An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers

Jason Cong*, Peng Li , Bingjun Xiao*, and Peng Zhang

Computer Science Dept. & Electrical Engineering Dept.

University of California, Los Angeles

*{cong, xiao}@cs.ucla.edu

ABSTRACT

High-level synthesis (HLS) tools have made significant progress in compiling high-level descriptions of computation into highly pipelined register-transfer level (RTL) specifications. The high-throughput computation raises a high data demand. To prevent data accesses from being the bottleneck, on-chip memories are used as data reuse buffers to reduce off-chip accesses. Also memory partitioning is explored to increase the memory bandwidth by scheduling multiple simultaneous memory accesses to different memory banks. Prior work on memory partitioning of data reuse buffers is limited to uniform partitioning. In this paper, we perform an early-stage exploration of non-uniform memory partitioning. We use the stencil computation, a popular communication-intensive application domain, as a case study to show the potential benefits of non-uniform memory partitioning. Our novel method can always achieve the minimum memory size and the minimum number of memory banks, which cannot be guaranteed in any prior work. We develop a generalized microarchitecture to decouple stencil accesses from computation, and an automated design flow to integrate our microarchitecture with the HLS-generated computation kernel for a complete accelerator.

1. INTRODUCTION

Accelerator-centric architectures can bring 10-100x energy efficiency by offloading computation from general-purpose CPU cores to application-specific accelerators [1]. The engineering cost of designing massive heterogeneous accelerators is high, but can be much reduced by raising their abstraction level beyond RTL to C by high-level synthesis (HLS) [2]. Data access optimization has a strong impact on HLS results. This significantly motivates recent work on data reuse [3,4] and memory partitioning [5–9] in HLS.

External memory bandwidth is a significant bottleneck for system performance and power consumption. Data reuse is an efficient technique of using on-chip memories to reduce external memory accesses. When an application contains a data array with multiple references, we can allocate a reuse buffer and keep each array element in the buffer from its first access until its last access. Then each array element needs to be fetched from the external memory only once, and the off-chip traffic is reduced to the minimum. Loop transformation can be applied to improve data locality and reduce the size of the data reuse buffer [3,4].

When the innermost loop of an application is fully pipelined, an accelerator needs to perform multiple load operations from the same reuse buffer every clock cycle. To avoid contention on memory ports, memory partitioning of the reuse buffer is required. Since the transistor count of memory control logics is proportional to the number of memory banks after partitioning, the optimization goal of memory partitioning is to minimize the number of memory banks. The constraint is that the multiple array elements to be loaded every clock cycle are always stored in different memory banks. The work in [5,6] provides solid frameworks of memory partitioning. Further optimizations, including memory access rescheduling [7] and multi-dimension arrays [8], are also proposed. But none of them can guarantee the optimal solution for a given case in terms of the number of banks. The reason is that their optimization space is limited to uniform memory partitioning, i.e., all the memory banks have to be of the same size. It is an unnecessary constraint which was assumed by commodity HLS tools, e.g., [10]. Other work partitions different fields of a single data *structure* into multiple memory banks for data parallelism based on profiling results [9]. It is orthogonal to the problems on multiple array references in [5–8] and this paper.

In this paper, we go beyond the limitation of uniform memory partitioning, and propose a novel method based on non-uniform memory partitioning. As a result, we can achieve fewer memory banks than the optimal solutions in prior work [5–8] which were limited to uniform memory partitioning. As an early-stage exploration of non-uniform memory partitioning, in this paper we focus on stencil computation, a popular communication-intensive application domain. We develop a microarchitecture with novel structures of memory systems which achieve the theoretical minimum number of memory banks for any stencil access patterns. Experimental results show that we can reduce $25-100\%$ of various resources including BRAMs, logic slices, and DSPs compared to prior work [8], along with slightly improved timing.

2. PRELIMINARY

2.1 Stencil Computation

Stencil computation comprises an important class of kernels in many application domains, such as image processing, constituent kernels in multigrid methods, and partial differential equation solvers. These kernels often contribute to most workloads in these applications. Even in the recent publications on memory partitioning [7,8] which were developed for general applications, all the benchmarks used in the publications are in fact stencil computation.

The data elements accessed in stencil computation are on a large multi-dimensional grid which usually exceeds on-chip memory capacity. The computation is iterated as a stencil window slides over the grid. In each iteration, the computation kernel accesses all the data points in the stencil window to calculate an output. Both the grid shape and the stencil window can be arbitrary as specified by the given stencil applications. A precise definition of stencil computation can be found in [11,12].

```
void denoise2D( float A[768][1024],
                float B[768][1024] )
{
    for( int i = 1; i < 767; i++ )
        for( int j = 1; j < 1023; j++ )
            B[i][j] =
                pow(A[i][j] - A[i][j-1], 2) +
                pow(A[i][j] - A[i][j+1], 2) +
                pow(A[i][j] - A[i-1][j], 2) +
                pow(A[i][j] - A[i+1][j], 2);
}
```

Listing 1: Example C code of a typical stencil computation (5-point stencil window in the kernel 'DENOISE' in medical imaging [13]).

