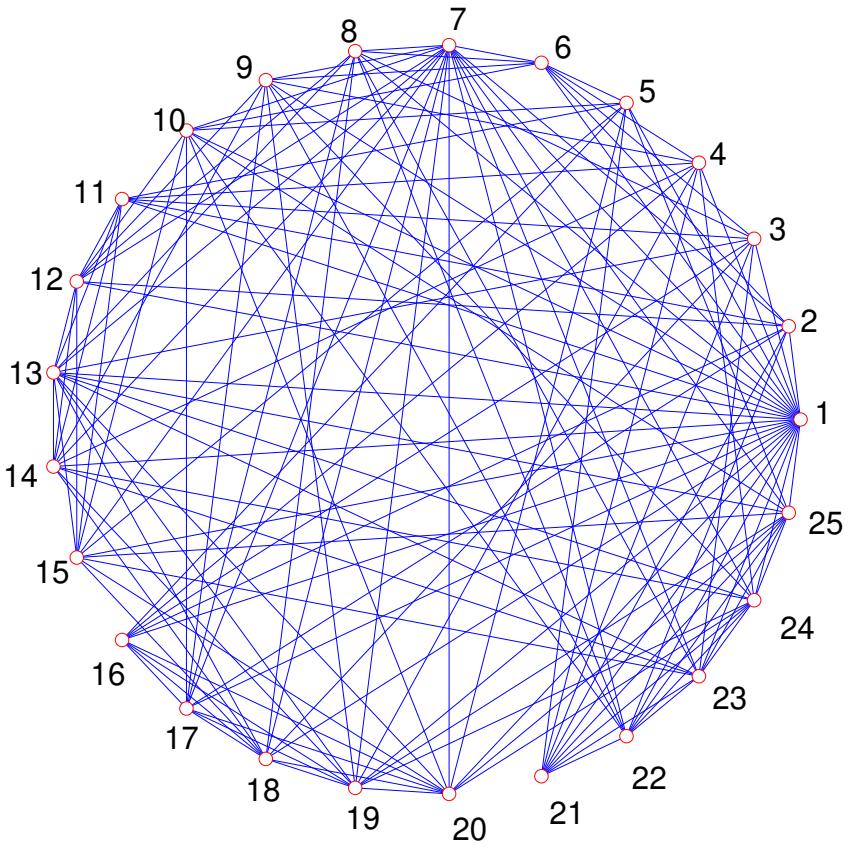


Figure 5.5: Full conflict graph for the GE kernel on a 5x5 matrix. [2] DOI 10.1145/3195970.3196088

The simple form of the mapping function in methods such as [3] and [14], that could be in theory used to do a conflict free mapping for any kernel, means that not all the edges of the conflict graph can be considered (when arranging the nodes on a grid, not all edges might follow the function specified by the mapping function), for a practical partition factor at least. Even with a large partition factor, a single or even a couple families of hyperplanes ([16], [17]) might be insufficient to consider all conflicts .

In theory one could increase the complexity of the mapping function, adding exception and modifying the from, until all edges in the conflict graph are considered. This is the main idea of [2] and the Arbitrary Mapping Function presented in it. We take this idea and expand on it, circumventing the aforementioned problem of needing to store, in the worst case scenario, a mapping solution as large as the problem itself applying the technique from the following section.

Graph overlapping

The way we propose to achieve this conflict minimization is by using an extension of the supertile concept from [17]. In the aforementioned work, the supertile was the smallest rectangle for which a mapping pattern that ensured conflict-free parallel access for a given stencil kernel repeated itself. This idea can be extended to encode the most common combinations of nodes from an arbitrarily large problem by overlapping regions of size n $\times$ m of the original problem grid where each node has integer coordinates over one another, keeping all the edges and then projecting everything over a single n $\times$ m region, adding the weight of the edges in the same relative position. The size and geometry of the region (rectangle, square, etc) can be arbitrary, but ideally it is such that self conflicts are minimized.

Applying this procedure to the graph in figure 5.5 on a 3x3 supertile gives us the graph from figure 5.6

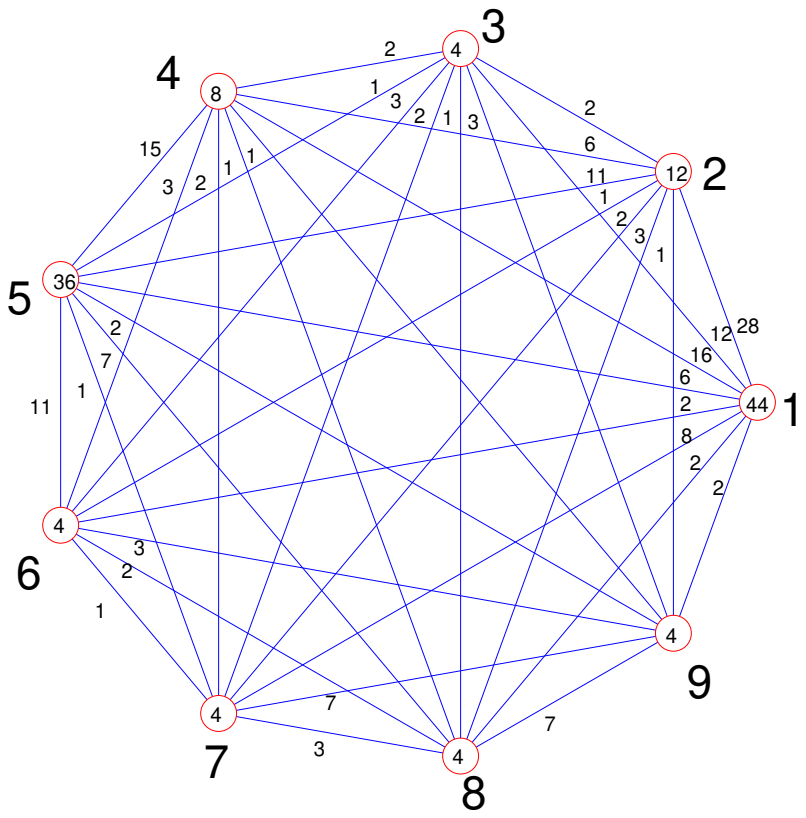


Figure 5.6: Conflict graph of a 3x3 supertile for the GE kernel on a 5x5 matrix. [2] DOI 10.1145/3195970.3196088

Because in the worst case, the resulting graph can become complete, traditional coloring algorithms that ensure conflict free access are not adequate for our goal. First, they might need up to n $\times$ m banks to do the mapping, exceeding the practical maximum even for relatively small work areas. Second, given the nature of the resulting graph, it is not only possible but expected that there are self-conflicts, this is, some loops will have edges to themselves. This kind of conflicts are unavoidable for the proposed methodology and the only way to reduce them is by changing the

geometry of the supertile. After considering all this, we propose the following priority coloring algorithm to reduce the total number fo conflicts.

Priority coloring

We propose the algorithm seen in algorithm 5.7 to do the mapping. For every edge, from heaviest (most repeated pair of nodes accessed together) to lightest, we try to assign distinct colors to the nodes connected by the edge. If both nodes have been mapped, then the edge cannot be removed and it will have an impact in the final total conflict count. In all other cases (both nodes not mapped, or just one node mapped) it is always possible to assign distinct nodes for a partition factor greater than 1. Ideally, one would like to also consider all colors used by the nodes connected to the one under consideration to avoid even more conflicts. Using the least used color in the subset of unused colors by the adjacent nodes helps even out the distribution of colors. Since the algorithm is best-effort based, the number of colors, and thus, the partition factor, is arbitrary. Section 5 illustrates how different partition factors affect the total number of conflicts.

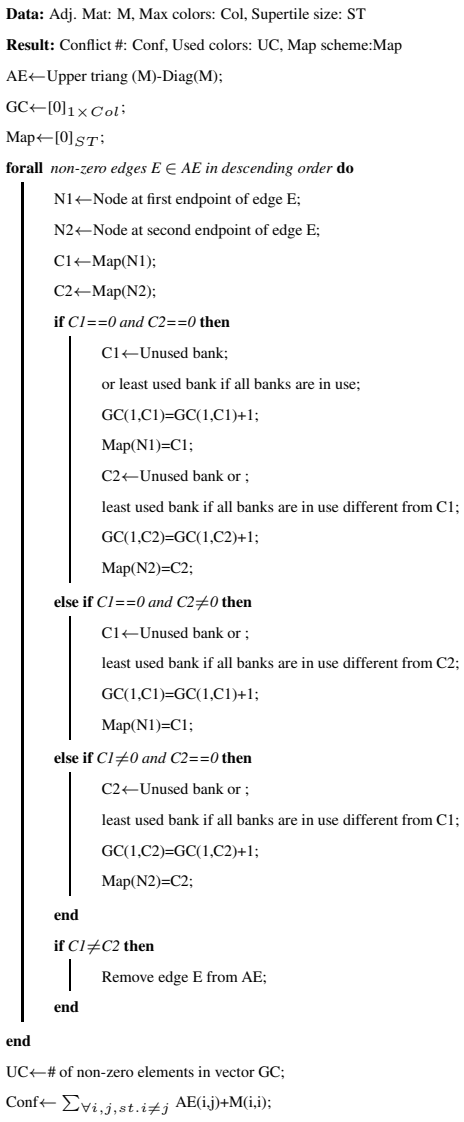


Figure 5.7: Edge removal algorithm. [2] DOI 10.1145/3195970.3196088

Bank mapping

Once the nodes of the supertile are colored, the mapping can be taken directly from it as can be seen in section 5, figure 5.3. To obtain the bank, one simply would need to take the coordinates of

the memory access, mod them by the corresponding dimension of the supertile (called bank super tile or BST), and access it. This is, for a k-dimensional problem:

$$\begin{aligned}\text{Bank (B)} = \text{BST}(&x_0 \bmod \text{ST}_0,\ x_1 \bmod \text{ST}_1,\ \ldots \\ &,\ x_k \bmod \text{ST}_k)\end{aligned} \tag{5.4}$$

Address calculation

The address, or intra bank offset, calculation is very straight forward. Similarly to the work in [17], the address calculation is made of two portions: intra-supertile offset and global offset.

The intra-supertile offset is calculated at compile time by scanning the supertile and assigning all the elements of the same bank a distinct number. To access the intra-supertile offset, one accesses the address supertile (AST) in the same fashion one would d access the bank super tile. this is:

$$\begin{aligned}\text{Intra-supertile offset (STO)} = \text{AST}(&x_0 \bmod \text{ST}_0, \\ &x_1 \bmod \text{ST}_1,\ \ldots,\ x_k \bmod \text{ST}_k)\end{aligned} \tag{5.5}$$

An example of this can be seen in section 5, figure 5.3

To calculate the global offset, given a memory access point in k-dimensional space, one just has to count the number of supertiles in the lower dimensions and multiply it by the maximum offset of the bank per supertile. To simplify calculations, one can consider the same maximum offset M for all banks as the greatest of all offsets in the supertile. Note this might lead to some memory waste

because some colors might have been used more often than other, but since the coloring algorithm tries to balance color utilization, it is expected that the intra-supertile offsets for all the banks to be roughly the same, the waste is expected to remain minimal.

This is, a memory access with coordinates $(x_0, x_1, \ldots, x_k)$ for a problem size $w_0 \times w_1 \times \cdots \times w_k$, and a supertile of size $\text{ST}_0 \times \text{ST}_1 \times \cdots \times \text{ST}_k$, with a maximum intra-supertile offset of M, the global offset can be calculated as:

$$\text{Global offser (GO)} = \text{M} * \sum_{i=0}^{k-1} (\lfloor \frac{x_i}{ST_i} \rfloor * \prod_{j=0}^{i-1} \lceil \frac{w_j}{ST_j} \rceil) \tag{5.6}$$

Note for any given problem, equation 5.6 can be expanded at compile time given the dimensionality of the problem does not change, thus being able to pre-compute some of the terms. Particularly all the products in $\prod$.

