Efficient Memory Partitioning for Parallel Data Access in Multidimensional Arrays

Chenyue Meng, Shouyi Yin, Peng Ouyang, Leibo Liu, Shaojun Wei
Tsinghua National Laboratory for Information Science and Technology (TNList)
Institute of Microelectronics, Tsinghua University, Beijing, China
yinsy@tsinghua.edu.cn

ABSTRACT

Memory bandwidth bottlenecks severely restrict parallel access of data from memory arrays. To increase bandwidth, memory partitioning algorithms have been proposed to access multiple memory banks simultaneously. However, previous partitioning schemes propose complex partitioning algorithms, which leads to non-optimal memory bank space utilization and unnecessary storage overhead. In this paper, we develop an efficient memory partitioning strategy with low time complexity and low storage overhead for data access in multidimensional arrays. Experimental results show that our memory partitioning algorithm saves up to 93.7% in the amount of arithmetic operations, 96.9% in execution time and 31.1% in storage overhead, compared to the state-of-the-art approach.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: Design Aids - automatic synthesis

General Terms

Algorithms, Design, Performance

Keywords

memory partitioning, parallel data access, storage overhead

1. INTRODUCTION

As the complexity in SoC designs increases exponentially, hardware acceleration starts to play a significant role in meeting the demanding performance requirements. While the number of computational elements rises to enhance processing speed, further performance improvement is limited by memory bandwidth bottleneck. Through increasing memory bandwidth, multiple data can be accessed simultaneously in a single clock cycle. However, it is unrealistic to simply add more physical memory ports due to the expensive cost of power and area consumption [8] and its inapplicability to devices like FPGA whose number of ports is fixed. Duplicating the original memory array into multiple copies [4] can increase the throughput of data access as well but it has significant storage overhead. In comparison, partitioning the original memory array into multiple banks separately can increase memory bandwidth significantly without bringing in too much overhead.

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA

http://dx.doi.org/10.1145/2744769.2744831

Memory partitioning algorithms have been studied to improve system throughput and reduce power consumption under specific circumstances [7][10]. For general purposes, different partitioning schemes are proposed to realize automatic multi-banking strategies [5][1]. For further optimization of performance and power, memory partitioning techniques are discussed together with scheduling [3][2]. And in [9], a memory partitioning algorithm is developed for image and video processing in multidimensional loop nests.

However, in order to guarantee that data are mapped to distinctive addresses in memory banks, plenty of storage space is wasted in previous memory partitioning algorithms. Besides former work provides partition algorithm of exponential time complexity which requires too much time for processing especially for multidimensional memory arrays. Moreover, too many bank numbers can lead to unnecessary hardware resource consumption, which has not been discussed as a constraint before. In this work, we propose an efficient memory partitioning algorithm with limited bank number constraint in multidimensional arrays to realize low storage overhead.

The main contributions of our work are:

1. We formulate the memory partitioning as a multi-objective optimization problem, which is flexible to different design considerations such as minimal memory overhead, fast accessing speed or limited memory bank number.

2. We propose a general solution framework to resolve the memory partitioning problem, which has fast speed and low complexity.

3. The proposed approach can achieve minimal memory overhead, compared to the state-of-the-art methods.

The remainder of this paper is organized as follows. In Section 2, we provide a motivational example and specific solutions for memory partitioning. Section 3 formulates definitions and problem. In Section 4, our proposed algorithm and storage overhead analysis are discussed in detail. Section 5 provides experimental results to compare our memory partitioning algorithm with other work. Section 6 summarizes our key results.

2. MOTIVATIONAL EXAMPLE

In the field of computer vision, edge detection can be operated by Laplacian of Gaussian (LoG) operator proposed in [6]. A typical 5×5 image convolution kernel is described as a coefficient matrix shown in Fig. 1(a). To detect edges in a 640×480 gray-scale image, 2-dimensional loop nest is required, where data elements with non-zero kernel coeffi-

cients should be accessed in each inner loop for processing. The edge detection code is shown in Fig. 1(b).

In this case, 13 out of 25 positions in LoG kernel have non-zero weight coefficients. The access pattern of those 13 positions in memory array is shown in Fig. 2(a). Each dot, whose two-dimensional coordinate is $(x_0, x_1)^T$, represents a data element, among which black dots represent the 13 data elements in LoG kernel that need accessing in a single loop iteration, and gray dots represent the left 12 data elements that do not need to be accessed.

0	0	-1	0	0
0	-1	-2	-1	0
-1	-2	16	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

(a)

1: Define $X[1:640][1:480]$
2: **for** $i = 3; i \leqslant 638; i++$ **do**
3: **for** $j = 3; j \leqslant 478; j++$ **do**
4: $Y[i][j] = -X[i-2][j] - X[i-1][j-1] - 2X[i-1][j] - X[i-1][j+1] - X[i][j-2] - 2X[i][j-1] + 16X[i][j] - 2X[i][j+1] - X[i][j+2] - X[i+1][j-1] - 2X[i+1][j] - X[i+1][j+1] - X[i+2][j]$
5: **end for**
6: **end for**

(b)