Listing 1 shows an example stencil computation in the kernel 'DENOISE' in medical imaging [13]. Its grid shape is a 768 × 1024

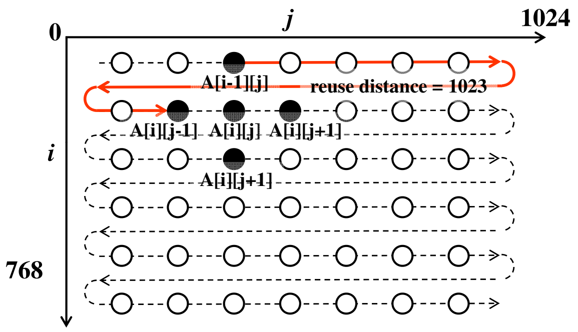

Figure 1: Iteration domain of the example stencil computation in Listing 1.

rectangle, and its stencil window contains 5 points, as shown in Fig. 1. Five data elements need to be accessed in each iteration. In addition, many data elements will be repeatedly accessed among these iterations. For example, $A[2][2]$ will be accessed five times, when $(i, j) \in \{(1, 2), (2, 1), (2, 2), (2, 3), (3, 2)\}$. This leads to high on-chip memory port contention and off-chip traffic, especially when the stencil window is large (e.g., after loop fusion of stencil applications for computation reduction as proposed in [14]). Therefore, during the hardware development of a stencil application, a large portion of engineering effort is spent on data reuse and memory partitioning optimization.

2.2 Microarchitecture for Stencil Accesses

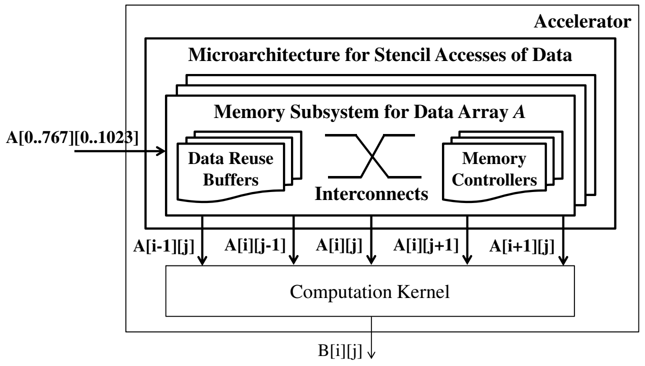

Figure 2: The overall architecture of our microarchitecture for stencil computation. It decouples stencil accesses from computation.

In this work we develop a microarchitecture to decouple the stencil access patterns from the computation, as shown in Fig. 2. The microarchitecture contains multiple memory systems, and each is optimized to a data array with stencil accesses. Since there are no reuse opportunities among different data arrays, the memory systems for different arrays are independent of each other. Each memory system receives a single data stream which iterates on a multi-dimensional grid without any repeated external access. Each memory system contains data reuse buffers, memory controllers and interconnects that have been customized for the access patterns of the data array used in the target stencil computation. A memory system sends data to the computation kernel via each data port associated with each array reference in the original user code. If all the data are consumed by the computation kernel, the memory system will immediately prepare the data used in the next cycle to feed into the fully pipelined computation kernel.

```
void denoise2D_kernel( volatile float* a0_ptr,
                       volatile float* a1_ptr,
                       volatile float* a2_ptr,
                       volatile float* a3_ptr,
                       volatile float* a4_ptr,
                       volatile float* b_ptr )
{
    for( int i = 1; i < 767; i++ )
        for( int j = 1; j < 1023; j++ )
        {
#pragma AP pipeline II=1
            float a0 = *a0_ptr;
            *b_ptr =
                pow(a0 - *a1_ptr, 2) +
                pow(a0 - *a2_ptr, 2) +
                pow(a0 - *a3_ptr, 2) +
                pow(a0 - *a4_ptr, 2);
        }
}
```

Listing 2: Example code of the computation kernel where all the memory accesses are offloaded to our microarchitecture. The keyword 'volatile' in the code informs HLS tools of potential data change after access. The pragma 'pipeline is used in Xilinx Vivado HLS [10] to pipeline the innermost loop.

With our microarchitecture, the C code of the computation kernel can be simplified to Listing 2, where users no longer need to optimize memory accesses, which is offloaded to our microarchitecture. Users can assume that each data access to the points should get the same data from our accelerator microarchitecture as the original load operation. The C code of the computation kernel can be compiled by HLS tools for a fully pipelined hardware implementation with the most efficient resource usage.

2.3 Design Objectives

We have three design objectives for the proposed microarchitecture:

1. *Full pipelining*. As the stencil window slides every clock cycle, the microarchitecture is able to send out all the data in the stencil window to the computation kernel and get all the data ready for the consecutive accesses in the next cycle.