Finally, the definitive address is the sum of the intra-supertile offset (STO) and the global offset (GO):

$$\text{Address} = STO + GO \tag{5.7}$$

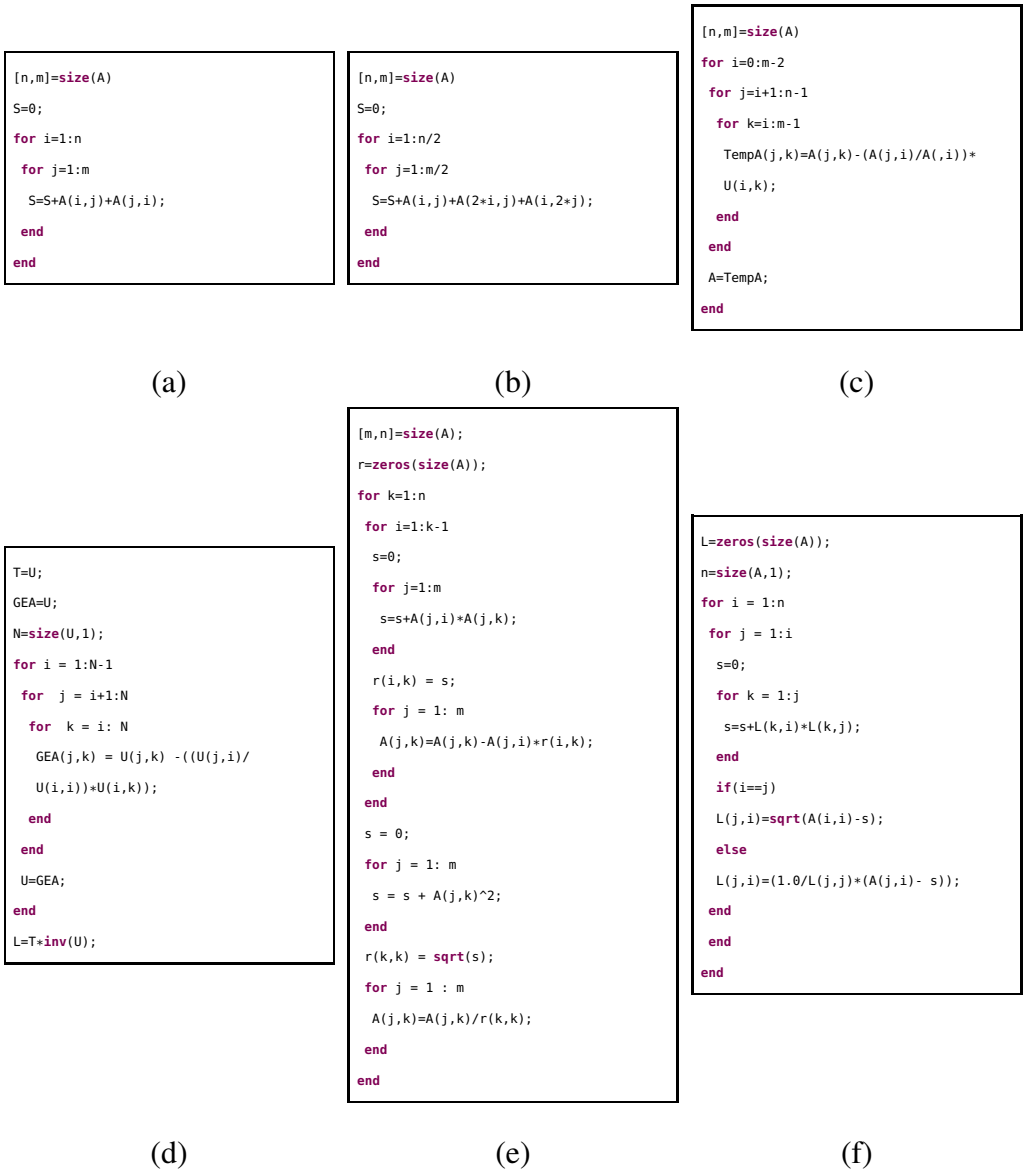
Methodology

We used Matlab 2014a to obtain the problems full conflict graphs for the code kernels seen in figure 5.8, perform the overlapping, and color the resulting graphs using algorithm 5.7 storing the resulting mapping function to later be used in a piece of code to be synthesized.

The kernels include commonly used pieces of code in linear algebra such as the LU factorization, Gauss-Jordan elimination (forward), QR factorization and Cholesky decomposition. In addition to 2 custom kernels, Double row/Double column and Row/Column to test the generality of the method.

To determine the tile sizes to be used we generated a set of two figures per kernel. One with the ratio between self-conflicts for each tile size of up to 10×10, and another with the partition factor needed to eliminate all inter node conflicts for a given tile size, again up to 10×10. All sets were generated on a 10×10 problem. A generic greedy coloring algorithm was used to obtain this coloring.

Based on the results obtained, we selected 3 different supertile sizes of 8x8, 16x16, and 32x32, a maximum partition factor of 32 (available banks) to study the effects on conflict reduction for a given tile size if more banks are used. We limited the problem sizes up to 128x128 for each kernel due to computational power restrictions. Note the tile sizes are all powers of 2, which although might not yield the optimal partition number, makes the multiplexer implementation much more efficient.



```
[n,m]=size(A)
S=0;
for i=1:n
 for j=1:m
  S=S+A(i,j)+A(j,i);
 end
end
```

(a)

```
[n,m]=size(A)
S=0;
for i=1:n/2
 for j=1:m/2
  S=S+A(i,j)+A(2*i,j)+A(i,2*j);
 end
end
```

(b)

```
[n,m]=size(A)
for i=0:m-2
 for j=i+1:n-1
  for k=i:m-1
   TempA(j,k)=A(j,k)-(A(j,i)/A(,i))*
   U(i,k);
  end
 end
 A=TempA;
end
```

(c)

```
T=U;
GEA=U;
N=size(U,1);
for i = 1:N-1
 for  j = i+1:N
  for  k = i: N
   GEA(j,k) = U(j,k) -((U(j,i)/
   U(i,i))*U(i,k));
  end
 end
 U=GEA;
end
L=T*inv(U);
```

(d)

```
[m,n]=size(A);
r=zeros(size(A));
for k=1:n
 for i=1:k-1
  s=0;
  for j=1:m
   s=s+A(j,i)*A(j,k);
  end
  r(i,k) = s;
  for j = 1: m
   A(j,k)=A(j,k)-A(j,i)*r(i,k);
  end
 end
 s = 0;
 for j = 1: m
  s = s + A(j,k)^2;
 end
 r(k,k) = sqrt(s);
 for j = 1 : m
  A(j,k)=A(j,k)/r(k,k);
 end
end
```

(e)

```
L=zeros(size(A));
n=size(A,1);
for i = 1:n
 for j = 1:i
  s=0;
  for k = 1:j
   s=s+L(k,i)*L(k,j);
  end
  if(i==j)
  L(j,i)=sqrt(A(i,i)-s);
  else
  L(j,i)=(1.0/L(j,j)*(A(j,i)- s));
 end
 end
end
```

(f)

Figure 5.8: Matlab code for: (a) Row/Column (RC), (b) Double Row/ Double (DRDC), (c) Combined Gauss-Jordan forward elimination (GE), (d) LU factorization (LU), (e) Cholesky decomposition (CHO), (f) QR decomposition (QR). [2] DOI 10.1145/3195970.3196088

We synthesized the mapping functions for the aforementioned supertile sizes of the GE kernel (4 memory accesses) using a partition factor of 16. The Verilog code was generated by implementing the function seen in figure 5.9 in Vivado HLS 2015.4 and synthesized Vivado Hlx 2015.4 on a Kintex 7 xc7k160tffg676-3 chip. The function takes the 4, 2-D coordinates in $\mathbf{D}$ of the references as inputs and outputs the corresponding bank and address of the elements.

```
#include <stdio.h>
#include <math.h>

#define Mat0 128 / /Size of the problem
#define ST 8 / /Size of the supertile
#define HC Mat0/ST / /Total number of supertiles per row of the problem

void MapFun(unsigned int *X1,unsigned int *Y1,unsigned int *X2,unsigned int *Y2, unsigned int *X3,unsigned int *Y3,unsigned int *X4,unsigned int *
    ↪ Y4, unsigned char *B1,unsigned int *A1,unsigned char *B2,unsigned int *A2,
unsigned char *B3,unsigned int *A3,unsigned char *B4,unsigned int *A4)
{

unsigned char tempSTx1,tempSTy1,tempSTx2,tempSTy2,tempSTx3,tempSTy3,tempSTx4,tempSTy4;

unsigned char BST[8][8]={
{1,2,3,4,5,6,7,8},      {12,9,4,3,6,5,8,7},     {13,14,10,2,7,8,5,6},
{14,13,15,11,8,7,6,5},  {15,1,9,14,16,2,3,4},   {9,15,11,16,13,12,4,3},
{10,16,12,1,10,14,10,2},        {11,1,13,9,11,15,12,16},
}; / /Bank map
#pragma HLS ARRAY_PARTITION variable=BST complete dim=0

unsigned char AST[8][8]={
{0,0,0,0,0,0,0,0},      {0,0,1,1,1,1,1,1},      {0,0,0,1,2,2,2,2},
{1,1,0,0,3,3,3,3},      {1,1,1,2,0,2,2,2},      {2,2,1,1,2,1,3,3},
{1,2,2,2,2,3,3,3},      {2,3,3,3,3,3,3,3},
}; / /Intra-tile address offset
#pragma HLS ARRAY_PARTITION variable=AST complete dim=0

while(1)
{
#pragma HLS PIPELINE II=1
/ /Calculate intra-tile coordiante
tempSTx1=*X1%ST; tempSTy1=*Y1%ST; tempSTx2=*X2%ST;
tempSTy2=*Y2%ST; tempSTx3=*X3%ST; tempSTy3=*Y3%ST;
tempSTx4=*X4%ST; tempSTy4=*Y4%ST;
/ /Bank assingment
*B1=BST[tempSTy1][tempSTx1]; *B2=BST[tempSTy2][tempSTx2];
*B3=BST[tempSTy3][tempSTx3]; *B4=BST[tempSTy4][tempSTx4];
/ /Address Assignment
*A1=AST[tempSTy1][tempSTx1]+floor(*X1/ST)+HC*floor(*X1/ST);
*A2=AST[tempSTy2][tempSTx2]+floor(*X2/ST)+HC*floor(*Y2/ST);
*A3=AST[tempSTy3][tempSTx3]+floor(*X3/ST)+HC*floor(*Y3/ST);
*A4=AST[tempSTy4][tempSTx4]+floor(*X4/ST)+HC*floor(*Y4/ST);
}
}
```

Figure 5.9: Code template for the mapping function of the GE with a supertile of size 8x8. [2] DOI 10.1145/3195970.3196088

Results

To determine if there was an optimal tile size we first generated two figures per kernel. One with the ratio between self-conflicts for each tile size of up to 10×10, and another with the partition factor needed to eliminate all inter node conflicts for a given tile size, again up to 10×10. These results can be seen in table 5.1 and 5.1. The [Y,X] coordinates in each figure correspond directly to the size of tile used.

As we can see from table 5.1, right column, for the stencil kernels, intra-node conflicts (self-conflicts) can be reduced to 0 after a certain supertile size smaller than the problem size. Once in this region, the best solution is to pick the corresponding location in the left column such that it has the smallest partition factor. Note that due to timing and processing power constraints, an approximate coloring algorithm was used. Using an optimal coloring algorithm should yield the same "best" tile size and partition factor as [17].

Table 5.2 on the other hand shows,on the right column,that for the non-stencil kernels, intra-node conflicts (self-conflicts) cannot be reduced to 0 in general unless the entire problem size is used as a tile. In this case, we must decide an arbitrary tile size such that the ratio of unavoidable conflicts is less than the maximum allowed according to the performance requirements. One can then choose from all the valid tile sizes the one that offers the smallest partition factor (left column).