Figure 1: (a) LoG kernel coefficients, (b) LoG edge detection code

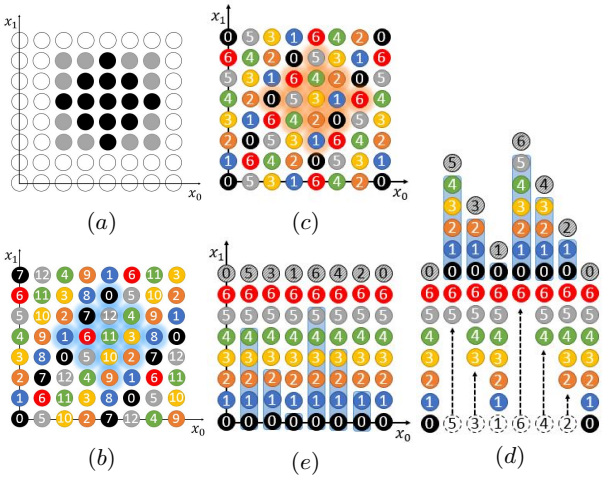

Figure 2: (a) Access pattern of LoG kernel, (b) our 13-bank partitioning solution, (c) 7-bank partitioning solution, (d) storage reorganization, (e) data storage in banks

For simplicity, we suppose the access bandwidth of memory array equals 1. To realize multiple simultaneous accesses, the original memory array must be partitioned into at least 13 memory banks.

A heuristic memory partitioning strategy based on linear transformation is proposed in [9]. It iterates over all the possible linear transformation vectors to find the optimal solution that partitions the original 640×480 array into exactly 13 banks in this case but comes with a huge calculation cost of 1053 arithmetic operations and the storage overhead caused by extra padding is 5450 data elements.

In comparison, for any pattern, our work can quickly generate a particular linear transformation based on the pattern shape and guarantees all data elements in the pattern can be mapped to different memory banks. Through our approach, the complexity of finding transformation is dropped to constant from exponential by avoiding unnecessary global search. For the access pattern in Fig. 2(a), a 13-bank (optimal) solution based on our algorithm is provided with only 92 arithmetic operations (reduced by 91.3%) and 640 extra storage positions for data elements (reduced by 88.3%). The specific partitioning solution is presented in Fig. 2(b). The number in each dot represents the bank index that element is mapped to. For any 13 elements which LoG pattern masks, like the highlighted dots in Fig. 2(b), bank indexes are different. The details of this solution are illustrated as case study in Section 5.

Besides, given the constraint that bank number is limited to be no greater than 10, this work can provide a 7-bank same-bank-size solution that all 13 elements in the LoG pattern can be accessed in two cycles, shown in Fig. 2(c). In other words, given any 13 elements in LoG pattern, there are at most 2 elements marked in the same bank index.

As for the strategy of mapping each data element to a distinctive address in memory banks, we take the partition solution in Fig. 2(c) as an instance. First we move each dot column towards the positive x_1-axis so that bank indexes are the same in every row, shown in Fig. 2(d). For each column, the moving distance is equal to the bank index of the dot lying in x_0-axis, i.e. $0, 5, 3, 1, 6, 4, 2, 0$. Then ignoring gray dots, we move those dots that are above the row of mark-in-6 dots, highlighted in Fig. 2(d), back to empty positions. Then the memory array is well-organized and each row represents data element storage in each bank, shown in Fig. 2(e). Specific mapping strategy of those dots as well as the left gray dots will be illustrated in Section 4.4.

3. PROBLEM FORMULATION

Parallel access to data elements in multiple memory arrays implies accessing data from each memory array in parallel, which can be realized by partitioning each memory array into several banks according to its corresponding access pattern. Towards this goal, we propose a universal and efficient memory partitioning algorithm to realize parallel data access. The bandwidth of each memory array is assumed to be 1 and it's easy to extend our algorithms to the situation where bank bandwidth is B by combing B banks together.

Definition 1. (**Data Domain**) Given a finite n-dimensional memory array X, the address of any data element $\vec{x} \in X$ can be represented by $\vec{x} = (x_0, x_1, \cdots, x_{n-1})^T$, where the interval of x_i is $[0, w_i - 1]$, which indicates that total memory size is $W = \prod_{i=0}^{n-1} w_i$.

Definition 2. (**Pattern**) A pattern consisting of m adjacent data elements is defined as $P = \left\{\vec{\Delta}^{(1)}, \vec{\Delta}^{(2)}, \cdots, \vec{\Delta}^{(m)}\right\}$ where $\vec{\Delta}^{(i)} = (\Delta_0^{(i)}, \Delta_1^{(i)}, ..., \Delta_{n-1}^{(i)})^T$ and $\Delta_j^{(i)}$ is constant for $\forall 1 \leqslant i \leqslant m, 0 \leqslant j \leqslant n-1$. If the position offset of P from origin is $\vec{s} = (s_0, s_1, \cdots, s_{n-1})^T$, then the addresses of data elements in P are $P_{\vec{s}} = \left\{\vec{s} + \vec{\Delta}^{(1)}, \vec{s} + \vec{\Delta}^{(2)}, \cdots, \vec{s} + \vec{\Delta}^{(m)}\right\}$.

Definition 3. (**Bank Mapping**)[9] Any data element $\vec{x}$ in the original memory is mapped to its unique address in memory banks by function $B(\vec{x})$ and $F(\vec{x})$, where $B(\vec{x})$ represents the bank index that $\vec{x}$ is mapped to, and $F(\vec{x})$ represents the offset of $\vec{x}$ inside the bank. Assume the size of bank i is wb_i, then total bank size is $Wb = \sum_{i=0}^{N-1} wb_i$. Storage overhead can be expressed as $\Delta W = Wb - W$.