2. *Minimum data reuse buffer size*. When a data element is fully reused for each access, it stays in on-chip memories from its first access until its last access. Meanwhile, other elements in the array are loaded from external memory every clock cycle. Therefore, the theoretical minimum size of the reuse buffer for a data array is equal to the maximum lifetime of any element in the array. In the example in Fig. 1, $A[2][2]$ is accessed now by the array reference $A[i + 1][j]$ for the first time, and is accessed 2048 cycles later by the reference $A[i - 1][j]$ for the last time. Therefore, the minimum size of the data reuse buffer for array A will be 2048.

3. *Minimum number of reuse buffer banks*. Suppose the stencil window of an input array contains n points, i.e., there are n data references to the array. It means that each clock cycle, n data elements need to be read out, either from reuse buffers or from the external memory. Suppose we use dual-port memories to implement reuse buffers. One port of a buffer bank is occupied by the replacement of an expired data element with a new element from the external memory every cycle. There is only one port left in each memory bank for us to read the n elements needed by the stencil window. Suppose one of the n element happens to be the new element from the external memory in a certain smart data reuse mechanism. There are still $n - 1$ data elements to read, and we would need at least $n - 1$ memory banks. In the example of Fig. 1, $n = 5$ indicates that we need at least four memory banks. As the stencil window slides, all five data elements in the window should never be in the same bank. This is a tough constraint to satisfy, and prior

work [5–8] had to use more banks to eliminate bank conflicts in difficult cases. Fig. 3 shows that the number of banks ranges from

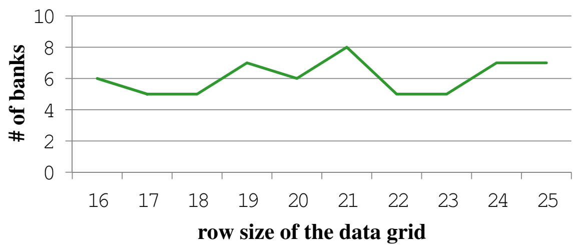

Figure 3: Even for a constant stencil window in Fig. 1, the number of banks varies as the row size of the data grid changes [5].

five to eight in [5] as the row size of the data grid changes, even if the stencil window keeps the constant shape in Fig. 1. The technologies proposed in [7, 8] can keep the number of banks consistently to be five in the case of the stencil window shown in Fig. 1. However when the stencil window changes to some other shape

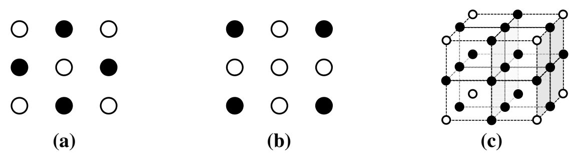

Figure 4: Example stencil windows where more banks are needed than the # of array references in [7, 8]. (a) 4-point stencil in 'BICUBIC' [15]. (b) 4-point stencil in 'RICIAN' [16]. (c) 19-point stencil in 'SEGMENTATION_3D' [8].

in other applications, e.g., the ones shown in Fig. 4, the methods in [7, 8] will need 5, 5, 20 banks respectively, which are larger than the minimum values.

In this work we will present a generalized microarchitecture that can simultaneously achieve these optimal design objectives for any application that falls in the category of stencil computation.

3. METHODOLOGY

3.1 Overview

The internal structure of a memory system in our microarchitecture is illustrated by the example in Fig. 5, which is generated for the stencil computation in Listing 1. Suppose the stencil window contains n points ($n = 5$ in the example of Listing 1). Our memory system will contain $n - 1$ data reuse FIFOs as well as n data path splitters and n data filters connected together in the way shown in Fig. 5. The data reuse FIFOs provide the same storage as conventional data reuse buffers, and the data path splitters and filters work as memory controllers and data interconnects. In contrast to conventional cyclic memory partitioning [5, 7, 8] which uses uniform buffer sizes, the sizes of reuse buffers in our design are nonuniform. They are customized to the shape of the stencil window in the target application.

3.2 Denotations

To better explain the working principle of our memory system, we provide a table of denotations that will be used in the following sections in Table 1. The precise definitions of the denotations are given in [11].