Table 5.1: Number of colors using approximate coloring to ensure no inter-node conflict (Left) and ratio of intra-node, self-conflict, vs total conflicts (Right)) for a particular stencil kernel with tile size X $\times$ Y. [2] DOI 10.1145/ 3195970. 3196088

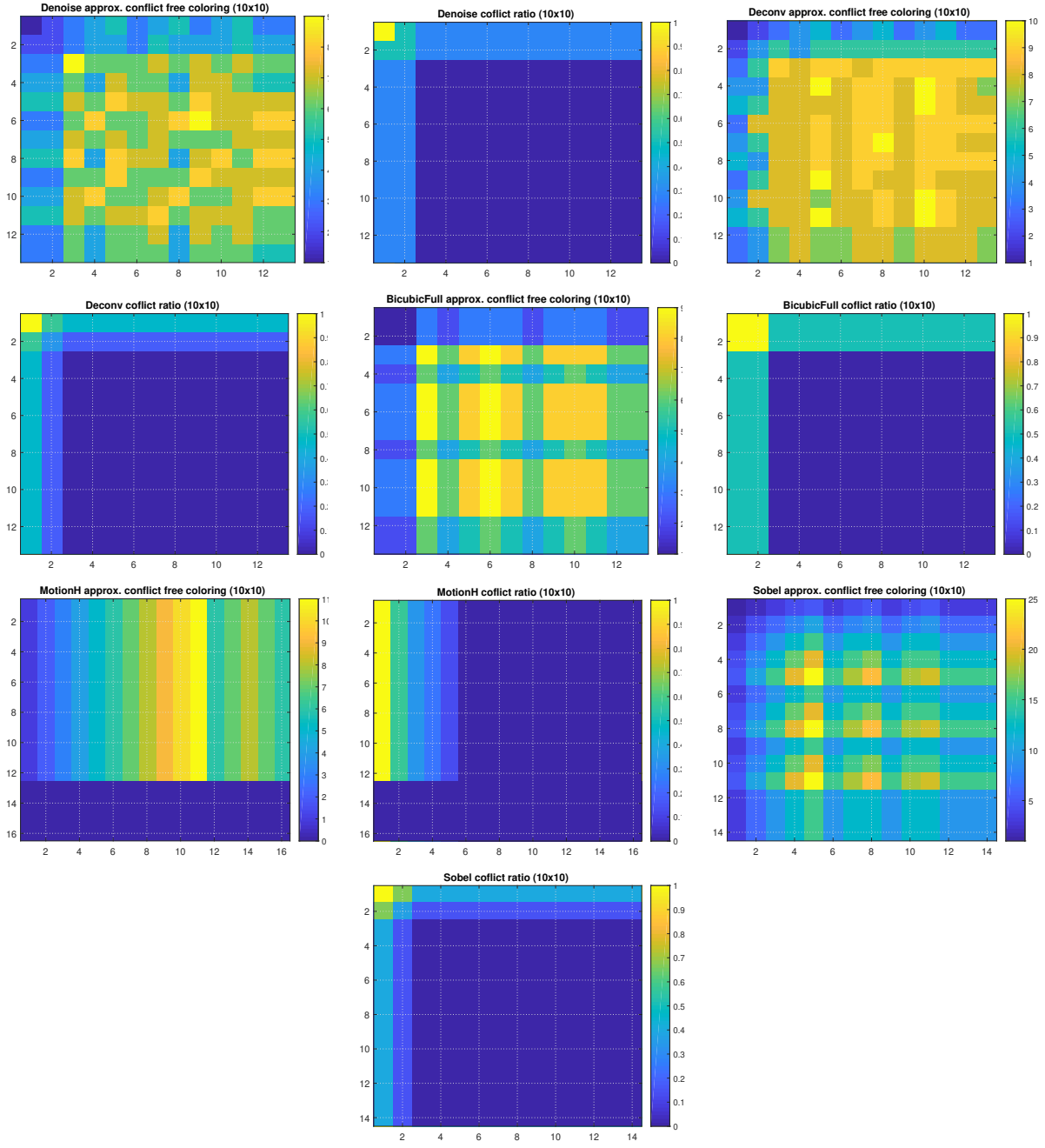
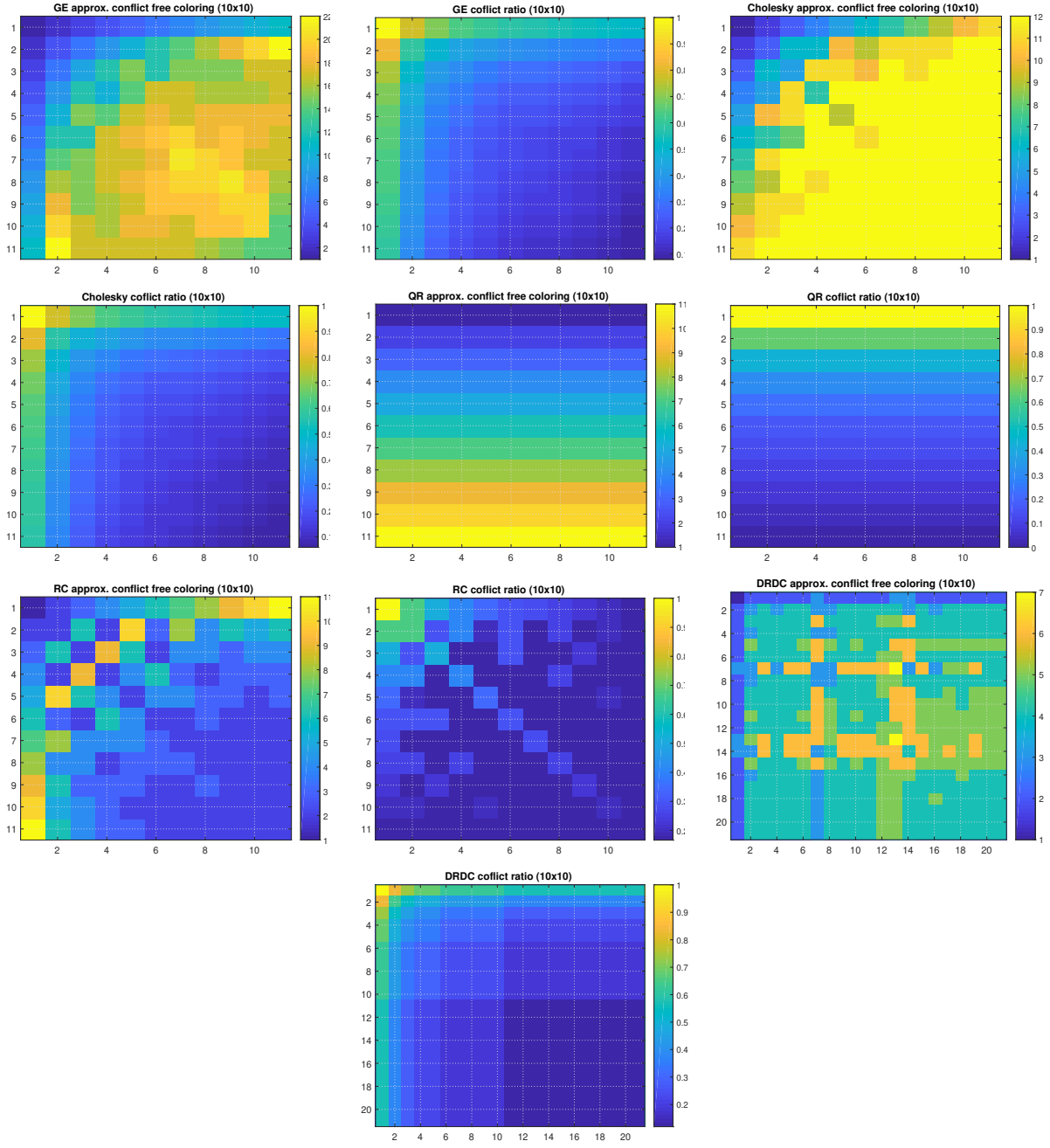


Table 5.2: Number of colors using approximate coloring to ensure no inter-node conflict (Left) and ratio of intra-node, self-conflict, vs total conflicts (Right)) for a particular non-stencil kernel with tile size X $\times$ Y. [2] DOI 10.1145/ 3195970.3196088



Given the tile size is dependent of particular performance requirements for non-stencil kernels and most stencils are smaller than a 6x6 rectangle, we chose test tile sizes of 8x8, 16x16, and 32x32 since they allow for an efficient hardware implementation of multiplexers. Using these tile sizes we obtained the number of conflicts remaining when one allowed up to n banks to be used. Figures 5.10 through 5.15 show the conflict ratio for all the test kernels using the aforementioned supertile sizes and a partition factor (number of banks) of up to 32.

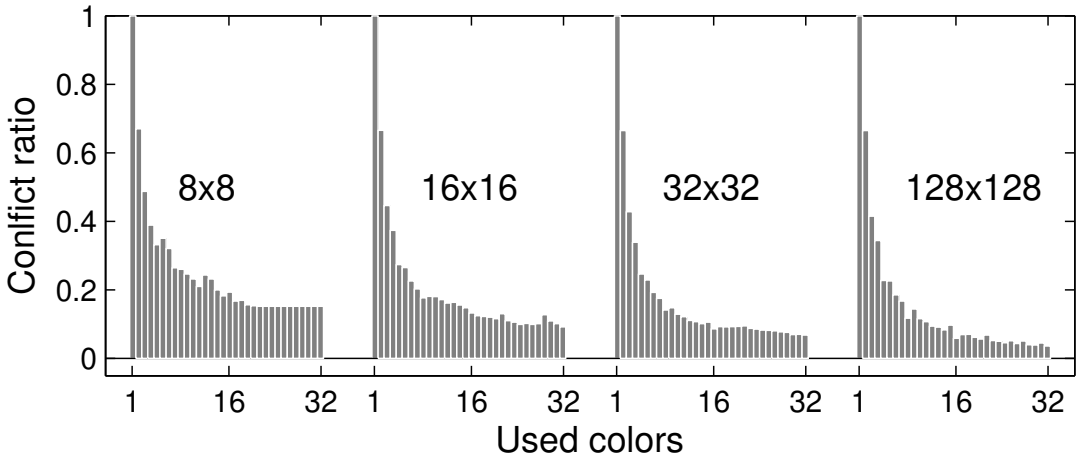


Figure 5.10: Conflict ratio for the GE kernel on a problem of size 128x128 and a supertile of size: 8x8, 16x16, 32,x32, and 128x128. [2] DOI 10.1145/ 3195970. 3196088

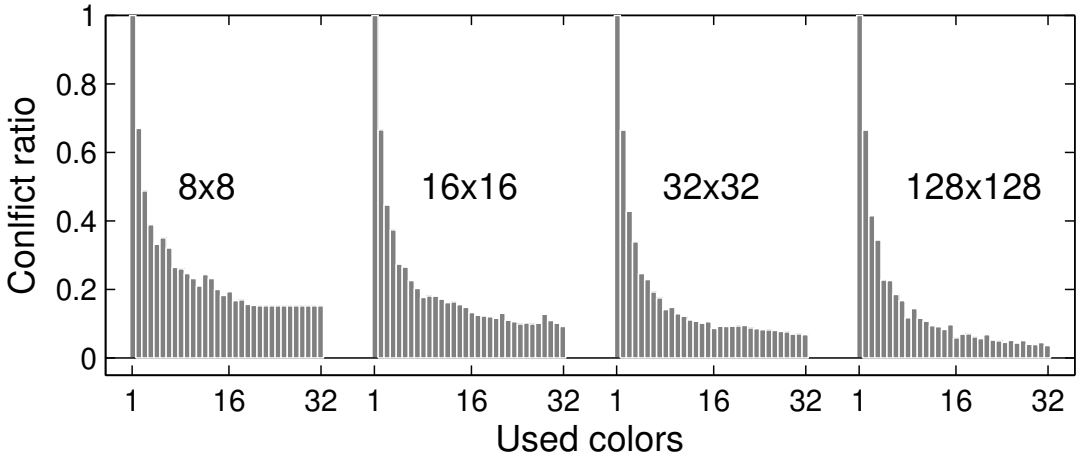


Figure 5.11: Conflict ratio for the GE kernel on a problem of size 128x128 and a supertile of size: 8x8, 16x16, 32,x32, and 128x128. [2] DOI 10.1145/ 3195970. 3196088

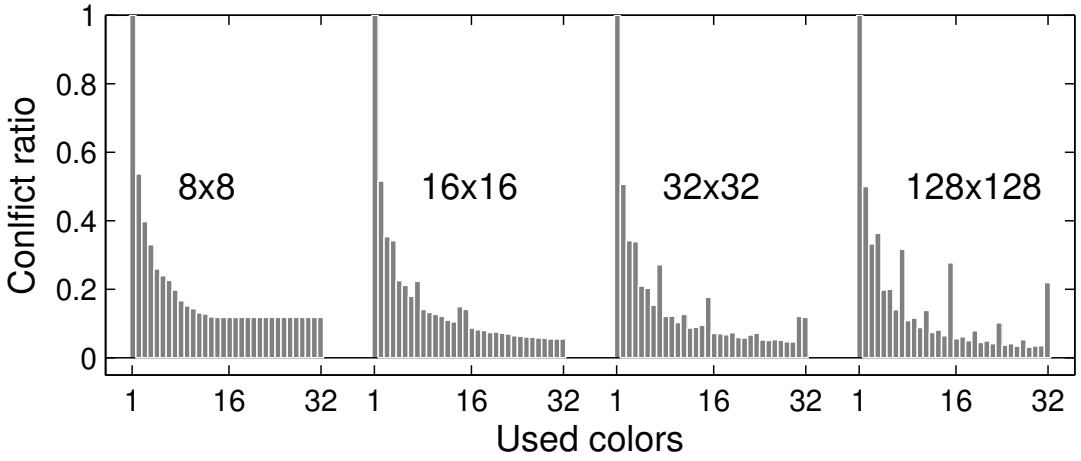


Figure 5.12: Conflict ratio for the Cholesky kernel on a problem of size 128x128 and a supertile of size: 8x8, 16x16, 32,x32, and 128x128. [2] DOI 10.1145/ 3195970. 3196088