Definition 4. (**Additional Initiation Interval**) $\delta(II)$ is defined as the increase in original loop initiation interval (II) due to delay in data accessing. For a position offset $\vec{s}$, the addresses of data elements in P are $P_{\vec{s}} = \left\{\vec{x}^{(1)}, \vec{x}^{(2)}, \cdots, \vec{x}^{(m)}\right\}$ and $B_P^{(\vec{s})} = \left\{B\left(\vec{x}^{(1)}\right), \cdots, B\left(\vec{x}^{(m)}\right)\right\}$ represent the corresponding bank indexes. Assume $n^{(\vec{s})}$ is the mode[1] in

[1]Mode: most frequent value in a data set.

$B_P^{(\vec{s})}$, then $A_P^{(\vec{s})}$ is defined as the the number of occurrences of $n^{(\vec{s})}$ in $B_P^{(\vec{s})}$. Then $\delta_P(II)$ can be expressed as $\delta_P = \max_{\forall \vec{s} \in X}\{A_P^{(\vec{s})}\} - 1$.

In other words, $\delta_P = 0$ means all m data elements in pattern P are accessed in one cycle. To avoid access conflicts, all elements are supposed to be mapped to different banks.

Given any valid memory partitioning solution, any two different data elements in the original memory array must be mapped to different addresses in banks. (constraint 1) Besides, too many memory banks can also cause unnecessary hardware cost, like area, routing and control logic. We set the upper bound N_{max} to limit the number of banks. (constraint 2)

Problem 1. Given any P accessing m elements, find B and F to minimize
1. δ_P towards 0 (additional II)
2. N towards m (bank number)
3. ΔW towards 0 (storage overhead)

subject to
1. $\forall \vec{x} \neq \vec{y} \in X,\ B(\vec{x}) \neq B(\vec{y})$ *or* $B(\vec{x}) = B(\vec{y}), F(\vec{x}) \neq F(\vec{y})$
2. $N \leqslant N_{max}$

For this three-object optimization problem, we consider each variable one by one to simplify their complicated relationship. Apparently different optimizing orders lead to solutions of different concerns. In Section 4, we first aim at $\delta_P = 0$, followed by minimizing N. ΔW will be discussed generally because the storage overhead of our partitioning strategy is only related to the bank number N.

4. PARTITIONING ALGORITHMS

To map a data element $\vec{x}$ to a new position in banks according to its n-dimensional address vector $(x_0, x_1, \cdots, x_{n-1})^T$, a fast and low complexity method is linear transformation. Suppose the memory is divided into N banks and the transformation vector is defined as $\vec{\alpha} = (\alpha_0, \alpha_1, ..., \alpha_{n-1})$, then the bank index B can be determined by

$$B(\vec{x}) = (\vec{\alpha} \cdot \vec{x}) \,\%N$$

Simultaneous access to all m data elements in $P \Rightarrow \delta_P = 0$, which satisfies that

$$\forall \vec{s} \in X,\ \forall \vec{x}^{(i)}, \vec{x}^{(j)} \in P_{\vec{s}}, i \neq j \Rightarrow B\left(\vec{x}^{(i)}\right) \neq B\left(\vec{x}^{(j)}\right)$$

which implies $\vec{\alpha} \cdot \vec{x}^{(i)} \neq \vec{\alpha} \cdot \vec{x}^{(j)}$. So $\vec{\alpha}$ must satisfy that each data element in P is assigned with a distinctive number after the linear transformation.

The overview of our partitioning strategy is to determine $\vec{\alpha}$ quickly, based on which, δ_P can be optimized to be 0, i.e. all data elements in pattern can be mapped to different banks. Then an algorithm is proposed to minimize the bank number N under $\delta_P = 0$ condition and with $N \leqslant N_{max}$ constraint, followed by the analysis of storage overhead caused by multi-banking.

4.1 Linear Transformation

The process of calculating $\vec{\alpha} \cdot \vec{x} = \sum_{i=0}^{n-1} \alpha_i x_i$ is equal to multiplying a weight coefficient α_i to each component x_i and sum them up. In comparison of conversion from binary to decimal, the k_{th} digit multiplies its weight coefficient 2^{k-1}. Here 2 is equal to the maximum difference of digit numbers (0 and 1) plus one. Based on the same idea, here are the details of us determining $\vec{\alpha}$.

Assume P consisting of m elements is described as $P = \left\{\vec{\Delta}^{(1)}, \vec{\Delta}^{(2)}, \cdots, \vec{\Delta}^{(m)}\right\}$, where $\vec{\Delta}^{(i)} = (\Delta_0^{(i)}, \Delta_1^{(i)}, ..., \Delta_{n-1}^{(i)})^T$. Define the maximum difference of each component number to be

$$D_j = \max_{1 \leqslant i \leqslant m}\{\Delta_j^{(i)}\} - \min_{1 \leqslant i \leqslant m}\{\Delta_j^{(i)}\} + 1$$

Apparently $\forall j$ satisfying $0 \leqslant j \leqslant n-1 \Rightarrow D_j \geqslant 1$. Then each component of $\vec{\alpha} = (\alpha_0, \alpha_1, ..., \alpha_{n-1})$ can be defined as

$$\alpha_j = \prod_{k=j+1}^{n-1} D_k$$

Specially, $\alpha_{n-1} = \prod_{k=n}^{n-1} D_k = 1$.