3.3 Working Principle

Since our microarchitecture is based on nonuniform memory partitioning in contrast to uniform partitioning in prior work, the memory controlling mechanism cannot follow the modulo scheduling of data accesses among memory banks in prior work [5, 7, 8]. Instead, our microarchitecture is a novel design based on data streaming. Each module in our design is autonomous and can work in a full pipeline as long as its upstream module produces a data element and its downstream module consumes an element every clock

Denotations	Meanings	Example by Fig. 1
$\vec{i}$	loop iteration vector	$\vec{i} = (1, 2)$
A	data array	the array A with the five references
A_x	array reference	the five references such as $A[i+1][j]$
$\vec{h}$	data access index	$\vec{h}_x = (2, 2)$ being accessed by A_x
$\vec{f}$	data access offset	$\vec{f}_x = \vec{h}_x - \vec{i} = (1, 0)$ constant for A_x
$\mathcal{D}_{A_x}$	data domain	$\{(i, j) \mid 2 \le i \le 767, 1 \le j \le 1022\}$
$\mathcal{D}_A$	input data domain	$\{(i, j) \mid 0 \le i \le 767, 0 \le j \le 1023\}$
$\vec{r}_{A_x \leftarrow A_y}$	reuse distance vector	(1,-1) from $A[i+1][j]$ to $A[i][j+1]$
$\lVert\vec{r}_{A_x \leftarrow A_z}\rVert$	maximum reuse distance	2048 from $A[i+1][j]$ to $A[i-1][j]$
$\succ_l$	lexicographic order	$(1, 0) \succ_l (0, 1) \succ_l (0, 0) \succ_l (-1, 0)$

Table 1: Denotations used in the working principle of our memory system.

cycle. As we shall discuss next, our design achieves function correctness, data streaming without stalling, the minimum reuse buffer size, and the minimum number of buffer banks.

3.3.1 *Function Correctness*

Each data path splitter in Fig. 5 reads any existing data element from its precedent FIFO and sends the data element to the successive FIFO as well as to the data filter below. Each data filter customizes the data stream that flows into the computation kernel to fit the access patterns of the associated array reference. A data filter for an array reference A_x receives the data stream which iterates in $\mathcal{D}_A$ and sends out the data which iterates in $\mathcal{D}_{A_x}$. For example, filter 0 in Fig. 5 sends the data element in set $\mathcal{D}_{A_0} = \{(i, j) | 2 \le i \le 767, 1 \le j \le 1022\}$ out of the input data domain $\mathcal{D}_A$ in Table 1 and discards the first two rows in the 2D grid of Fig. 1. This guarantees the correctness of the data set sent to each data port of the computation kernel. Due to the property of stencil computation, the data elements accessed by an array reference are in the same lexicographic order as the loop iteration (see details in [11]). Our microarchitecture based on data streaming enforces this order, as long as the input data stream is also in the lexicographic order (i.e., data iterated from innermost loop to outermost loop). The lexicographic order of input data is usually realized without hardware overhead since it fits well with burst accesses to external memory or inter-accelerator communication patterns (see discussion in [11]). By providing the correct data set and correct data order to the array references, our design can guarantee that as the stencil window slides, the data elements received by the computation kernel are always consistent with the array references.

3.3.2 *Data Streaming without Stalling*

One key challenge is to ensure that the data streaming structure will not be stalled due to any FIFO emptiness or fullness. To prevent FIFO emptiness, we can sort the data access offsets $\vec{f}$ of the data array references in the descending lexicographic order when we map them to data filters from 0 to $n-1$, e.g., $(1, 0) \to (0, 1) \to (0, 0) \to (0, -1) \to (-1, 0)$ in Fig. 5. To prevent FIFO fullness,

FIFO ID	precedent/successive references	FIFO size	physical impl.
FIFO 0	$A[i+1][j] \to A[i][j+1]$	1023	BRAM
FIFO 1	$A[i][j+1] \to A[i][j]$	1	register
FIFO 2	$A[i][j] \to A[i][j-1]$	1	register
FIFO 3	$A[i][j-1] \to A[i-1][j]$	1023	BRAM

Table 2: Reuse FIFOs with nonuniform sizes calculated from maximum reuse distances of adjacent array references and mapped to different physical implementations (block memory, distributed memory, or register) if targeted an FPGA platform.

we calculate the maximum reuse distances of all the pairs of adjacent array references and allocate reuse FIFO sizes accordingly, as shown in Table 2 for the example in Listing 1. Detailed can be found in [11].

3.3.3 *Design Optimality*

Minimum Reuse Buffer Size. Due to the linearity of maximum reuse distances, the sum of the sizes of all the reuse FIFOs is equal to the maximum reuse distance between array reference A_0 and A_{n-1}. Due to the sorting of array references by $\vec{f}$ in the descending lexicographic order, A_0 is the earliest reference and A_{n-1} is the

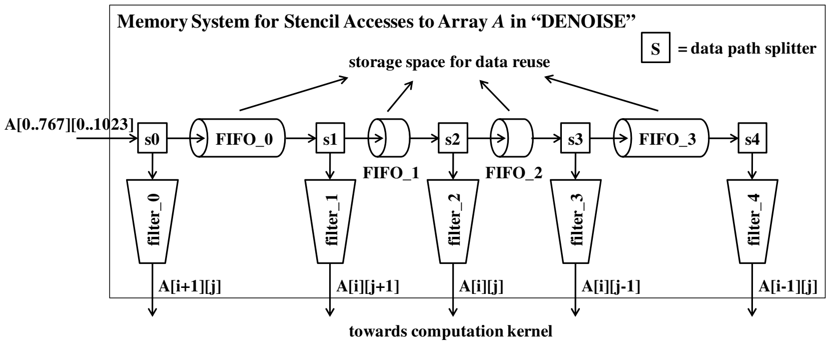

Figure 5: The example circuit structure of our memory system generated for array A in the stencil computation of Listing 1.

