2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)

Exploiting Irregular Memory Parallelism in Quasi-Stencils through Nonlinear Transformation

Juan Escobedo
School of Electrical and
Computer Engineering
University of Central Florida
Orlando, Florida 32816

Mingjie Lin
School of Electrical and
Computer Engineering
University of Central Florida
Orlando, Florida 32816

***Abstract*—Non-stencil kernels with irregular memory accesses pose unique challenges to achieving high computing performance and hardware efficiency in high-level synthesis (HLS) of FPGA. We present a highly versatile and systematic approach to effectively synthesizing a special and important subset of non-stencil computing kernels, *quasi-stencils*, which possess the mathematical property that, if studied in a particular kind of high-dimensional space corresponding to the *prime factorization* space, the distance between the memory accesses during each kernel iteration becomes constant and such an irregular non-stencil can be considered as a stencil. This opens the door to exploiting a vast array of existing memory optimization algorithms, such as memory partitioning/banking and data reuse, originally designed for the standard stencil-based kernel computing, therefore offering totally new opportunity to effectively synthesizing irregular non-stencil kernels.**

We show the feasibility of our approach implementing our methodology in a KC705 Xilinx FPGA board and tested it with several custom code segments that meet the quasi-stencil requirement vs some of the state-of the art methods in memory partitioning. We achieve significant reduction in partition factor, and perhaps more importantly making it proportional to the number of memory accesses instead of depending on the problem size with the cost of some wasted space.

I. Introduction

High-Level Synthesis (HLS) has advanced significantly in compiling high-level "soft" programs into efficient register-transfer level (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 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.

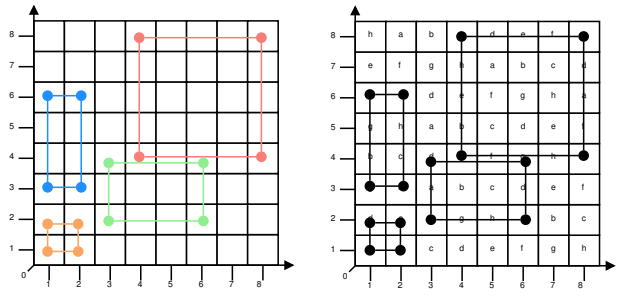
Many irregular scientific codes, unlike stencil-like computing kernel with static memory offsets, exhibit much more general and sophisticated memory access patterns, thus posing much greater challenges to achieving effective memory partitioning and mapping in order to facilitate parallel memory accesses. 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. Specifically, we claim the following contributions:

- We define a kind of non-stencil code we call *Quasi-Stencil* code that, under certain considerations, can have a memory partition factor independent of its problem size.
- We introduce a non-linear transformation of original data domain that allows a Quasi-Stencil code to behave just like a conventional stencil.
- We show a way of circumventing the problems associated with the naive implementation of our proposed nonlinear memory transformation in order to reduce its address calculation complexity and memory overhead.

II. Motivational example

It is well-known that the irregular memory access pattern found in non-stencil kernel computing renders the well-known hyperplane- [1], lattice- [2], or graph-based [3] HLS techniques almost totally ineffective. This is because that all

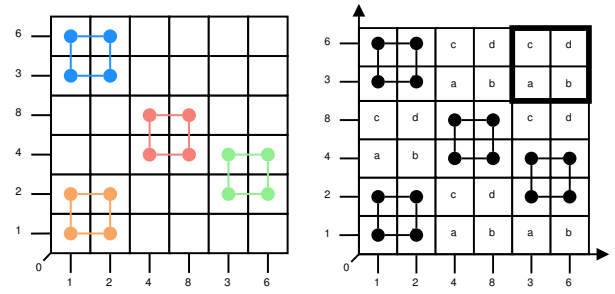
2576-2621/19/$31.00 ©2019 IEEE
DOI 10.1109/FCCM.2019.00039



(a) (b)

Fig. 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 [1]. Number of banks is proportional to the problem size

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. 1(a). If we directly apply the classical GMP memory banking scheme [1] to this irregular non-stencil case, as shown in Fig. 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

Fig. 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 [3]. Number of banks is independent of problem size. Work in [1] given the same number of banks but different layout.

Fortunately, as discussed in Section III-A, the motivational code segment shown in Fig. 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 III-A while Section III-C 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. 1 is shown in Fig. 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 [3], 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.

Finally, to further demonstrate the effectiveness of our methodology, we present one possible circuit implementation of our method in Fig. 3. In general, our hardware requirements are very similar to that of the traditional memory banking schemes for stencil computations [1]–[3] 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 III-F.