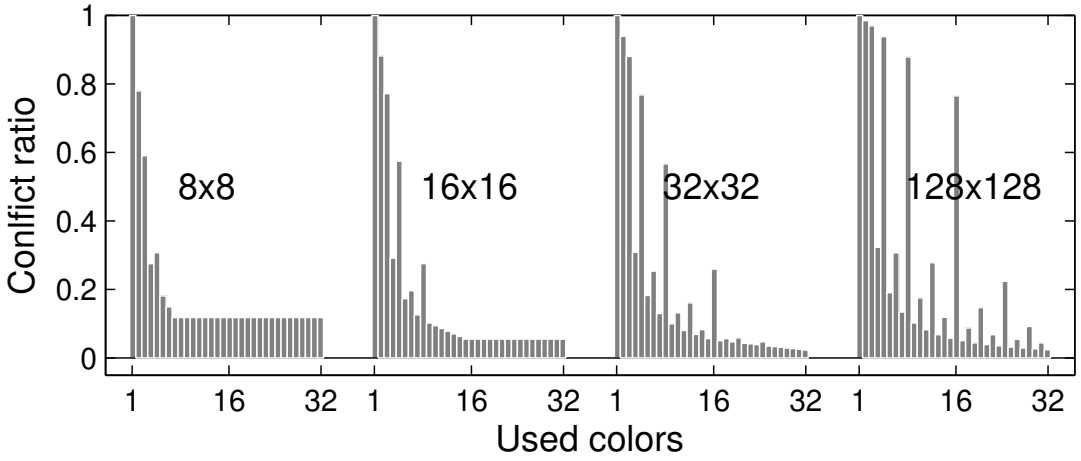


Figure 5.13: Conflict ratio for the QR kernel on a problem of size 128x128 and a supertile of size: 8x8, 16x16, 32,x32, and 128x128. [2] DOI 10.1145/ 3195970. 3196088

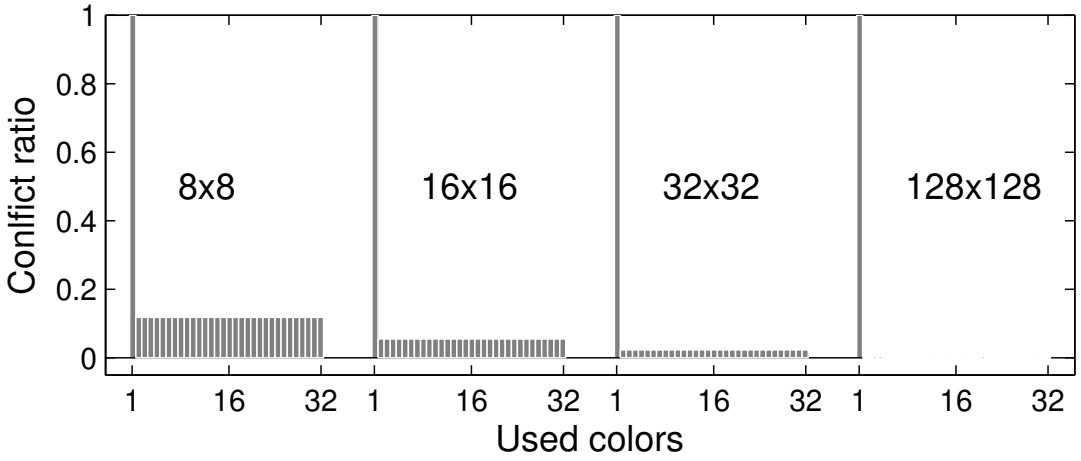


Figure 5.14: Conflict ratio for the RC kernel on problem of size 128x128 and a supertile of size: 8x8, 16x16, 32,x32, and 128x128/ [2] DOI 10.1145/ 3195970. 3196088

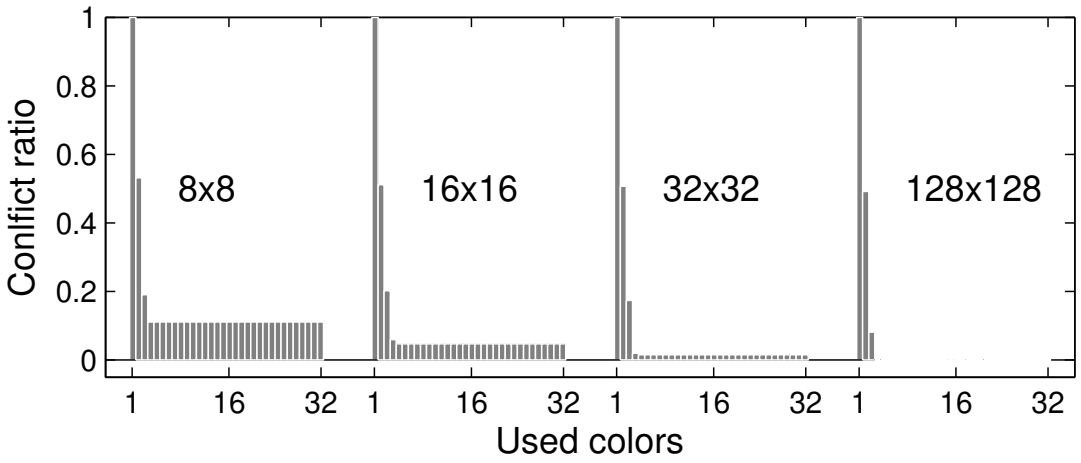


Figure 5.15: Conflict ratio for the DRDC kernel on a problem of size 128x128 and a supertile of size: 8x8, 16x16, 32,x32, and 128x128. [2] DOI 10.1145/ 3195970. 3196088

As we can see from figures 5.10 through 5.15 both the geometry of the supertile and the partition factor (used banks) have an effect on the total conflict reduction.

In general, for a fixed supertile size, the more banks are available, the greater the conflict reduction. This trend continues until there are enough colors to ensure all adjacent nodes can be assigned a distinct color. The number of banks required to reach this point will vary depending on the coloring strategy implemented. A greedy approach like the one used in this paper will most likely not give the optimal coloring scheme except for the most simplest of conflict graphs such as the DR and DRDC kernels. Once all inter-node edges are removed, only the self conflicts will contribute to the total amount of conflicts. The only way to eliminate or modify these self conflicts is by changing the geometry of the supertile. This effect can be seen by comparing the number of conflicts for a given partition factor between two supertile sizes.

Note that some kernels, most notably the QR factorization in figure 5.13, posses a much higher

conflict ratio for a particular partition factor than previous, smaller ones. This is most likely due to a compound effect different of factors such as the color selection priority of the coloring algorithm, the number of available colors, and the resulting structure of the conflict graph. These factors influence the order in which the nodes are colored, and the total amount of available colors determines the distribution of the colors. The algorithm tries to balance the colors used in order to achieve an uniform distribution (no color is used significantly more than any other), the effects of the greedy selection can accumulate over time leaving low-weight edges considered at the end of the algorithm connecting same-colored nodes . However, one can still observe the aforementioned conflict reduction trend.

Table 5.3 shows the resource utilization and achievable clock period for a fully pipelined (II=1) implementations of the mapping function for the 4 accesses of the GE kernel on a problem size of 128x128 elements for different supertile sizes.

Table 5.3: Resource utilization for the GE kernel, problem size 128, and different supertile sizes. [2] DOI 10.1145/3195970.3196088

ST size	Resource	Utilization	Available	Utilization (%)	CP (nS)
8x8	LUT	8936	101400	8.81	8.55
	LUT RAM	1	35000	0.01	
	FF	6435	202800	3.17	
	BRAM	7	325	2.15	
	DSP	68	600	11.33	
16x16	LUT	9094	101400	8.97	8.6
	LUT RAM	1	35000	0.01	
	FF	6467	202800	3.19	
	BRAM	7	325	2.15	
	DSP	68	600	11.33	
32x32	LUT	9698	101400	9.56	8.6
	LUT RAM	1	35000	0.01	
	FF	6480	202800	3.2	
	BRAM	7	325	2.15	
	DSP	68	600	11.33	

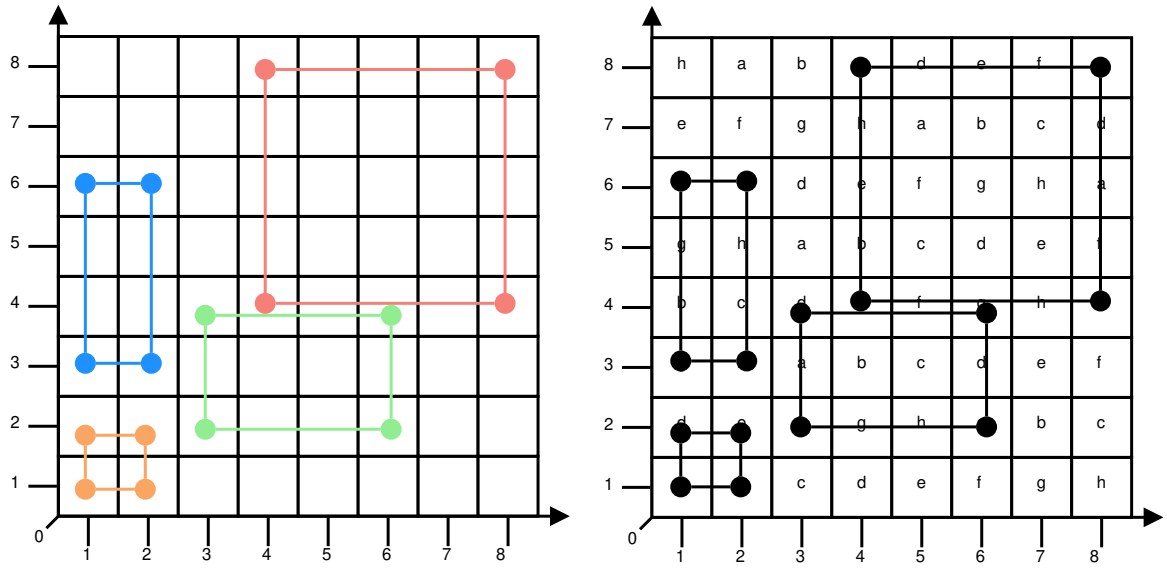
As we can see from the table, the resource utilization does not change in any significant way when the supertile size increases from 8x8 to 32x32. This is in part due to the size of the supertiles being a power of 2, which allows for better resource optimization. Another observation is that the number of BRAM used remained constant at 7. The only parameter which remained constant

was the partition factor, which was set at 16. This tells us that the bulk of the mapping was likely performed by this BRAM, and since the number of IO remained constant, the small difference in the LUT and FF is due to the maximum intra bank address, which reaches a higher value for bigger supertiles.

CHAPTER 6: NON-LINEAR TRANSFORMATION BASED APPROACH FOR OPTIMAL MEMORY BANKING IN QUASI-STENCILS

This chapter is based on our previously published work in Juan Escobedo and Mingjie Lin, Exploiting Irregular Memory Parallelism in Quasi-Stencils through Nonlinear Transformation, 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) ©2011 IEEE [20] and encompasses our novel approach for handling non-stencil code and finding an optimal banking scheme using non-linear transformations.

We will begin this chapter with a motivational example that will be used throughout this chapter to demonstrate concepts and exemplify the contribution of this work.