latest reference. Therefore, the total reuse buffer size is equal to the maximum reuse distance between the earliest and latest references, which is the theoretical minimum. As shown in Table 2 for the example in Listing 1, the total size is 2048, the same as the minimum value discussed in Section 2.3. If the maximum reuse distance is so large that the buffer sizes exceed the on-chip memory capacity, our microarchitecture allows tradeoff of offchip bandwidth occupation for smaller memory usage (see discussion in [11]).

Minimum Number of Buffer Banks. The structure of our design guarantees that for n array references, there are $n-1$ buffer banks (reuse FIFOs) as described in Section 3.1. It is the theoretical minimum value as discussed in Section 2.3.

Since our design achieves both the minimum reuse buffer size and the minimum number of buffer banks, our microarchitecture is optimal.

3.4 Insights Gained From RTL Simulation

Our microarchitecture is quite different from conventional designs with centralized controllers [5–8]. The major tasks of a conventional controller includes two aspects:

1. *Filling up reuse buffers.* Before the computation starts, the controller will first fill reuse buffers with data elements needed by the computation kernel.
2. *Evict expired data from reuse buffers.* The challenging part in this function is when the reuse distance between array references changes as the execution advances. This often happens on a skewed data grid. In this case, the number of data elements stored in reuse buffers changes as time goes, and the symmetry between read and write is broken.

There is no specific module that takes charge of these key tasks in our microarchitecture. Instead, by observing the execution of our design in RTL simulation, we found that these key tasks are done automatically by the coordination of our distributed modules.

3.4.1 Automatic Filling of Reuse Buffers

The filling process of reuse buffers in our microarchitecture is shown in Table 3. The data filter 4 associated with $A[i-1][j]$ is first stalled at cycle 1. This is when the filter 4 tries to send data $A[0][1]$ to the computation kernel but all the other data filters are bypassing this data. As a result, the computation kernel will be waiting for data from the other data filters and will not consume data from the filter 4. This stalling will lead to filling up of data in the FIFO 3 between $A[i-1][j]$ and $A[i][j-1]$. The other four filters will keep the data stream advancing since all of them bypass the first row in the data domain. 1023 cycles later, the filter 3 will try to to send data $A[1][0]$ to the computation kernel but will be stalled as well. Then the FIFO 2 will start being filled up, and the data path splitter s3 will stop sending data to FIFO 3. The following process is similar until FIFO 1 and FIFO 0 are filled up consecutively, as shown in Table 3. Then at cycle 2049, the filter 0 receives $A[2][1]$ and will send the data to the computation kernel. All the data at the five inputs of the computation kernel become valid, and the kernel consumes all the five data to produce the first output. Then all the stalled filters can continue to send new data to the computation kernel every clock cycle until the end of the iteration domain.

3.4.2 Automatic Adjustment of Reuse Data Amount

An application can have a skewed data grid as shown in Fig. 6. This kind of application is usually needed when a rectangular grid

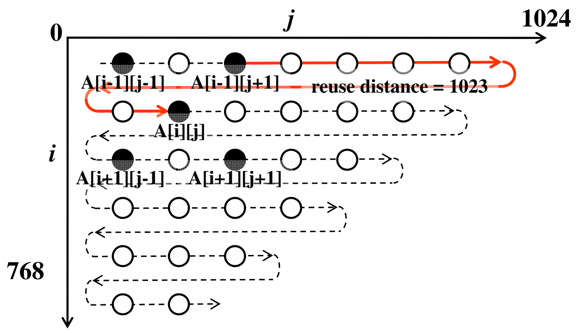

Figure 6: Non-rectangular iteration domain of an example application with dynamically changeable reuse distance.

is iterated along the 45° direction after certain loop transform [17]. The skewed grid leads to the challenge that the number of data stored in each reuse buffer will change as the iteration goes on, and often requires a complex memory controlling scheme in a centralized design [5, 7, 8]. However, this difficulty can be automatically handled by our distributed modules. Following the design schematic in Section 3.1, we will order the array references and map them to five filters 0–4. Among them, filter 2 is for reference $A[i][j]$ and filter 3 is for reference $A[i-1][j]$. Note that $\vec{h}_2$ of filter 2 advances $\vec{h}_3$ of filter 3 by one row when $\vec{h}_2$ and $\vec{h}_3$ are synchronized by the computation kernel, as shown in Fig. 6. At each turn around to the next row in the data domain, the filter 3 will fetch one more data from the FIFO splitter than the filter 2 since the filter 3 is iterating over a longer row. Then the number of data stored in FIFO 2 between filter 2 and filter 3 will be reduced by one. This achieves the dynamic adaption of the number of data stored in a reuse buffer to the change of reuse distance in the case of a skewed data grid.