III. 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.

A. 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:

$$R_k = A_k \cdot \vec{i}, \tag{1}$$

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.

In order to intuitively visualize the behavior of a non-stencil kernel code, we develop a new diagram, where each memory

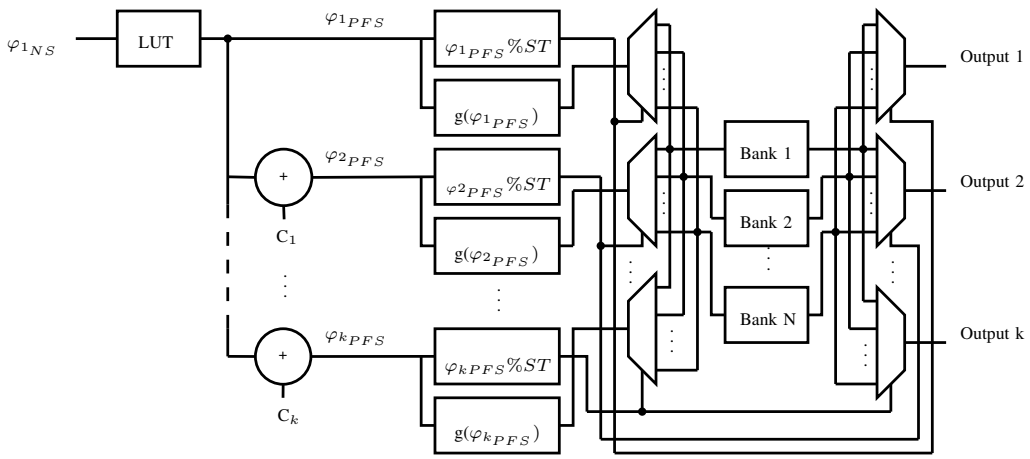


Fig. 3. Circuit diagram of our implementation. After our layer of indirection represented by a LUT, the circuit schematic remains the same as traditional banking schemes.

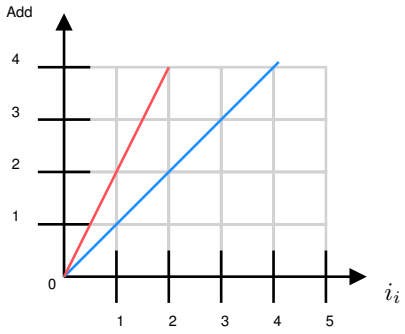


Fig. 4. Memory accesses formed by the lines $\varphi 1_1$:i(blue) and $\varphi 1_2$=2*i(red) in the plane ID,DD with as single intersection point at the origin.

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 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:

$$\varphi_i = A_{k_{i,i}} \cdot \vec{i}. \tag{2}$$

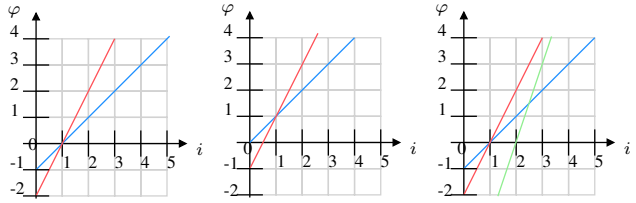
For example, in the code segment listed in Section II, two such line diagrams of the first two accesses $M(i, j)$ and $M(2 * i, j)$ along the first dimension are depicted in Fig. 4.

B. 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 accesses of the from in

$$R_k = A_k \cdot \vec{i} + b_k, \tag{3}$$

sill 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

Fig. 5. (a) Single intersect at $\varphi = 0$, $i = 0$. $\varphi 1_1 : i - 1$(blue) and $\varphi 1_2 = 2(i - 1)$ (red). (b) Single intersect at $\varphi = 0$, $i = 0$. $\varphi 1_1 : i$(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)

1) 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 stencil representation of the code. An example of this can be seen in figure 5 (a).

2) 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 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.

3) Multiple intersect points with integer time delay: There are cases such as the ones seen in Fig. 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.

C. 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 III-A and III-B 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.

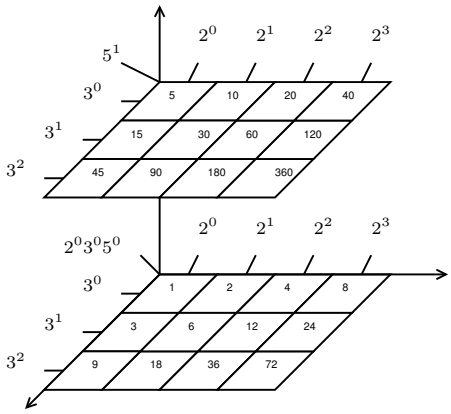


Fig. 6. Partial PFS for the first 3 primes 2, 3, and 5. We can represent numbers from 1-6 without gaps.