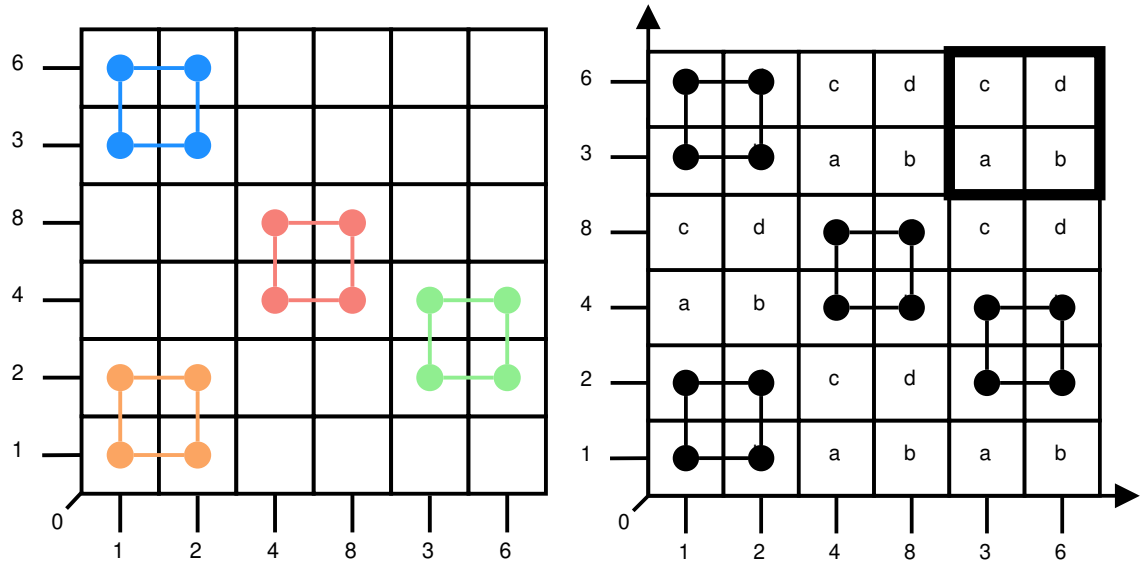


(a) (b)

Figure 6.1: (a) Memory access geometry for 4 distinct iterations in the original space: (i,j)=[(1,1),(2,3),(3,1),(4,4)]. Note the geometry changes. (b) Memory partition with 8 banks using the GMP method from [3]. Number of banks is proportional to the problem size. ©2019 IEEE

It is well-known that the irregular memory access pattern found in non-stencil kernel computing renders the well-known hyperplane- [3], lattice- [16], or graph-based [1] HLS techniques almost totally ineffective. This is because that all these approaches rely on exploiting the repeated patterning of memory accesses, thus can not effectively handle non-repeatable or irregular memory accesses. Consider a code segment of a loop with two independent loop variables $i : 1 \rightarrow n-1$ and $j : 1 \rightarrow m-1$. Its loop statement is $S = f(M[i,j], M[2i,j], M[i,2j], M[2i,2j])$ and operates on a 2-D data matrix. Clearly, this loop is an example of non-stencil kernel because the relative distance between its accessed memory locations varies with its iterations. To illustrate, we plots its four iterations, $(i,j) = [(1,1),(2,3),(3,1),(4,4)]$, in Fig. 6.1(a). If we directly apply the classical

GMP memory banking scheme [3] to this irregular non-stencil case, as shown in Fig. 6.1(b), we will require 8 independent memory banks for a mere 8×8 matrix, which makes directly utilizing the GMP method infeasible for any realistic input data size.



(a) a (b) b

Figure 6.2: (a) Memory access geometry for 4 distinct iterations in our transformed domain: (i,j)=[(1,1),(2,3),(3,1),(4,4)]. Note the constant shape. (b) Memory partition with 4 banks using the ESG method in [1]. Number of banks is independent of problem size. Work in [3] given the same number of banks but different layout. ©2019 IEEE

Fortunately, as discussed in Section 6, the motivational code segment shown in Fig. 6.1 can be classified as a quasi-stencil kernel. A quasi-stencil code is a type of non-stencil and affine kernel code for which we can find a non-linear data domain layout transformation based on prime factorization such that the code behaves effectively as a stencil in this new data layout. Further details regarding the definition of quasi-stencil and its requirements can be found in Section 6 while Section 6 contains more information regarding the non-linear transformation that converts a quasi-stencil code

into a stencil-based one. After this nonlinear memory transformation, the modified data domain of the non-stencil code depicted in Fig. 6.1 is shown in Fig. 6.2. Here we access the same memory locations with the same indexes during the same iteration but comparing the relative distances of the memory locations accessed it is evident that now we have code that behaves like a stencil. This allows us to use the vast repertoire of memory partitioning algorithms that exist in literature. Applying the partitioning algorithm from [1], we only need 4 memory banks to do the partition, which is the optimal solution given the fact that we only have 4 memory accesses. Furthermore, our solution is independent of the problem size, meaning it scales well for larger problems.

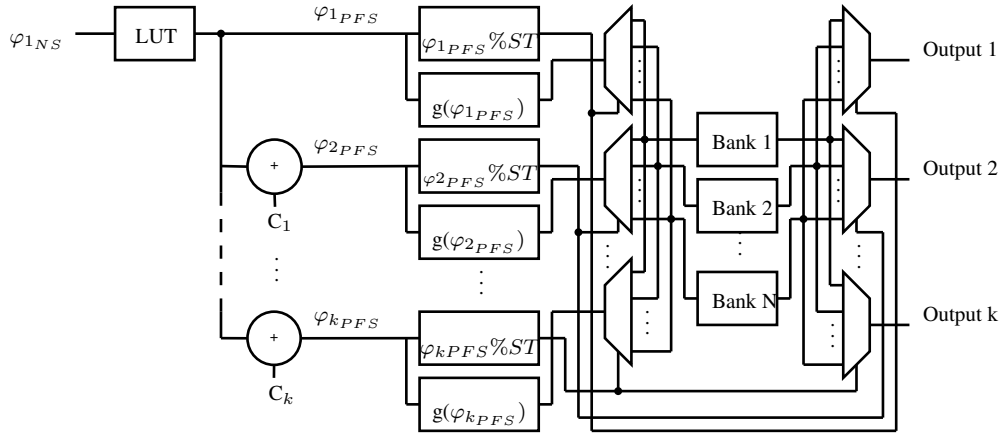


Figure 6.3: Circuit diagram of our implementation. After our layer of indirection represented by a LUT, the circuit schematic remains the same as traditional banking schemes. ©2019 IEEE

Finally, to further demonstrate the effectiveness of our methodology, we present one possible circuit implementation of our method in Fig. 6.3. In general, our hardware requirements are very similar to that of the traditional memory banking schemes for stencil computations [1, 3, 16] with only an addition of an extra layer of indirection which can be implemented using LUT's or even BRAM. Because the memory pattern is a stencil in the transformed domain we only need to calculate one of the addresses in the new domain and the others can be inferred by the offset of the

stencil. Further details re-gathering implementation details will be discussed in Section 6.

Overall Methodology

In this section, we will first present the definition of quasi-stencil code, the requirements to be classified as one, and some additional cases that can be transformed into Quasi-stencil under certain conditions. In addition, we discuss the mathematical theory behind our nonlinear memory transformation, i.e., the prime factorization space and its linearization, as well as its hardware implementations details.

Quasi-stencil: Definition and Criteria

We define a quasi-stencil memory access pattern as a kind of affine and non-stencil kernel code where each memory access R_k can be written as:

$$R_k = A_k \cdot \vec{i}, \tag{6.1}$$

where each A_k is a square $n \times n$ non-singular and diagonal matrix such that $A_i \neq A_j, \forall i \neq j$, n is the loop depth, and $\vec{i}$ is the iteration vector. In this work, we consider only a perfectly nested loop or its equivalent one. This condition leads to the observation that each dimension is controlled by a single loop variable, which is the same for all accesses, but with different step sizes for each. This kind of code has the property that now the exploration of all Data Domain dimensions is independent of the other and we can proceed to analyze them independently. By doing this, we can perform analysis on a 1D case and use it to extend the results to an n-dimensional problem.

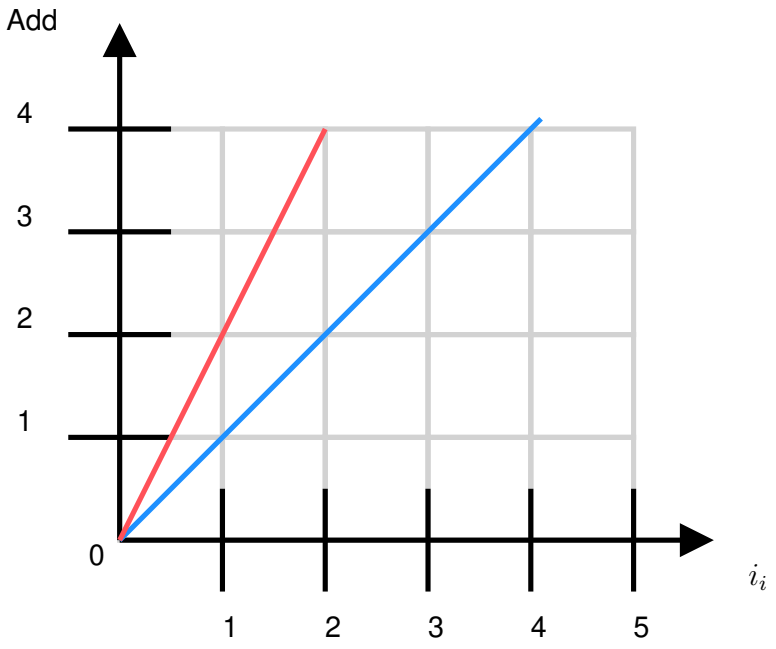


Figure 6.4: Memory accesses formed by the lines φ_{1_1}:i(blue) and φ_{1_2}=2*i(red) in the plane ID,DD with as single intersection point at the origin. ©2019 IEEE

In order to intuitively visualize the behavior of a non-stencil kernel code, we develop a new diagram, where each memory access along each dimension can be represented as a straight line with slope $A_{k_{i,i}}$ for access k in dimension i. For the aforementioned conditions on the A matrix in Equation 6.1, all such lines intersect at the origin. In other words, for an access to a n-dimensional memory in the form $M(\varphi_1, \varphi_2, \ldots, \varphi_n)$, the value of the coordinate for each dimension can be expressed as:

$$\varphi_i = A_{k_{i,i}} \cdot \vec{i}. \tag{6.2}$$

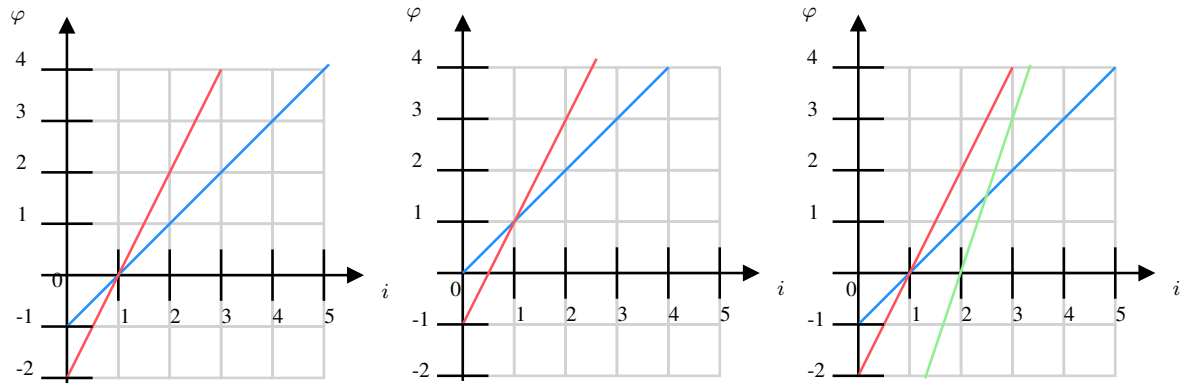
For example, in the code segment listed in Section 6, two such line diagrams of the first two accesses $M(i, j)$ and $M(2 * i, j)$ along the first dimension are depicted in Fig. 6.4.

Generalizing Quasi-Stencil

To accommodate various memory access patterns found in real-world applications, we now attempt to generalize the cases of non-stencil kernels that we can handle, especially these with a constant in its address. Let us consider the case where we have memory accesses of the form:

$$R_k = A_k \cdot \vec{i} + b_k, \tag{6.3}$$

Still with the same restriction as before on the A_k matrices in terms of being non-singular, $n \times n$, diagonal matrices all different from each other. We consider three special cases that we can handle, each of which will require a particular modification to its data domain besides the non-linear transformation.



(a) a (b) b (c) c

Figure 6.5: (a) Single intersect at $\varphi = 0, i \neq 0$. $\varphi_{1_1} : i - 1$(blue) and $\varphi_{1_2} = 2(i-1)$ (red). (b) Single intersect at $\varphi \neq 0, i \neq 0$. $\varphi_{1_1} : i$(blue) and $\varphi_{1_2} = 2i - 1$ (red). (c) Multiple intersect, integer delay. $\varphi_{1_1} : i - 1$ (blue), $\varphi_{1_2} = 2(i-1)$ (red), and $\varphi_{1_2} = 3(i-2)$ (green). ©2019 IEEE

Single intersect point at $\varphi_k = 0, i \neq 0$

If the intersect point is on the x axis, corresponding to the loop variable controlling the movement in the corresponding dimension, we do not need to take extra considerations. The prime factorization space will natively give a representation of the code. An example of this can be seen in figure 6.5 (a).