3.5 Miscellaneous Design Issues

3.5.1 Heterogeneous Mapping of Reuse Buffers

Reuse buffers in our design have different sizes, as shown in Table 2, and may prefer different physical implementations. For example, if the target platform is FPGA, the physical implementation candidates include block memory, distributed memory and slice registers. They are efficient for a large buffer, a medium buffer and a small buffer respectively. Table 2 shows the heterogeneous mapping of reuse buffers to different physical implementation.

3.5.2 Data Filter in Polyhedral Domain

Note that though the data domains are rectangles in the example of Listing 1, they could be any polyhedrons on a multi-dimensional grid. Comparisons on loop bounds is not a universal solution to data filtering. To select $\mathcal{D}_{A_x}$ out of $\mathcal{D}_A$, the data filter in our microarchitecture is implemented with a data switch controlled by two

clock cycle	data in stream	filter status (forwarding/bypassing/stalled)					FIFO status (# of data)			
		filter 0	filter 1	filter 2	filter 3	filter 4	FIFO 0	FIFO 1	FIFO 2	FIFO 3
1	$A[0][1]$	bypass	bypass	bypass	bypass	forward→stall	0	0	0	0
1024	$A[1][0]$	bypass	bypass	bypass	forward→stall	stall	0	0	0	1023
1025	$A[1][1]$	bypass	bypass	forward→stall	stall	stall	0	0	1	1023
1025	$A[1][2]$	bypass	forward→stall	stall	stall	stall	0	1	1	1023
2028	$A[2][1]$	bypass→forward	stall→forward	stall→forward	stall→forward	stall→forward	1023	1	1	1023
2049–...	$A[2][2]$–...	forward	forward	forward	forward	forward	1023	1	1	1023

Table 3: The execution flow of our microarchitecture in the example of Listing 1. The latency among the data streams at different modules is ignored here for demonstration purpose only.

counters, let's say input counter and output counter, as shown in Fig. 7. The input counter iterates over the input data stream $\mathcal{D}_A$ of

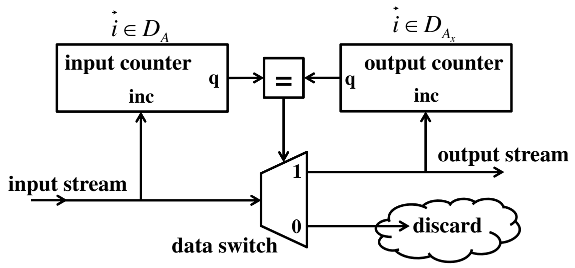

Figure 7: Structure of the data filter which can be applied to general polyhedral data domains.

array A, e.g., $A[0..767][0..1023]$. The output counter iterates over the data domain $\mathcal{D}_{A_x}$ of array reference A_x, e.g., $A[2..767][1..1022]$. The input counter proceeds when the filter receives an input data. The output counter proceeds when its value is equal to the input counter. It is also the condition that the data switch forwards the input data to the output. In contrast, when the output counter is not equal to the input counter, the data switch bypasses the input data.

4. DESIGN AUTOMATION FLOW

We develop a design automation flow to generate the complete accelerator for a given stencil application, as shown in Fig. 8. It

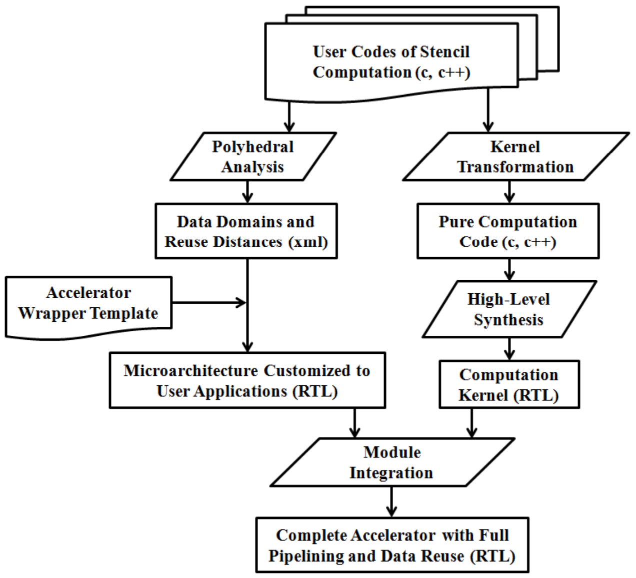
User Codes of Stencil Computation (c, c++)
Polyhedral Analysis
Kernel Transformation
Data Domains and Reuse Distances (xml)
Pure Computation Code (c, c++)
Accelerator Wrapper Template
High-Level Synthesis
Microarchitecture Customized to User Applications (RTL)
Computation Kernel (RTL)
Module Integration
Complete Accelerator with Full Pipelining and Data Reuse (RTL)