4.2 Additional Initiation Interval

First, we prove that for any pattern P, δ_P can be optimized to be 0 with $\vec{\alpha}$ determined in Section 4.1.

Given $\forall \vec{s} \in X$, $P_{\vec{s}} = \left\{\vec{s} + \vec{\Delta}^{(1)}, \vec{s} + \vec{\Delta}^{(2)}, \cdots, \vec{s} + \vec{\Delta}^{(m)}\right\}$. Then we define the numbers after linear transformation, $\left\{\vec{\alpha} \cdot \left(\vec{s} + \vec{\Delta}^{(1)}\right), \vec{\alpha} \cdot \left(\vec{s} + \vec{\Delta}^{(2)}\right), \cdots, \vec{\alpha} \cdot \left(\vec{s} + \vec{\Delta}^{(m)}\right)\right\}$ to be $\left\{y^{(1)}, y^{(2)}, \cdots, y^{(m)}\right\}$.

THEOREM 1. *Given $\vec{\alpha}$ defined above, we have*

$$\forall\ 1 \leqslant i < j \leqslant m,\ y^{(i)} \neq y^{(j)}$$

PROOF. First we remove the uncertainty from $\vec{s}$. Define $\left\{z^{(1)}, z^{(2)}, \cdots, z^{(m)}\right\} = \left\{\vec{\alpha} \cdot \vec{\Delta}^{(1)}, \vec{\alpha} \cdot \vec{\Delta}^{(2)}, \cdots, \vec{\alpha} \cdot \vec{\Delta}^{(m)}\right\}$. Then $y^{(i)} = z^{(i)} + \vec{\alpha} \cdot \vec{s}$, where $\vec{\alpha} \cdot \vec{s}$ is the same part for every $y^{(i)}$ and won't affect the difference between $y^{(i)}$ and $y^{(j)}$. So if $z^{(i)}$ is proofed to be different from each other, then Theorem 1 will be justified. For $1 \leqslant i \leqslant m$, we have $z^{(i)} = \vec{\alpha} \cdot \vec{\Delta}^{(i)} = \sum_{j=0}^{n-1} \alpha_j \Delta_j^{(i)} = \sum_{j=0}^{n-1} \left(\prod_{k=j+1}^{n-1} D_k\right) \Delta_j^{(i)} = \Delta_{n-1}^{(i)} + \Delta_{n-2}^{(i)} D_{n-1} + \Delta_{n-3}^{(i)} D_{n-2} D_{n-1} + \cdots + \Delta_0^{(i)} \prod_{k=1}^{n-1} D_k$

Given $\forall\ 1 \leqslant i < j \leqslant m$, $\vec{\Delta}^{(i)} \neq \vec{\Delta}^{(j)}$. Then we have

$$z^{(i)} = \Delta_{n-1}^{(i)} + D_{n-1}\left(\Delta_{n-2}^{(i)} + D_{n-2}\left(\Delta_{n-3}^{(i)} + \cdots\right)\right) \quad (1)$$

$$z^{(j)} = \Delta_{n-1}^{(j)} + D_{n-1}\left(\Delta_{n-2}^{(j)} + D_{n-2}\left(\Delta_{n-3}^{(j)} + \cdots\right)\right) \quad (2)$$

The converse-negative proposition of theorem is proved as follows: Suppose $z^{(i)} = z^{(j)}$ then Eqn.1 minus Eqn.2 is

$$0 = \Delta_{n-1}^{(\delta)} + D_{n-1}\left(\Delta_{n-2}^{(\delta)} + D_{n-2}\left(\Delta_{n-3}^{(\delta)} + \cdots\right)\right) \quad (3)$$

where $\Delta_k^{(\delta)} = \Delta_k^{(i)} - \Delta_k^{(j)}$. Given the definition that $D_j = \max_{1 \leqslant i \leqslant m}\{\Delta_j^{(i)}\} - \min_{1 \leqslant i \leqslant m}\{\Delta_j^{(i)}\} + 1 \Rightarrow \left|\Delta_k^{(\delta)}\right| \leqslant D_k - 1$, we have $D_k \nmid \Delta_k^{(\delta)}$. Unless $D_k = 1$ which is equivalent to $\Delta_k^{(i)} = \Delta_k^{(j)}$. Given $\vec{\Delta}^{(i)} \neq \vec{\Delta}^{(j)} \Rightarrow \exists\ l$ satisfying $0 \leqslant l \leqslant n-1$ s.t. $\Delta_l^{(\delta)} \neq 0$ and $\Delta_{n-1}^{(\delta)} = \Delta_{n-2}^{(\delta)} = \cdots = \Delta_{l+1}^{(\delta)} = 0$.