Single intersect point at $\varphi_k \neq 0$

If the intersect point is at any point in the x axis such that it is not on $\varphi = 0$, as seen in Fig 6.5 (b) where the intersect point is at (1,1), we take note of the coordinate in the y axis, corresponding to the φ. This will be used to modify the Data Domain.

Multiple intersect points with integer time delay

There are cases such as the ones seen in Fig. 6.5 (c) where if we delay the execution of one or more memory access we can make the lines intersect at one point. Depending if the y coordinate is 0 or not we end up with one of the previous cases. For (c), if we delay thee execution of the memory access corresponding to the green line by -1 iterations, meaning we advance the execution of the code one iteration, we end up with all lines intersection at the point (1,0). As it is evident, this is one of the previous cases which we can handle.

Prime Factorization Space

The Prime Factorization Space belongs to the logarithmic-space family of memory layouts. We take advantage of the property of logarithms to transform multiplications into addition in order to

obtain a memory layout that converts non-stencil code that meets the quasi-stencil criteria defined in Section 6 and 6 into a stencil. To explain this basic idea, we consider a simple example of a two-element non-stencil memory accesses consisting of $\varphi_1 = a_1 \cdot i$ and $\varphi_2 = a_2 \cdot i$ with $a_1 \neq a_2$ and i is a loop variable. As iteration progresses, it is obvious that the distance between the accesses $\Delta d = d_1 - d_2 = a_1 \cdot i - a_2 \cdot i = (a_1 - a_2) \cdot i$ will change according to i, thus clearly a non-stencil kernel code. Now if we calculate the logarithm of each memory address, we will obtain $\varphi_1' = \log \varphi_1 = \log(a_1 \cdot i) = \log a_1 + \log i$ and $\varphi_2' = \log \varphi_2 = \log(a_2 \cdot i) = \log a_2 + \log i$. Now, if we take the difference of the accesses in this logarithmic space, we will obtain $\Delta d = \varphi_1' - \varphi_2' = \log a_1 + \log i - (\log a_2 + \log i) = \log a_1 - \log a_2 = \log\left(\frac{a_1}{a_2}\right) = c$, which clearly is a constant and shows that, in the log space, the distance between the two memory accesses is independent of the iteration we are in, thus behaving exactly like a stencil. The iteration will now just center the access around a memory location but the relative position of the accesses will remain he same.

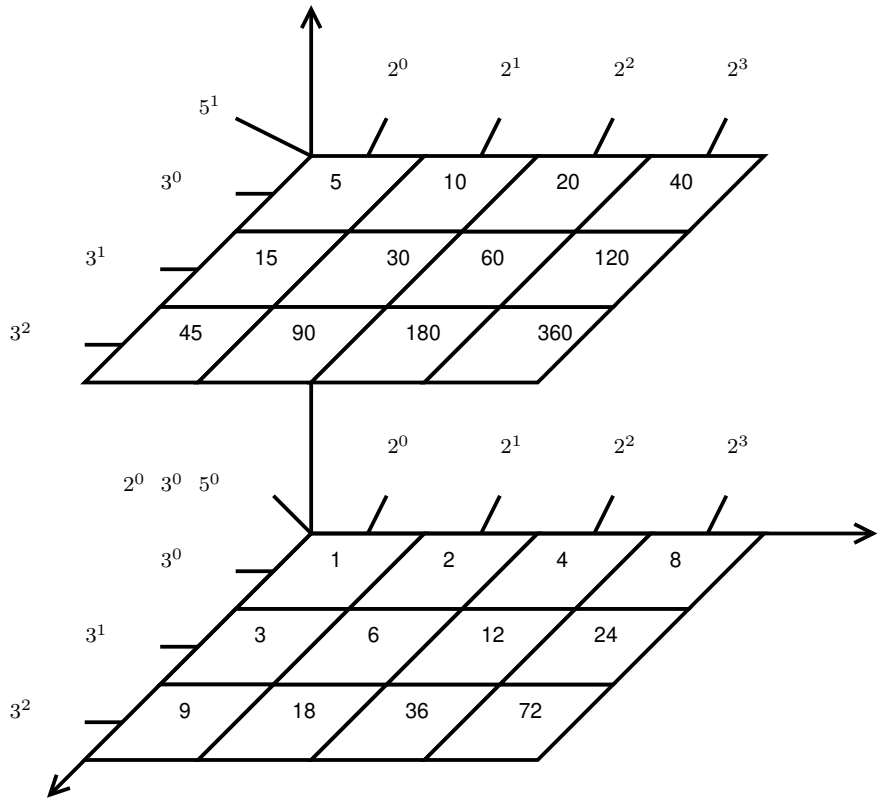


Figure 6.6: Partial PFS for the first 3 primes 2, 3, and 5. We can represent numbers from 1-6 without gaps. ©2019 IEEE

However, directly performing such logarithmic operations to memory addresses will need to represent decimal values that implies the need for floating point arithmetic. We circumvent this issue through utilizing a spacial case of the logarithm spaces: the Prime Factorization Space.

It is well-known that any number can be written as a product of prime numbers. This is, any rational number n can be written as $n = 2^{k_1} \cdot 3^{k_2} \cdot 5^{k_3} \cdot \cdots \cdot p_n^{k_n} \ldots$. If we calculate the log of n, we obtain $\log n = \log(2^{k_1} \cdot 3^{k_2} \cdot 5^{k_3} \cdot \cdots \cdot p_n^{k_n} \ldots) = k_1 \log 2 + k_2 \log 3 + k_3 \log 5 + \cdots + k_n \log p_n + \ldots$, where k_n is an integer. With this in mind, we can consider each of the integers k_n as the coordinate of the number in a m-dimensional space, where m is the number of primes below the maximum

address of the memory we will ever access during the execution of the program. Each of this m dimensions is a 1-D space corresponding to one of the primes and movement along this dimension corresponds to changes of the exponent of the corresponding prime in the prime factorization of the number. Now we can represent all integer numbers in log space with integer coordinates. Even negative numbers since we can mirror the space and consider negative numbers as movement in the negative direction (towards this mirrored space). We can see an example of how a partial prime factorization space looks for the first 3 primes, namely: 2, 3, and 5, forming a 3D space in Fig. 6.6. Note that numbers where we have more than 1 non-zero coordinate corresponds to the multiplication of all the corresponding primes based on which coordinate is non-zero, to the power of the coordinate. This allows us to represent any positive rational numbers, particularly integers, natively. Unfortunately, this log-based memory space typically possesses a very high dimensionality, which makes it hard to implement with traditional memory architectures and increases the complexity of memory address calculations. In the following section, we circumvent this issue by means of linearizing such a log-based memory space.

Transformed data domain

As mentioned previously, the memory space based on prime factorization, which converts a quasi-stencil kernel code into a stencil one, can be very high-dimensional. Unfortunately, such a transformed memory space is difficult to implement with traditional memory architectures. To solve this feasibility problem, we linearize the PFS row/column wise starting from the 2D space with the smallest origin coordinate. To illustrate, given the sample PFS in Fig. 6.6, we can obtain a linearized PFS depicted in Fig. 6.7(a). This corresponds to the case in which the memory addresses of every dimension have a single intersect point at (0,0). For the case where we have an intersect point of $(x, 0)$, where $x \neq 0$, we simply add the element 0 to the beginning of the linearized space and mirror the linearized space around it as needed but with negative offset numbers. This will allow

us to represent memory locations that occurred before the intersect point as shown in Fig. 6.7(b). Note that, in general, memory locations are always positive, so negative memory locations do not occur. Therefore, this case usually only happens when we also have a non-zero intersect coordinate in the y axis. Finally, the general case where the intersect happens at (x, y) can be seen in Fig. 6.7(c). Here we simply take the value of the intersect and add it to the mirrored domain. In this case, we are considering $y = 3$ as an example. Note some of the elements on the transformed domain ($\geq$-3) now become positive and could be accessed by the code.

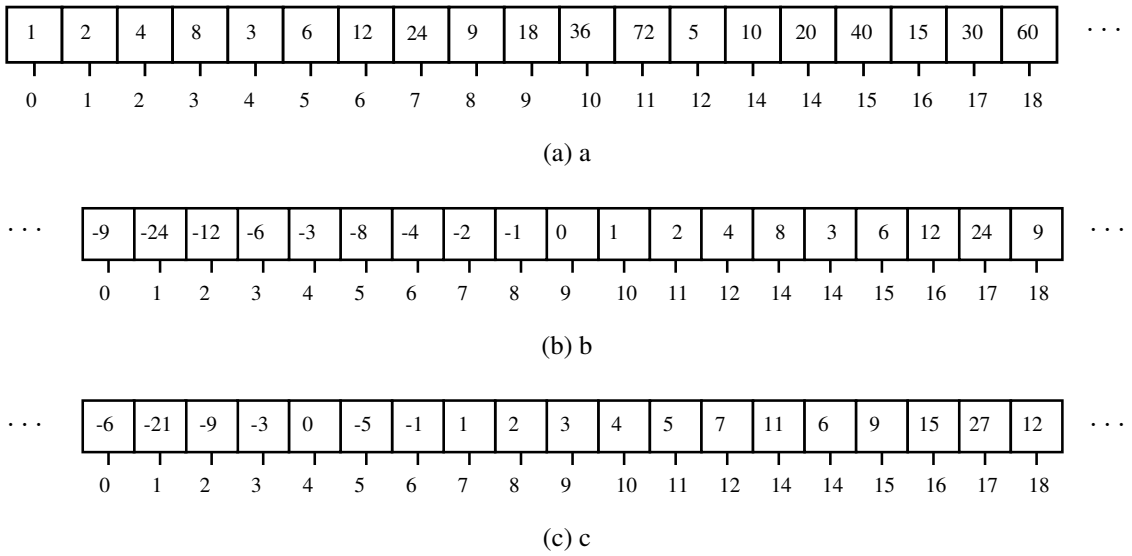


Figure 6.7: (a) Linearized PFS for the case where memory locations intersect at the origin. (b) Extended linearized PFS for the case where memory locations intersect at $(x, 0)$, $x > 0$. (c) Shifted linearized space for the case where memory locations at (x, y), $x, y > 0$. For this example $y = 3$. ©2019 IEEE

Overhead Reduction

Although it is possible to calculate the address in the linearized PFS during runtime it would be computationally expensive since it would require us to find the prime factorization of at least one of the addresses in the original domain and perform a number of Multiply-Accumulate (MAC) operations to calculate the address in the linearized PFS. To avoid this we have opted to use a lookup table that could be implement either in fabric for speed or in a BRAM if the indirection vector is large enough. This lookup table simply contains the correct address in the linearized PFS of the address in the original space. Using a lookup table also allows us to add some non-linearities to the address conversion that given the nature of the rectangular linearizion technique we use to reduce the dimensionality of the PFS would translate to an amount of wasted memory space that would render the method almost unusable.

Take the example of section 6. The full linearized PFS in both dimensions would look like the one from figure 6.8 (a). Note now the transformed domain has the same dimensionality of the original data domain. The areas in gray are memory locations that are never accessed for (i,j)$\leq$5 either because of the nature of the code or they are artificial memory addresses generated when we linearize the PFS row/column wise. A data domain that was originally only 10×10, for a total of 100 data elements now consist of a 2D grid of 17×17 elements for a total of 289 elements. And increase in the total number of elements close to 200%.