Figure 8: Design automation flow of accelerator generation for stencil computation.

starts from the original source codes of a user application, e.g., the code in Listing 1. In the left branch, we first apply polyhedral analysis to extract the polyhedrons of data arrays with stencil accesses. We calculate the data domain of each array reference and the reuse distance of each pair of adjacent array references. This information is used to instantiate the data filters and reuse FIFOs in our microarchitecture. Then the flow generates a microarchitecture instance, e.g., the design in Fig. 5, with the memory systems optimized for stencil accesses in user applications. In the right branch, we first apply source-to-source code transformation to extract the kernel code with pure computation, e.g., Listing 2. Then high-level synthesis is applied on the transformed code for a fully pipelined hardware implementation of the computation kernel in RTL. Finally, we integrate the microarchitecture with the computation kernel for a complete accelerator with full pipelining and data reuse, e.g., the design in Fig. 2.

5. EXPERIMENTS

5.1 Experiment Setup

Our polyhedral analysis in Fig. 8 is implemented by the LLVM-Polly framework [18]. The kernel transformation is performed by the open source compiler infrastructure ROSE [19], and the high-level synthesis is performed by Xilinx Vivado HLS [10]. Although our methodology is applicable to both ASIC and FPGA designs, we choose FPGA as the target device in this work due to the availability of downstream behavioral synthesis and implementation tools. The Xilinx Virtex7 FPGA XC7VX485T and ISE 14.2 tool suite [20] are used in our experiments. The target clock frequency is set at 200MHz.

The benchmarks used in prior memory partitioning work [7, 8] make up a rich set of real-life stencil computation kernels. Among them, we select the more challenging benchmarks with non-rectangular stencil windows for our experiments. DENOISE (2D/3D), RICIAN (2D), and SEGMENTATION (3D) are from medical imaging [13]. BICUBIC (2D) is from bicubic interpolation process [15]. SOBEL (2D) is from Sobel edge detection algorithm [16]. We choose the more recent memory partitioning work [8] as our experiment baseline.

5.2 Results

	Original *II*	Target *II*	# of Banks		Total Size	
			[8]	Ours	[8]	Ours
DENOISE	5	1	5	4	2050	2048
RICIAN	4	1	5	3	2050	2048
SOBEL	9	1	9	8	2054	2050
BICUBIC	4	1	5	3	2050	2048
DENOISE_3D	7	1	7	6	2240	2048
SEGMENTATION	19	1	20	18	2630	2112

Table 4: High-level partitioning results.

The comparison results of memory partitioning are shown in Table 4. We list the pipeline *II* of the original user codes which suffer memory port contentions before memory partitioning, which is equal to the number of memory load operations on the data array. We also list the *II* that the computation kernel targets to achieve via memory partitioning. The number and total size of reuse buffer banks are reported for both [8] and our method. The buffer size is in the unit of data element. As shown in Tabel 4, our method saves the partitioning bank number of all of the six benchmarks. In addition, our method does not need the padding technique in [8] which increases the grid size at certain dimensions to relax the partitioning complexity. Our methods saves the buffer size, especially when the padding introduces more overhead in a high-dimensional data grid, e.g., Fig. 4(c).

The post-synthesis results are listed in Table 5. Physical resource usage (block RAMs, logic slices, and DSPs) and timing informa-

		BRAM	Slice	DSP	CP (ns)
DENOISE	[8]	5	703	5	4.502
	ours	2	636	0	4.519
	comp.(%)	**-60**	**-9.5**	**-100**	0.37
RICIAN	[8]	5	582	4	4.472
	ours	2	544	0	4.337
	comp.(%)	**-60**	**-6.5**	**-100**	**-3.02**
SOBEL	[8]	9	1937	9	4.416
	ours	2	1088	0	4.239
	comp.(%)	**-78**	**-43.8**	**-100**	**-4.01**
BICUBIC	[8]	5	535	4	4.309
	ours	2	493	0	4.196
	comp.(%)	**-40**	**-7.8**	**-100**	**-2.62**
DENOISE_3D	[8]	7	980	7	4.656
	ours	2	859	0	4.762
	comp.(%)	**-71**	**-12.3**	**-100**	2.27
SEGMENTATION_3D	[8]	20	7533	19	4.995
	ours	2	2251	0	4.985
	comp.(%)	**-90**	**-70.1**	**-100**	**-0.2**
Average(%)		**-66**	**-25**	**-100**	**-1.2**

Table 5: Synthesis experimental results.