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 $n = 2^{k_1} \cdot 3^{k_2} \cdot 5^{k_3} \cdot \cdots \cdot p_n^{k_n} \cdots$. If we calculate the logarithm of n to 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. 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.

D. 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, we can obtain a linearized PFS depicted in Fig. 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. 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. 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.

E. 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

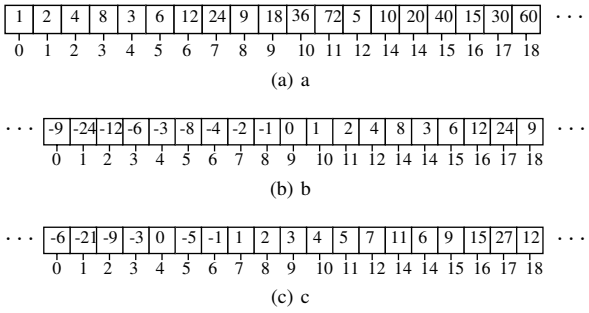


Fig. 7. (a) Linearized PFS for the case where memory locations intersect at the origin. (b) Extended linearized PFS for the case where memory locations intersect at $(x, 0)$, $x > 0$. (c) Shifted linearized space for the case where memory locations at (x, y), $x, y > 0$. For this example $y = 3$.

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 II. The full linearized PFS in both dimensions would look like the one from figure 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$5 either because of the nature of the code or they are artificial memory addresses generated when we linearize the PFS row/column wise. A data domain that was originally only 10$\times$10, for a total of 100 data elements now consist of a 2D grid of 17$\times$17 elements for a total of 289 elements. And increase in the total number of elements close to 200%.

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.

F. 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 [illegible] any of the available stencil banking schemes existing in literature. In this work we have decided to implement the method from [3] as our banking scheme given the proven optimality results for any stencil in terms of partition factor

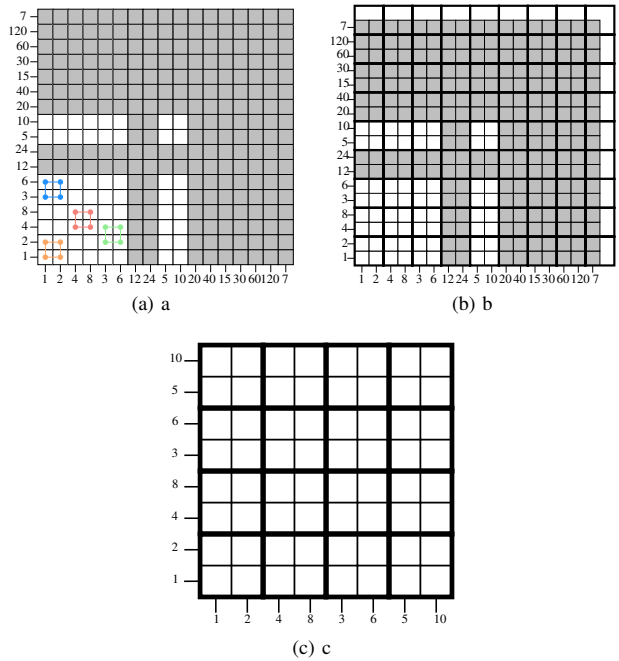


Fig. 8. (a) Full Linearized PFS for the motivational example. Gray cells are memory locations that are never accessed for i,j$\leq$5. (b) Full Linearized PFS with overlapped region of repeated banking. (c) Pruned linearized PFS

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.

$$\mathrm{B} = \mathrm{Off}_{\mathrm{ST}}(\varphi' \% \mathrm{ST}) \tag{4}$$

The intra bank address φ_{B} can be calculated from the transformed memory address φ' by the formula seen in Eq. 5:

$$\varphi_{\mathrm{B}} = \mathrm{Off}_{\mathrm{ST}}(\varphi' \% \mathrm{ST}) + \sum_{i=0}^{n} (\varphi'_i / M_i) \cdot k_i \tag{5}$$