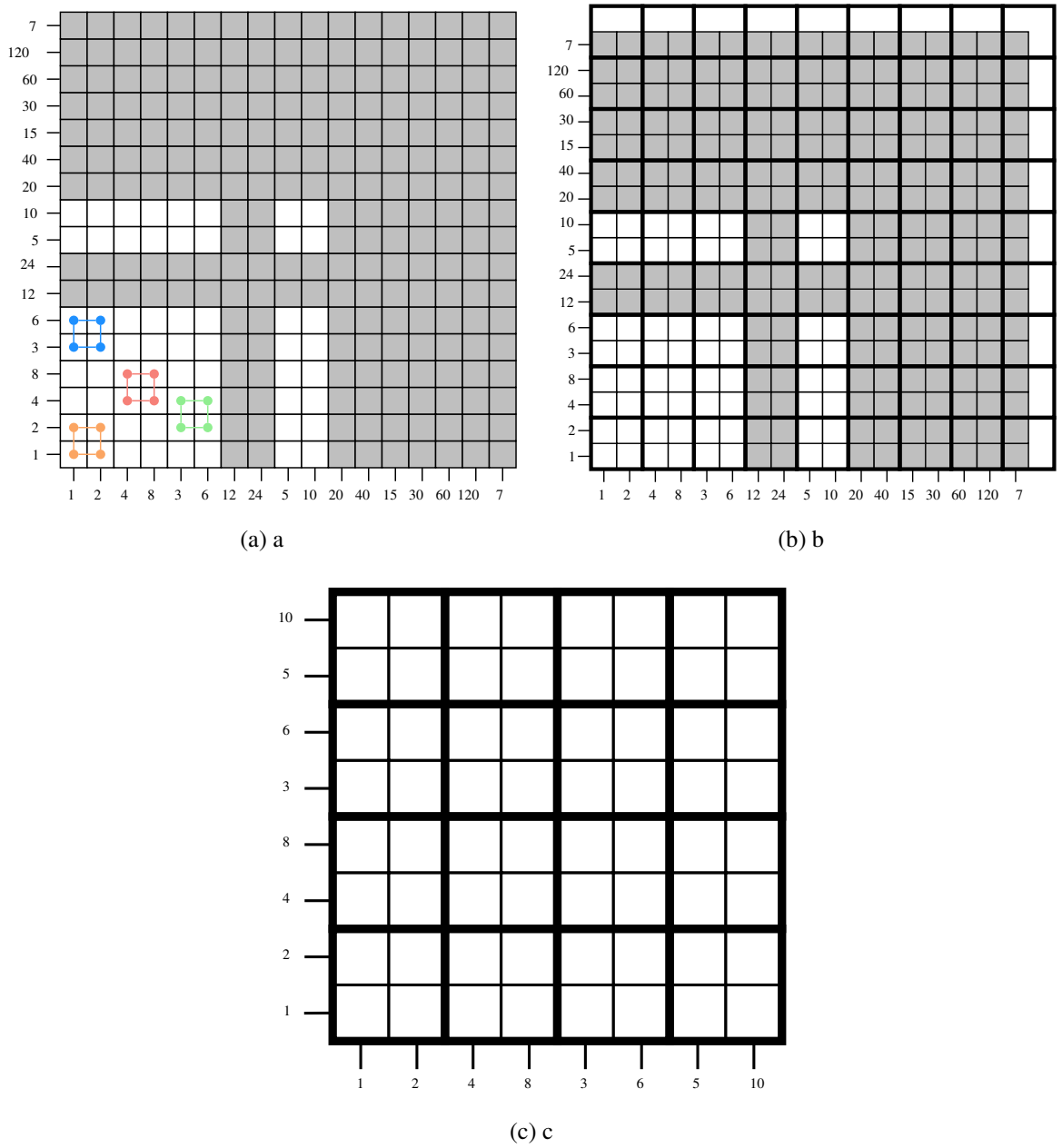


Figure 6.8: (a) Full Linearized PFS for the motivational example. Gray cells are memory locations that are never accessed for i,j≤5. (b) Full Linearized PFS with overlapped region of repeated banking. (c) Pruned linearized PFS. ©2019 IEEE

This approach, which introduces nonlinearities in the address mapping, is particular straightforward to do by using our lookup table approach to do the memory address indirection ayer instead of performing the calculation during runtime.

Implementation

For the final implementation scheme we have opted to use the pruned linearized PFS to keep memory overhead to a manageable amount while using a LUT as an indirection layer to translate addresses in the original domain to the transformed domain for hardware simplicity and speed.

Once we have transformed the problem, the quasi stencil code now behaves like a stencil and thus we are free to use any of the available stencil banking schemes existing in literature. For this work we have decided to implement the method from [1] as our banking scheme given the proven optimality results for any stencil in terms of partition factor and good performance metrics in terms of resource utilization and clock period.

For this method, bank selection is done by accessing a small memory of the same dimensionality as the original problem but much smaller size. This memory represents the smallest rectangle containing a repeating pattern in the banking scheme. The size of this memory is independent of the problem size and only depends on the geometry of the stencil. The access is done by applying modulo operations to each of the dimensions of the transformed address by the size of the memory in the corresponding dimension to obtain the intra-tile index. Accessing it provides the right bank for parallel, conflict-free access for a given stencil.

$$\mathrm{B} = \mathrm{Off}_{\mathrm{ST}}(\varphi' \% \mathrm{ST}) \tag{6.4}$$

The intra bank address φ_{B} can be calculated from the transformed memory address φ' by the formula seen in Eq. 6.5:

$$\varphi_{\mathrm{B}} = \mathrm{Off}_{\mathrm{ST}}(\varphi' \% \mathrm{ST}) + \sum_{i=0}^{n} (\varphi'_i / M_i) \cdot k_i \tag{6.5}$$

As mentioned in section 6, the intra-bank address function is composed of 2 parts: the first is the same as the bank assignment, an access to a small memory of the same dimensionality as the original problem and applying modulo operations in the corresponding dimension by the size of the tile in that dimension. exactly the same as accessing the memory containing the banking information. This memory contains the number of accesses to a given bank given a certain exploration order. The second corresponds to an accumulation operation where we count the number of tiles that have happened in lower dimensions by diving the coordinate by the size of the tile in that dimension. The constant k_i is calculated off-line and contains the number of elements in each bank there are in each tile per row, plane, cube, etc. While in native stencil code the accumulation operation can be implemented via counters given the regular exploration of the data space, in this case, since the center coordinates of the stencil we area accessioning depends on the prime factorization of an address which to the best of our knowledge, will not produce a predictable pattern for any arbitrary sequence of addresses and thus we need to perform the necessary multiplication and divisions in real time.

Experimental Setup and Results

To test the validity of our approach we implement our method on a workstation with 16GB of RAM, an Intel i7-4770 processor, running the latest build of Windows 10.

We use the code from figure 6.9 as our test cases. We try matrix sizes of 20,40,and 80 elements for the 1D cases as a proof of concept and 1080x1080 for the 2D cases to demonstrate the real effectiveness of our method.

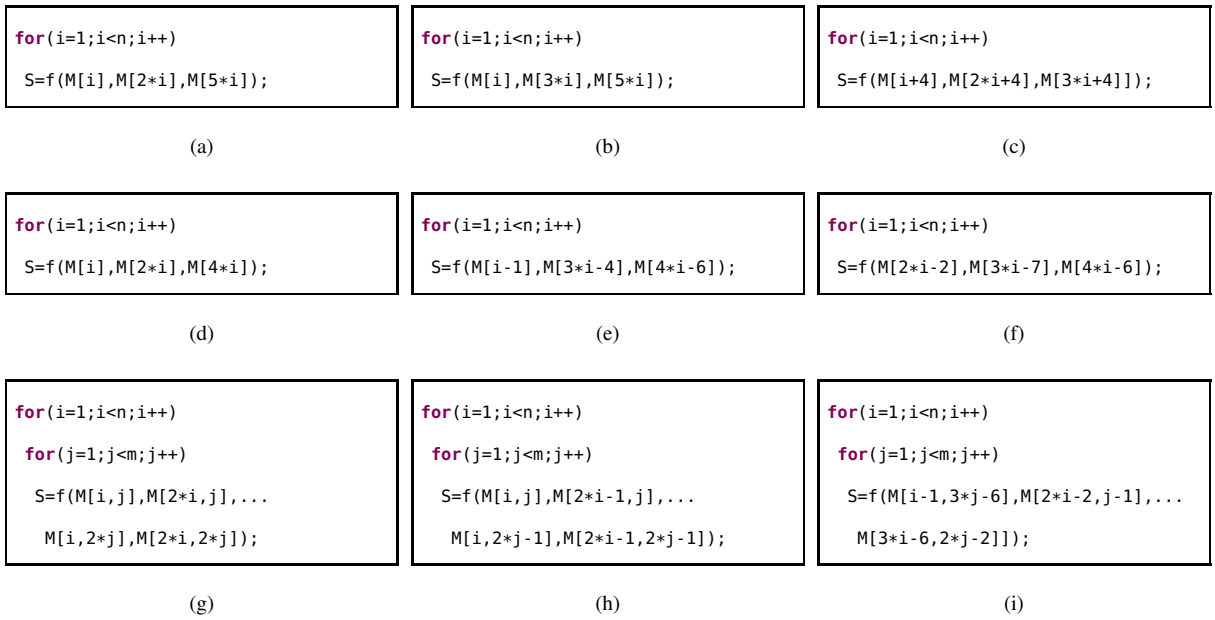


Figure 6.9: (a),(b) and (g) Code for the Base case. (c), (d), (h) Code for the Single Intersect, Non-Zero case. (e), (f) (i) Code for the Multiple Intersect. ©2019 IEEE

Our methodology is as follows: we first input the matrices that represent the affine, non-stencil memory accesses and loop bounds to Matlab 2017a and we calculate the intersection points of all the lines. Note that this is an intermediate representation of the code that can also be automatically extracted from a piece of code by more sophisticated software such as LLVM.

Once this is done, we use the classification of the intersection point to determine which of the three categories each memory access pattern falls into: Base, where all lines intersect at the origin. Single Intersect, Non-Zero (SI,NZ), Where all lines intersect at one point with address different from zero(0). And finally Multiple Intersect (MI), where the lines have multiple intersection points

but can be made to converge by adding an integer delay to one or more of the memory accesses.

We obtain the PFS and the stencil shape for the current problem size. We use the method from [1] to obtain the partition factor as well as the banking scheme. Once we have the size of the Super Tile, we can apply our pruning procedure to reduce the memory overhead. From the pruned PFS we can obtain our LUT with the address for the translation table. We use all these parameters to generate C code that will execute our algorithm that will be used by Vivado HLS 2015.4 to generate Verilog code which will be then synthesized and implemented by Vivado HLx 2015.4 on a Kintex 7 xc7k160tffg-1 FPGA. We can see the results obtained in terms of memory overhead with respect to original data space size, as well as partition factor, clock period and resource utilization in table 6.1.

Table 6.1: Partition factor, memory overhead, clock period, and resource utilization for all the test cases for different problem sizes. ©2019 IEEE

Cat.	Test Case	Bank #	Prob. Size	Overhead(%)	CP(nS)	LUT	FF's	DSP	Power(mW)
Base	(a)	3	20	65	3.25	1836	2277	3	411
		3	40	72.5	3.3	1856	2276	3	409
		3	80	76.5	3.3	1879	2319	3	443
	(b)	4	20	500	3.25	1857	2308	3	404
		3	40	485	3.25	1949	2332	3	436
		3	80	635	3.25	2107	2463	3	453
SI,NZ	(c)	3	20	55	3.3	1789	2240	3	405
		3	40	112.5	3.35	1936	2278	3	390
		3	80	73.75	3.35	1868	2319	3	425
	(d)	3	20	55	3.35	1791	2241	3	392
		3	40	112.5	3.35	1842	2278	3	390
		3	80	73.75	3.3	1883	2319	3	4.39
MI	(e)	3	20	150	3.3	1870	2454	3	374
		4	40	265	3.3	1880	2382	7	373
		3	80	210	3.3	2088	2552	3	447
	(f)	3	20	60	3.2	1861	2421	3	389
		3	40	62.5	3.3	1902	2418	3	364
		3	80	75	3.3	1960	2506	3	409
Base	(g)	4	1080	33.3	3.51	5036	5444	34	557
SI,NZ	(h)	4	1080	33.3	4.48	5207	6043	34	621
MI	(i)	6	100	392	4.82	5981	7179	45	636

We can see that, on average, we have a memory overhead of 162% for all benchmark circuits. For

large problems running on systems with limited memory, this might make our method impractical (in some cases we have over 500% memory overhead, although in some cases the overhead can be as little as 33%). On the other hand, the partition factor achieved by our approach remains proportional to the number of memory accesses and not a function of the problem size. Note that different problems sizes can have different PFS, which in turn change the geometry of the resulting stencils and also influence the partition factor that can be achieved. Also note that, in many cases, the partition factor matches the number of memory accesses, which guarantees the optimality of our results if data reuse is not considered.

For comparison, we run the algorithm GMP described in [3] to obtain the partition factor that state-of-the-art methods would yield (see Table 6.2). For all non-stencil codes we consider, we see the method GMP needs the number of independent memory banks proportional to the problem size, while our method requires a partition factor proportional to the number of memory access. However, the GMP method incurs no memory waste, while, depending on the nature of the memory access, our method can have a high percentage of memory overhead. For the 2D code under our consideration, we observe that the MI case has almost 400% memory overhead. This clears shows the price we need to pay in order to keep the partition factor and banking interconnect simple enough to implement for a Quasi-Stencil code. Because of the large number of banks, we were unable to implement the algorithm from [3] for the 2D test kernels, which corresponds to more realistic applications.