Then Eqn.3 can be simplified as

$$0 = \prod_{k=l+1}^{n-1} D_k\left(\Delta_l^{(\delta)} + D_l\left(\Delta_{l-1}^{(\delta)} + D_{l-1}\left(\Delta_{l-2}^{(\delta)} + \cdots\right)\right)\right)$$

$$0 = \Delta_l^{(\delta)} + D_l\left(\Delta_{l-1}^{(\delta)} + D_{l-1}\left(\Delta_{l-2}^{(\delta)} + \cdots\right)\right)$$

$\Delta_l^{(i)} \neq \Delta_l^{(j)} \Rightarrow D_l \nmid \Delta_l^{(\delta)}$. Then to satisfy the above equation $\Rightarrow \Delta_l^{(\delta)} = 0$, which is contradictory to our previous assumption. □

According to Thm.1, $\forall N \geqslant \max_{1 \leqslant i \leqslant m}\{y^{(i)}\} - \min_{1 \leqslant j \leqslant m}\{y^{(j)}\} + 1$ satisfies that all m data elements in P can be mapped into

different banks because the bank index $B\left(\vec{x}^{(k)}\right) = y^{(k)}\%N$ are different from each other. Feasible partitioning solutions must exist such that $\delta_P = 0$, i.e. realizing simultaneous access to m data elements, regardless of N_{max}. The detailed discussion of minimizing N under the condition of N_{max} is in Section 4.3.

4.3 Bank Number

With $\vec{\alpha}$ defined in Section 4.1, there exists infinite $\delta_P = 0$ solutions partitioning the original memory array into N banks without the constraint of N_{max}. We define N_f as the bank number with no N_{max} constraint, and N_c as the bank number satisfying $N_c \leqslant N_{max}$.

4.3.1 Minimize N_f

Algorithm 1 Minimize N_f

```
1: for i = 1; i ⩽ m; i++ do
2:     z^(i) = α⃗ · Δ^(i)
3: end for
4: Define Q = ∅
5: for i = 1; i < m; i++ do
6:     for j = i + 1; j ⩽ m; j++ do
7:         Q = Q ∪ |z^(i) − z^(j)|
8:     end for
9: end for
10: Define M = max{Q} //M = max_{1⩽i⩽m}{z^(i)} − min_{1⩽j⩽m}{z^(j)}
11: Define E[1 : M] = [0, 0, ··· , 0]
12: while Q ≠ ∅ do
13:     q ← Q //Choose any element q from Q
14:     Q = Q \ q
15:     E[q] = E[q] + 1
16: end while
17: Define N_f = m, k = 1
18: while kN_f ⩽ M do
19:     if E[kN_f] ≠ 0 then //E[d] ≠ 0 ⇒ d ∈ Q
20:         N_f = N_f + 1
21:         k = 1
22:     else
23:         k = k + 1
24:     end if
25: end while
```

Here is the main idea of Algorithm 1. Set $\mathbf{Q}$ consists of all the absolute values of subtractions of $z^{(i)}$ and $z^{(j)}$. N_f needs to satisfy that all the integral multiples of N_f, like kN_f, don't belong in $\mathbf{Q}$. Then $b^{(i)} = z^{(i)}\%N_f$ will be different from each other, which means all m data elements are partitioned into different memory banks.

Starting from $N_f = m$, we assume $N_f = m + C$ meets our demands after executing line 17 to 25 in pseudo code for C times. Then together with a constant time complexity of determining $\vec{\alpha}$, the time complexity of our algorithm can be estimated as $O\left(m^2 + \sum_{i=0}^{C}\left\lceil\frac{M}{m+i}\right\rceil\right) \approx O\left(m^2\right)$.

In comparison, the proposed algorithm in [9] iterates over all N^n linear transformation vectors for each N in the n-dimensional memory, and for each vector, it takes $O\left(m^2\right)$ times to justify the solution. Similarly, starting from $N = m$ and after C cycles it finds a valid partitioning solution. Then the time complexity is $O\left(m^2\sum_{i=0}^{C}(N+i)^n\right) \approx O\left(CN^n m^2\right)$.

4.3.2 Determine N_c

A fast approach is determining N_c based on N_f. If $N_f \leqslant N_{max}$, then $N_c = N_f$. Otherwise, we define $F = \left\lceil\frac{N_f}{N_{max}}\right\rceil$. Apparently we only need to access N_{max} memory banks F times at most to get all m data elements. And N_c can be determined by $N_c = \left\lceil\frac{N_f}{F}\right\rceil$. Then we can define

$$B\left(\vec{x}\right) = \left(\left(\vec{\alpha}\cdot\vec{x}\right)\%N_f\right)\%N_c$$

Using this fast approach, however, memory array might be partitioned into multiple banks of different sizes because N_f might not be divisible by N_c. To partition memory into same-size banks, we propose an alternative approach satisfying minimum δ_P and $N_c \leqslant N_{max} < N_f$.

Given $\vec{\alpha}$ defined in Section 4.1, $A_P^{(\vec{s})}$ in Definition 4 is not related to $\vec{s}$ any more because every $y^{(i)} = \vec{\alpha}\cdot\vec{x}^{(i)}$ in $B\left(\vec{x}^{(i)}\right) = y^{(i)}\%N$ has the same part $\vec{\alpha}\cdot\vec{s}$. Then for each N, we have $B_P|_N = \left\{B\left(\vec{\Delta}^{(1)}\right),\cdots,B\left(\vec{\Delta}^{(m)}\right)\right\}$, where $B\left(\vec{\Delta}^{(i)}\right) = \left(\vec{\alpha}\cdot\vec{\Delta}^{(i)}\right)\%N$. $\delta_P + 1 = A_P$ represents the number of occurrences of the mode of B_P. For each N from 1 to N_{max}, we calculate $\delta_P|_N$. The minimum number represents the optimal solution, and the corresponding N is N_c.