tion are extracted from Xilinx ISE report. As shown in Table 5, we use 66% fewer block RAMs than [8]. This stems from 1) the minimum number of buffer banks achieved, and 2) the heterogeneous mapping of buffer banks to variable resources in addition to block RAMs as demonstrated in Table 2. We also use 25% fewer logic slices than [8], even though we implement some of the small reuse buffers in registers. That is because we avoid the modulo scheduling in conventional uniform memory partitioning which generates a hardware transformer to map the original data address to the bank ID and local address via a complex calculation involving multiplication and division. Instead our memory system only needs counters iterating over the data domains in the lexicographic order. This advantage is also reflected by the complete elimination of DPSs in our method. The clock period does not show too much difference between [8] and our method since the back-end flow will stop optimization as long as it meets the 200MHz target. However, our method generally has larger slacks from the target 5.0ns as shown in Table 5. It is mainly due to the distributed structure in our method. We tried to use the Xilinx XPower Analyzer for power estimation, but found that the FPGA power is dominated by the static power, and is almost invariant with custom circuits. If power gating is available in FPGA, the FPGA power will be proportional to resource usage, which is covered by Table 5.

6. CONCLUSIONS AND FUTURE WORK

In this work, we propose non-uniform partitioning which opens a new design space compared to conventional cyclic partitioning framework. As a starting point, we use stencil computation as the initial design target and show a novel memory system that works with non-uniform sizes of reuse buffer. The memory system in the extended design space can achieve the optimal solution with the minimum reuse buffer size and the minimum number of buffer banks. We develop a design automation flow that generates a microarchitecture with our memory systems and integrates it with the computation kernel for a complete design. Experimental results show that our method outperforms the recent memory partitioning work in terms of utilization of variable FPGA resources.

As this is the first work on non-uniform memory partitioning, our primary goal is to show the potential of this new approach. Though stencil computation is a popular application domain and attracts the attention of most memory partitioning work, it is still important to extend non-uniform memory partitioning to general cases. Our data streaming method may not be the only solution for utilizing the non-uniform reuse buffers. A modified modulo scheduling extended from conventional uniform memory partitioning is also a good candidate. We believe that there are many opportunities in future research.

7. ACKNOWLEDGEMENTS

The authors would like to thank Yuxin Wang for sharing the synthesizable codes transformed by [8]. The authors would also like to thank Xilinx for equipment donation and financial contributions. This work is partially supported by the Center for Domain-Specific Computing (CDSC), and C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

8. REFERENCES

[1] Y.-t. Chen, J. Cong, M. A. Ghodrat, M. Huang, C. Liu, B. Xiao, and Y. Zou, "Accelerator-Rich CMPs: From Concept to Real Hardware," in *International Conference on Computer Design*, 2013.

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[3] J. Cong, P. Zhang, and Y. Zou, "Optimizing memory hierarchy allocation with loop transformations for high-level synthesis," in *Design Automation Conference*, 2012, pp. 1229–1234.

[4] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '13*, p. 29, 2013.

[5] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," in *International Conference on Computer-Aided Design*, 2009, p. 697.

[6] Y. Wang, P. Zhang, C. Xu, and J. Cong, "An integrated and automated memory optimization flow for FPGA behavioral synthesis," in *Asia and South Pacific Design Automation Conference*, Jan. 2012, pp. 257–262.

[7] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong, "Memory partitioning and scheduling co-optimization in behavioral synthesis," in *International Conference on Computer-Aided Design*, 2012, pp. 488–495.

[8] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," in *Design Automation Conference*, 2013, p. 1.

[9] Y. Ben-Asher and N. Rotem, "Automatic memory partitioning," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2010, p. 155.

[10] Xilinx, "Vivado High-Level Synthesis." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm

[11] J. Cong, P. Li, B. Xiao, and P. Zhang, "An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers," Computer Science Department, UCLA, TR140009, Tech. Rep., 2014. [Online]. Available: http://fmdb.cs.ucla.edu/Treports/140009.pdf

[12] T. Henretty, J. Holewinski, N. Sedaghati, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Stencil Domain Specific Language (SDSL) User Guide 0.2.1 draft," OSU TR OSU-CISRC-4/13-TR09, Tech. Rep., 2013.

[13] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable Domain-Specific Computing," *IEEE Design and Test of Computers*, vol. 28, no. 2, pp. 6–15, Mar. 2011.

[14] A. A. Nacci, V. Rana, F. Bruschi, D. Sciuto, I. Beretta, and D. Atienza, "A high-level synthesis flow for the implementation of iterative stencil loop algorithms on FPGA devices," in *Design Automation Conference*, 2013, p. 1.

[15] "Bicubic interpolation." [Online]. Available: http://www.mpi-hd.mpg.de/astrophysik/HEA/internal/Numerical_Recipes/f3-6.pdf

[16] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: A Tool for Improved Derivation of Process Networks," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. 1–13, 2007.

[17] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *International Symposium on FPGAs*, 2013, p. 9.

[18] "LLVM-polly." [Online]. Available: http://llvm.org/svn/llvm-project/polly/

[19] "ROSE compiler infrastucture." [Online]. Available: http://rosecompiler.org/

[20] Xilinx, "Virtex-7 FPGA data sheets." [Online]. Available: http://www.xilinx.com/support/documentation/7_series.htm