Table 6.2: Partition factor for our method vs GMP for all considered problem sizes. Test cases (a)-(f) 20x1,40x1,80x1. (g)-(h) 1080x1080. (i) 100x100. ©2019 IEEE

Method	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)
Ours	3/3/3	4/3/3	3/3/3	3/3/3	3/4/3	3/3/3	4	4	6
GMP	20/40/80	20/40/80	20/40/80	20/40/80	20/40/80	20/40/80	1080	1080	100

Due to the lack of implementable methodologies to ensure parallel and conflict-free memory access for non-stencil code, we inputted the kernels as naive C code into Vivado HLS 2015.4 to obtain a point of reference for our results. We observed 2 behaviors: If no pragmas are given to the software, then the HLS tool resorts to data duplication. This yields a higher usage of BRAM (as many times as accesses the pattern has) but allows to keep a clock period that is close to the minimum that can be implemented on the FPGA (for our case, this was close to 2.8ns) and an unitary Initiation Interval (II). Data duplication also allows for extremely low resource utilization, because it only needs LUTs to perform simple arithmetic operations to calculate the indices of the memory accesses during every iteration. However, data duplication also means a sub-optimal utilization of on-chip resource since it cannot bring enough data during every iteration from off-chip in order to reserve space for the duplicates. As a consequence, off-chip accesses become quite costly in terms of memory access latency. The second behavior was observed when we explicitly instructed the HLS tool to avoid data duplication, as is the case when the matrix was forced to be stored on a ROM implemented using a fixed structure of BRAM through using pragmas. In this case, the software simply performed sequential memory access to the matrix, one per access, and did not try any other optimization. Therefore, as in the previous case, its hardware utilization and clock period are kept relatively low, but its initiation interval II has been increased to the number

of accesses during each kernel iteration.

In contrast, our methodology, with a small increase of its clock period due to its increased complexity in logic, can reduce the total number of clock cycles it takes to complete all the memory accesses to just one, yielding a very signification speed up, especially for memory patterns with higher access count. We believe that, if the above two behaviors are left unconstrained, the existing HLS tool, although taking the computationally least expensive route, will only achieve a sub-optimal utilization of on-chip memory resource and off-chip bandwidth, which is usually the bottleneck of the system. If better performance is desired, the programmer needs to carefully guide the software with pragmas or manually modifying the code to implement more complex algorithms based on memory partition.

CHAPTER 7: CONCLUSION AND FUTURE WORK

Conclusion

In conclusion, we have achieved our objective of compiling and improving upon our previously published work and have presented it in a coherent manner that showcases the evolution of our approaches.

In this dissertation we have updated the literature review to include additional relevant references related to memory banking and partitioning for both stencil and non-stencil kernels as well as expended the literature searched to include research done in the compiler community which served as an inspiration to some of the developed algorithms. The additions to the literature review also include newer work that directly tries to tackle the same problems addressed in our previous work but for several reasons, including but not limited to time constraint and later publication, were not included in our original analysis. We compared our work to current state-of-the-art algorithms at the time of publication of this dissertation to ensure the validity and relevance of out algorithms still hold.

We have also condensed and summarized the work done in our geometric based approach inspired in tessellation from [25] and [17] to a single chapter. Mostly based in the 2017 paper, chapter 3 consolidates our findings and algorithms developed for our tessellation based methodology to finding a usable banking scheme for stencil computations that ensures a conflict-free parallel memory access. The chapter not only contains the most recent results in terms of memory waste, resource utilization, and clock period, but also contains the algorithm to automatically find the optimal block size based in loop unrolling and more detailed explanations of the algorithm.

One of the major contributions of this dissertation is the additional analysis performed in our

graph-based approach for obtaining optimal banking schemes for stencil computations. First, using transitive edge reductions, an analysis technique applied to the original ESG, we were able to naively include data reuse analysis to the algorithm. Data reuse is a very popular and common technique used to improve the performance of memory subsystems and better utilize the available bandwidth. Incorporating this analysis we are able to reduce the partition factor even below the number of accesses, which for stencil kernels without data reuse it is considered optimal. We have also expanded the analysis of the algorithms applicable to our methods and we have developed a methodology based around defective coloring that can again, natively model the behavior of multi-port memories. Allowing for memories with more than one port can reduce the number of required memory banks for conflict-free parallel memory access since some of the simultaneous accesses now do not have to be handled exclusively by placing memory locations in different banks but also simply by using a different port in the same bank. Together, data reuse and defective coloring can reduce the partition factor required for a valid banking scheme significantly, reducing th complexity fo the interconnect and thus reducing resource utilization and improving performance.

Lastly we improved our testing emthodlogy and revised the results obtained for our approaches to solve the memory banking for non-stencil computations in both our graph based approach and our non-linear transformation aprproach.

Future Work

Despite being able to find the optimal partition factor for stencil computations, and even going beyond that when including data reuse analysis, with our graph-based approach, several work such as [35] and [32] focus instead on generating whole micro-architectures that fully utilize the information available at compile time and the regularity not only in access shape but memory access pattern itself to generate higly efficient memory subsystems that exploit data reuse opportunities.

In [35] they are capable of using just a singe bank, at the cost of long reuse buffers proportional to the problem size, to do the conflict-free memory partitioning for any stencil. And while [32] circumvents this problem by using more banks, the number of banks depends on the geometry of the stencil. One line of research is to try to obtain the benefits of both approaches, finding a micro-architecture that with a constant number of banks, it can do fully parallel conflict-free memory banking of any stencil computation, keeping the buffer sizes only proportional to the stencil size and not the problem size. To do this, one can take advantage of techniques widely used in the compiler work where one can take advantage of certain properties of stencil computation to re-order the iteration order, improving both temporal and spatial data locality.

On the other hand, although effective in some cases, our non-linear transformation approach to finding the optimal partition scheme for non-stencil code is very limited and only solves the problem in a small sub-set of affine forms non-stencil code can take. Because of this another line of research possible to explore is to use the knowledge gained form our graph theory-based approaches for stencils and non-stencils to analyze the conflict graph of any non-stencil kernels, particularly those generated by affine accesses since we have additional information at compile time, and find a way to convert or map it to a stencil generated graph which based on our previous work we know how to find the optimal partition scheme,

LIST OF REFERENCES

[1] J. Escobedo and M. Lin, "Graph-Theoretically Optimal Memory Banking for Stencil-Based Computing Kernels," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, (New York, NY, USA), pp. 199–208, ACM, 2018.

[2] J. Escobedo and M. Lin, "Extracting Data Parallelism in Non-stencil Kernel Computing by Optimally Coloring Folded Memory Conflict Graph," in *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, (New York, NY, USA), pp. 156:1—-156:6, ACM, 2018.

[3] Y. Wang, P. Li, and J. Cong, "Theory and Algorithm for Generalized Memory Partitioning in High-level Synthesis," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pp. 199–208, 2014.

[4] C. Carvalho, "The gap between processor and memory speeds," *Icca*, pp. 27–34, 2002.

[5] P. V. Sandt, Y. Chronis, and J. M. Patel, "Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?," p. 18.

[6] B. da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi, "Performance Modeling for FPGAs: Extending the Roofline Model with High-level Synthesis Tools," *Int. J. Reconfig. Comput.*, vol. 2013, pp. 7:7—-7:7, jan 2013.

[7] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, pp. 65–76, apr 2009.

[8] R. Fisher, S. Perkins, A. Walker, and E. Wolfart, "Feature detectors - Sobel edge detector," 2003.

[9] H. D. Shapiro, "Theoretical Limitations on the Efficent Use of Parallel Memories," *IEEE Transactions on Computers*, vol. c-27, pp. 421–428, may 1978.

[10] H. A. G. Wijshoff and J. V. Leeuwen, "Arbitrary versus Periodic Storage Schemes and Tessellations of the Plane Using One Type of Polyomino," *Information and Control*, vol. 62, pp. 1–25, 1984.

[11] H. A. G. Wijshoff, J. Van Leeuwen, and J. V. Leeuwen, "On Linear Skewing Schemes and d-Ordered Vectors," *IEEE Transactions on Computers*, vol. C-36, no. 2, pp. 233–239, 1987.

[12] H. A. G. Wijshoff, *Data Organization in Parallel Computers*. Springer US, 1989.

[13] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization," in *IEEE/ACM International Conference on Computer-Aided Design Digest of Technical Papers*, pp. 697–704, jun 2009.

[14] C. Meng, S. Yin, P. Ouyang, L. Liu, and S. Wei, "Efficient memory partitioning for parallel data access in multidimensional arrays," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, (New York, New York, USA), pp. 1–6, ACM Press, jun 2015.

[15] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, vol. 54, pp. 1242–1257, oct 2005.

[16] A. Cilardo and L. Gallo, "Improving Multibank Memory Access Parallelism with Lattice-Based Partitioning," *ACM Trans. Archit. Code Optim.*, vol. 11, pp. 45:1—-45:25, jan 2015.

[17] J. Escobedo and M. Lin, "Tessellating Memory Space for Parallel Access," in *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, vol. 1, pp. 75–80, IEEE, jan 2017.

[18] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang, "A New Approach to Automatic Memory Banking Using Trace-Based Address Mining," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, (New York, NY, USA), pp. 179–188, ACM, 2017.

[19] Y. Zou and M. Lin, "Graph-morphing: Exploiting hidden parallelism of non-stencil computation in high-level synthesis," in *Proceedings - Design Automation Conference*, (New York, New York, USA), pp. 1–6, ACM Press, 2019.

[20] J. Escobedo and M. Lin, "Exploiting Irregular Memory Parallelism in Quasi-Stencils through Nonlinear Transformation," in *The 27th IEEE International Symposium On Field-Programmable Custom Computing Machines*, pp. 236–244, IEEE, apr 2019.

[21] P. Zhang, P. Sadayappan, J. Cong, L. Angeles, L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based Data Reuse Optimization for Configurable Computing," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, (New York, NY, USA), pp. 29–38, ACM, 2013.

[22] M. Peemen, B. Mesman, and H. Corporaal, "Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 169–174, mar 2015.

[23] M. Milford and J. McAllister, "Constructive Synthesis of Memory-Intensive Accelerators for FPGA From Nested Loop Kernels," *IEEE Transactions on Signal Processing*, vol. 64, pp. 4152–4165, aug 2016.

[24] J. Liu, J. Wickerson, and G. A. Constantinides, "Tile size selection for optimized memory reuse in high-level synthesis," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, sep 2017.

[25] J. Escobedo and M. Lin, "Tessellation-based multi-block memory mapping scheme for high-level synthesis with FPGA," in *Proceedings of the 2016 International Conference on Field-Programmable Technology, FPT 2016*, pp. 125–132, IEEE, dec 2017.

[26] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory Partitioning for Multidimensional Arrays in High-level Synthesis," in *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, (New York, NY, USA), pp. 12:1—-12:8, ACM, 2013.

[27] A. Wigderson, "Improving the Performance Guarantee for Approximate Graph Coloring," *Journal of the Association for Computing Machinery*, vol. 30, pp. 729–735, 1983.

[28] E. Scheinerman, "Matgraph." Online, mar 2008.

[29] C. Uiyyasathian and S. Saduakdee, "Perfect Glued Graphs at Complete Clones," *Journal of Mathematics Research*, vol. 1, no. 1, pp. 25–30, 2009.

[30] N. Biggs, *Algebraic Graph Theory*. Cambridge University Press, 2nd ed., 1993.

[31] J. Su, F. Yang, X. Zeng, D. Zhou, and J. Chen, "Efficient Memory Partitioning for Parallel Data Access in FPGA via Data Reuse," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, pp. 1674–1687, oct 2017.

[32] W. Li, F. Yang, H. Zhu, X. Zeng, and D. Zhou, "An Efficient Data Reuse Strategy for Multi-pattern Data Access," in *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, (New York, NY, USA), pp. 118:1—-118:8, ACM, 2018.

[33] N. Christofides, "An algorithm for the chromatic number of a graph," *The computer journal*, vol. 14 (1), pp. 38–39, 1971.

[34] S. Fitzpatrick and L. Meertens, "Soft, Real-Time, Distributed Graph Coloring using Decentralized, Synchronous, Stochastic, Iterative-Repair, Anytime Algorithms," tech. rep., Kestrel Institute Technical Report KES.U.01.05, 2001.

[35] J. Cong, P. Li, B. Xiao, and P. Zhang, "An Optimal Microarchitecture for Stencil Computation Acceleration Based on Nonuniform Partitioning of Data Reuse Buffers," *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, pp. 77:1—-77:6, 2016.