4.4 Storage Overhead

As long as linear transformation $\vec{\alpha}$ is determined as before, our data mapping strategy is valid for any given bank number N. For a data element whose address is $\vec{x}$, the bank index it is mapped to is $B\left(\vec{x}\right) = \left(\vec{\alpha}\cdot\vec{x}\right)\%N$, and the offset inside the bank is $F\left(\vec{x}\right)$.

Similar to the approach demonstrated in Fig. 2(d)(e) in Section 2, given $\vec{x} = \left(x_0, x_1, \cdots, x_{n-1}\right)^T$, the coordinate after moving is $\left(x_0, x_1, \cdots, x_{move}\right)^T$, where $x_{move} = x_{n-1} + \left(\vec{\alpha}\cdot\left(x_0, x_1, \cdots, x_{n-2}, 0\right)^T\right)\%N = x_{n-1} + \left(\vec{\alpha}\cdot\vec{x} - x_{n-1}\right)\%N$ because $\alpha_{n-1} = 1$. Then we move those elements that exceed boundaries back to empty positions. Notice the fact that our mapping strategy can be extended to any moving distance m_d (Here $m_d = \left(\vec{\alpha}\cdot\vec{x} - x_{n-1}\right)\%N$) as long as $m_d\%N = \left(\vec{\alpha}\cdot\vec{x} - x_{n-1}\right)\%N$, we can set $m_d = \vec{\alpha}\cdot\vec{x} - x_{n-1}$ to simplify $x_{move} = x_{n-1} + \vec{\alpha}\cdot\vec{x} - x_{n-1} = \vec{\alpha}\cdot\vec{x}$.

4.4.1 Data Mapping Strategy

Here are the details of determining function $F\left(\vec{x}\right)$ which guarantees that different data elements are mapped to different addresses in memory banks.

Given $\forall\vec{x}\in X$, $\vec{x} = \left(x_0, x_1, \cdots, x_{n-1}\right)^T$, where the interval of x_i is $[0, w_i - 1]$. Define $K = \left\lfloor\frac{w_{n-1}}{N}\right\rfloor$ and we suppose banks are n-dimensional, the same as the original memory array. First we define $F\left(\vec{x}\right)$ for $\vec{x}$ whose $x_{n-1}\in[0, KN-1]$ as follows: $F\left(\vec{x}\right) = \left(x_0, x_1, \cdots, x_{n-2}, x_{new}\right)^T$ where

$$x_{new} = \left\lfloor\frac{\left(\vec{\alpha}\cdot\vec{x}\right)\%(KN)}{N}\right\rfloor$$

Comparing $F\left(\vec{x}\right)$ to $\vec{x}$, only the $(n-1)_{th}$-dimensional coordinate is different. For the original memory array where $x_{n-1}\in[0, KN-1]$, the storage size is $KN\prod_{i=0}^{n-2}w_i$. And for each memory bank, since the range of $\left(\vec{\alpha}\cdot\vec{x}\right)\%(KN)$ is $[0, KN-1] \Rightarrow 0 \leqslant x_{new} \leqslant K-1$. So the bank size is $K\prod_{i=0}^{n-2}w_i$. Then the total bank size is the same as the original memory. So the mapping function B and F causes no storage overhead.

Through this approach, every data element is guaranteed to have a unique address in memory banks. Proof is omitted due to page limit.

4.4.2 Storage Overhead

As for the left data elements $x_{n-1}\in[KN, w_{n-1}]$, we can access them one by one and map them into banks according

to their bank index, which causes no storage overhead but high complexity. We prefer using the above approach as well, which might cause extra storage space.

The storage overhead is $\left(\lceil \frac{w_{n-1}}{N} \rceil N - w_{n-1}\right) \times \prod_{k=0}^{n-2} w_k$. And the maximum overhead is $(N-1) \times \prod_{k=0}^{n-2} w_k$, which is $\frac{1}{n}$ of the overhead in [9] on average.

5. EXPERIMENTAL RESULTS

5.1 Case Study

To illustrate our algorithm, we take the pattern ($m = 13$) in 2-dimensional memory array ($n = 2$) shown in Fig. 2(a), also in Fig. 3(a) as an example. $P = \left\{\vec{\Delta}^{(1)}, \vec{\Delta}^{(2)}, \cdots, \vec{\Delta}^{(m)}\right\}$ can de described in $(x_0, x_1)^T$ as

$$P = \left\{ \begin{pmatrix} 2 \\ 4 \end{pmatrix} \begin{pmatrix} 3 \\ 3 \end{pmatrix} \begin{pmatrix} 3 \\ 4 \end{pmatrix} \cdots \begin{pmatrix} 5 \\ 4 \end{pmatrix} \begin{pmatrix} 5 \\ 5 \end{pmatrix} \begin{pmatrix} 6 \\ 4 \end{pmatrix} \right\}$$

$D_0 = 5, D_1 = 5 \Rightarrow \vec{\alpha} = (D_1, 1) = (5, 1)$. Then we have $\left\{z^{(1)}, z^{(2)}, \cdots, z^{(m)}\right\} = \{14, 18, 19, \cdots, 29, 30, 34\}$. According to Algorithm 1, $\mathbf{Q} = \{1, 2, \cdots, 11, 12, 14, 15, 16, 20\}$. For $N_f = 13$, $kN_f \notin \mathbf{Q}$ $(k = 1, 2, 3 \cdots)$ is satisfied. So we can define that $B(\vec{x}) = (\vec{\alpha} \cdot \vec{x}) \,\%N_f$, then the bank indexes of all the 13 data elements are mapped to are $\{1, 5, 6, 7, 9, 10, 11, 12, 0, 2, 3, 4, 8\}$, highlighted in Fig. 2(b). The number in each dot represents the bank index of the element mapped to.