As mentioned in section III-E, 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

```
for(i=1;i<n;i++)
  S=f(M[i],M[2*i],M[5*i]);
(a)
for(i=1;i<n;i++)
  S=f(M[i],M[2*i],M[4*i]);
(d)
for(i=1;i<n;i++)
  for(j=1;j<m;j++)
    S=f(M[i,j],M[2*i,j],...
      M[i,2*j],M[2*i,2*j]);
(g)
```

```
for(i=1;i<n;i++)
  S=f(M[i],M[3*i],M[5*i]);
(b)
for(i=1;i<n;i++)
  S=f(M[i-1],M[3*i-4],M[4*i-6]);
(e)
for(i=1;i<n;i++)
  for(j=1;j<m;j++)
    S=f(M[i,j],M[2*i-1,j],...
      M[i,2*j-1],M[2*i-1,2*j-1]);
(h)
```

```
for(i=1;i<n;i++)
  S=f(M[i+4],M[2*i+4],M[3*i+4]]);
(c)
for(i=1;i<n;i++)
  S=f(M[2*i-2],M[3*i-7],M[4*i-6]);
(f)
for(i=1;i<n;i++)
  for(j=1;j<m;j++)
    S=f(M[i-1,3*j-6],M[2*i-2,j-1],...
      M[3*i-6,2*j-2]]);
(i)
```

Fig. 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

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.

IV. 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 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.

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 [3] 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 to reduce the memory overhead. From the pruned PFS we can obtain our LUT with the address for the translation 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 I.

TABLE I
PARTITION FACTOR, MEMORY OVERHEAD, CLOCK PERIOD, AND RESOURCE UTILIZATION FOR ALL THE TEST CASES FOR DIFFERENT PROBLEM SIZES.

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 [1] to obtain the partition factor that state-of-the-art methods would yield (see Table II). 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

TABLE II
PARTITION FACTOR FOR OUR METHOD VS GMP FOR ALL CONSIDERED PROBLEM SIZES. TEST CASES (A)-(F) 20X1,40X1,80X1. (G)-(H) 1080X1080. (I) 100X100.

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

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 [1] for the 2D test kernels, which corresponds to more realistic applications.

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 to 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.

V. RELATED WORK

Extensive research has been performed to achieve optimal memory banking and data reuse for regular stencil-based kernel code. The classical method in [1] uses a single family of hyperplanes to partition the data. Its main limitation is that it only considers memory blocks in one dimension and the linear nature of the hyperplane families limits the possible solution space which sometimes leads to a suboptimal partition factor. On the other hand, the method from [4] tessellates the memory space and finds the smallest rectangle that contains a repeating pattern to do the banking. This methods offers the advantage that uses as many families of hyperplanes as the dimensionality of the space that allows it to find better solutions. The main limitation is that an upper bound to the partition factor is not given. The advantage of using more than one family of hyperplanes is also proven in [5] and later also in [2] where the authors assume the number of desired banks to be known and use that to find the basis of a lattice of the same rank as the original problem. They are also able to achieve arbitrarily small memory waste at the cost of increasingly complex memory address calculations. The main limitation of this method is that it is not guaranteed to find the optimal solution and would require an undetermined number of runs to find it, by doing exhaustive exploration of the solution space. Graph based approaches such as the one presented in [6] would still not be able to handle this kind of non-stencil kernels. In that work, the authors analyze the trace of the memory access pattern and tries to find a binary mask that maps the addresses to a smaller subset. The main idea is that this mapped addresses are considered nodes in a conflict graph and all masked addresses that are accessed together in any iteration are joined by and edge. This graph is colored and exploring all possible masks, with a fixed number of set bits, one is able to find for stencil kernels the one that maps to a graph with the smalls chromatic number ensuring zero conflicts. Although this method is general enough to be used in non-stencil kernels, 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. More recently in [3], unlike previous approaches, the authors propose a graph-based approach to find the optimal partition factor natively. Here they authors construct a special data structure called ESG, a spacial 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. One work that tries to tackle this problem is [7], where the author propose a best-effort approach that tries to average out the conflict graph to a manageable size by overlapping regions of it and colors 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.

An approach that tries to take advantage of the expressiveness of the polyhedral framework is explored in [8]. 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 [9] 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 [10]. In this work the authors take advantage of he cyclo-static representation 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 [11] 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.

VI. Conclusions and future work

Given any unstructured computing kernel with irregular memory accesses, if it satisfies the requirement of being a quasi-stencil, 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.

References

[1] 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.

[2] 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, 2015.

[3] 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.
[4] E. Juan and M. Lin, "Tessellating memory space for parallel access," in *ASP-DAC*, 2017.
[5] A. Darte, I. C. S. Member, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, vol. 10, 2005.
[6] 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.
[7] 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.
[8] 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.
[9] 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, March 2015.
[10] 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.
[11] 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, Sept 2017.