Considering a $N_{max} = 10$ constraint, we take the fast approach to calculate $F = \lceil \frac{13}{10} \rceil = 2 \Rightarrow N_c = \lceil \frac{13}{2} \rceil = 7$, which means bank 0 and 7, 1 and 8, $\cdots$, 5 and 12 should be combined together. To access all the 13 data elements, 7 memory banks need to be accessed twice. Each bank index is marked in different colors, shown in Fig. 2(b). Also, if the bandwidth of memory bank is 2, i.e. 2 data elements can be accessed simultaneously, we can also use this method to combine two banks together into one to reduce bank number from 13 to 7.

However, based on the fast approach, bank sizes are not the same because $7 \nmid 13$. For instance, new bank 0 consists of previous banks 0 and 7, while new bank 6 is the previous bank 6. Then an alternative strategy is proposed and we have $\left\{\vec{\alpha} \cdot \vec{\Delta}^{(1)}, \cdots, \vec{\alpha} \cdot \vec{\Delta}^{(m)}\right\} = \{14, 18, 19, \cdots, 29, 30, 34\}$. For each N, bank index set B_P is determined and the number of occurrences of the mode in B_P is shown as follows

N	1	2	3	4	5	6	7	8	9	10
$\delta_P\vert_N + \mathbf{1}$	13	9	5	6	5	3	2	3	2	3

The minimum $\delta_P|_N$ is 1, meaning banks need to be accessed twice to acquire all m data elements, and accordingly $N_c = 7$ or 9. We partition the original memory array into 7 same-size banks, shown in Fig. 2(c).

5.2 Experimental Setup

The experiments of comparing our work to the state-of-art linear transformation based (LTB) memory partitioning algorithm proposed in [9] are divided into two steps. First we apply the solutions provided by our and LTB algorithms to several benchmarks on Cyclone DE2-115 FPGA platform to compare the minimum bank numbers for parallel data access with no bank number limit and the storage overhead caused by partitioning the memory array to multiple banks. The storage overhead is measured in the number of 9kb memory blocks. Second we assess the performance of ours and LTB algorithm by the amount of arithmetic operations (addition, subtraction, multiplication, division, etc) and the execution time it takes to find that solution. The experimental platform is a 4-core 2.9GHz PC and we run the two algorithms on the same benchmarks for 10000 times to average.

The selected benchmarks are 5 common access patterns for edge detection in computer vision [11], shown in Fig. 3 and the following numbers in brackets represent the number of data elements in each pattern. Specially, Prewitt operator includes both vertical and horizontal kernels, which form the pattern as Fig. 3(c). Structure element (SE) in Fig. 3(d) is a detection operator proposed in [11]. Fig. 3(e) is a Sobel operator for 3D edge detection, and the 3_{rd}-dimension components are shown beside (e).

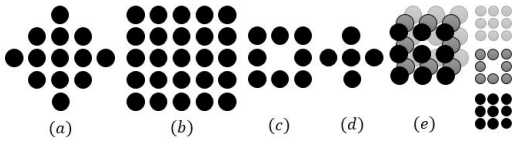

Figure 3: Access patterns (a) LoG (13), (b) Canny (25), (c) Prewitt (8), (d) SE (5), (e) Sobel(3D) (26)

For hardware experiments, the original memory size of data elements is set to be common image resolutions SD(640×480), HD(1280×720), FullHD(1920×1080), WQXGA(2560×1600) and 4K(3840×2160) to illustrate the practical storage cost for our and Wang's solutions. Specially for Sobel (3D) the $3rd$-dimension has 400 samples for all memory sizes.

5.3 Experimental Results

The experimental results are shown in Table 1 and the storage overhead is measured in *memory block*s. For the 5 selected edge detection patterns, the bank numbers calculated by our algorithm are exactly the same as LTB. However LTB iterates over all the possible linear transform vectors to find the minimum/optimal bank number so our bank number must be greater or at most equal to LTB's. To illustrate this point, we select two more access patterns for our experiments, where Median filter pattern has 7 elements and Gaussian filter pattern has 9 elements. Our algorithm needs to partition 8 and 13 banks respectively to realize parallel data access while LTB only needs 7 and 10. With bearable increase in bank numbers, our work can reduce the amount of arithmetic operations by 93.7% and execution time by 96.9% on average compared to LTB.

As for the storage overhead, both our and LTB solutions are related to the division relationship between the bank number and memory array size. Take LoG as an example, the remainder of w_{n-1} divided by 13 changes with the memory array size, where $w_{n-1} = 480, 720, 1080, 1600$ and 2160. For SD memory size, $\lceil \frac{480}{13} \rceil 13 - 480 = 1$ gives a quite low overhead, leading to a relatively high improvement. For WQXGA memory size, however, $\lceil \frac{1600}{13} \rceil 13 - 1600 = 12$ means lots of memory in banks are wasted so the storage overhead improvement is relatively low. Nevertheless, when our algorithm and LTB provide solutions of the same bank number, like the first 5 patterns, the storage overhead of ours is guaranteed to be no more than LTB's, and the average reduction is 43.0%. The storage overhead of our and LTB solutions to Sobel (3D) are both large compared to other patterns because the original memory size is 400 times larger caused by $3rd$-dimension. When the bank numbers are different, like the last 2 patterns, the situation is complicated. Taking Median pattern as an example, our bank number is 8, which can divide all array length so the storage overhead is 0 for all memory sizes; LTB offers a solution of 7-bank, where any array length is not divisible by 7, so the storage

Table 1: Experimental results

		Bank number	Storage overhead /(memory block) SD	HD	FullHD	WQXGA	4K	Arithmetic operations	Execution time /(ms)
LoG	LTB	13	10	28	49	58	106	1053	0.575
	ours	13	2	19	41	55	76	92	0.024
	improvement(%)	-	80.0%	32.1%	16.3%	5.2%	28.3%	91.3%	95.8%
Canny	LTB	25	32	38	79	43	142	5575	1.451
	ours	25	23	12	69	0	103	325	0.024
	improvement(%)	-	28.1%	68.4%	12.7%	100%	27.5%	94.2%	98.3%
Prewitt	LTB	9	14	9	12	24	12	2784	2.472
	ours	9	7	0	0	10	0	37	0.018
	improvement(%)	-	50.0%	100%	100%	58.3%	100%	98.7%	99.3%
SE	LTB	5	0	0	0	0	0	120	0.188
	ours	5	0	0	0	0	0	16	0.015
	improvement(%)	-	0%	0%	0%	0%	0%	86.7%	92.0%
Sobel 3D	LTB	27	8193	24578	36864	78508	105984	4564742	1108
	ours	27	2731	8192	18432	36409	73728	352	0.025
	improvement(%)	-	66.7%	66.7%	50.0%	53.6%	30.4%	100.0%	100.0%
Median	LTB	7	7	4	27	20	33	217	0.241
	ours	8	0	0	0	0	0	30	0.015
	improvement(%)	-	100%	100%	100%	100%	100%	86.2%	93.8%
Gaussian	LTB	10	0	0	0	0	0	3996	3.038
	ours	13	2	19	41	55	76	50	0.017
	improvement(%)	-	-100%	-100%	-100%	-100%	-100%	98.7%	99.4%
Average improvement(%)		-	31.1%					93.7%	96.9%

overhead is inevitable. Nevertheless, for general comparison, our work can reduce the storage overhead for 31.1%.

6. CONCLUSIONS

Memory partition can increase memory bandwidth significantly and enable parallel data access, which is formulated as a multi-object optimization problem in this work. An efficient partitioning algorithm that has low complexity and low storage overhead is proposed, which can be extended to limited bank number situation and even zero storage overhead demand by adjusting the optimizing order. Compared to the state-of-art memory partitioning strategy, our work reduces the arithmetic operation amount and execution time by 93.7% and 96.9%, respectively, and the storage overhead can be saved up to 31.1%.

7. ACKNOWLEDGEMENT

This work is supported in part by the China Major S&T Project (No.2013ZX01033001-001-003), the International S&T Cooperation Project of China grant (No.2012DFA11170), the Tsinghua Indigenous Research Project (No.20111080997), the NNSF of China grant (No.61274131) and the China 863 Program (No.2012AA012701).

8. REFERENCES

[1] Y. Ben-Asher and N. Rotem. Automatic memory partitioning: increasing memory parallelism via data structure partitioning. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 155–162. ACM, 2010.

[2] J. Cong, W. Jiang, B. Liu, and Y. Zou. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(2):15, 2011.

[3] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong. Memory partitioning and scheduling co-optimization in behavioral synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 488–495. ACM, 2012.

[4] Q. Liu, T. Todman, and W. Luk. Combining optimizations in automated low power design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1791–1796. European Design and Automation Association, 2010.

[5] A. Macii, E. Macii, and M. Poncino. Improving the efficiency of memory partitioning by address clustering. In *DATE*, pages 10018–10023, 2003.

[6] D. Marr and E. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 207(1167):187–217, 1980.

[7] S. Ramprasad, N. R. Shanbhag, and I. N. Hajj. Low-power distributed arithmetic architectures using nonuniform memory partitioning. In *Circuits and Systems, 1999. ISCAS'99. Proceedings of the 1999 IEEE International Symposium on*, volume 3, pages 470–473. IEEE, 1999.

[8] Y. Tatsumi and H. Mattausch. Fast quadratic increase of multiport-storage-cell area with port number. *Electronics Letters*, 35(25):2185–2187, 1999.

[9] 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*, page 12. ACM, 2013.

[10] M. Xie, D. Tong, K. Huang, and X. Cheng. Improving system throughput and fairness simultaneously in shared memory cmp systems via dynamic bank partitioning. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 344–355. IEEE, 2014.

[11] Y. Zhao, W. Gui, and Z. Chen. Edge detection based on multi-structure elements morphology. In *Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on*, volume 2, pages 9795–9798. IEEE, 2006.