A Hybrid Circular Queue Method for Iterative Stencil Computations on GPUs

Yang Yang[1,2], Hui-Min Cui[1,2], Xiao-Bing Feng[1], and Jingling Xue[3]

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *Graduate University of Chinese Academy of Sciences, Beijing 100190, China*

[3] *Programming Languages and Compilers Group, School of Computer Science and Engineering University of New South Wales, Sydney, NSW 2052, Australia*

Abstract In this paper, we present a hybrid circular queue method that can significantly boost the performance of stencil computations on GPU by carefully balancing usage of registers and shared-memory. Unlike earlier methods that rely on circular queues predominantly implemented using indirectly addressable shared memory, our hybrid method exploits a new reuse pattern spanning across the multiple time steps in stencil computations so that circular queues can be implemented by both shared memory and registers effectively in a balanced manner. We describe a framework that automatically finds the best placement of data in registers and shared memory in order to maximize the performance of stencil computations. Validation using four different types of stencils on three different GPU platforms shows that our hybrid method achieves speedups up to 2.93X over methods that use circular queues implemented with shared-memory only.

Keywords stencil computation, circular queue, GPU, occupancy, register

1 Introduction

Iterative stencil computations form what are known as stencil kernels in which some computations are performed successively in multiple time steps until convergence. The intermediate results are not of interest and only the results computed at the last time step are needed. Thus, it is important to improve performance by reusing the data accessed multiple times across the multiple time steps. Otherwise, memory bandwidth may become a serious performance bottleneck. Thus, a lot of prior work exploits this kind of reuse across multiple time steps[1-5].

The circular queue method introduced in [6] can exploit such data reuse across the time steps. By keeping only necessary temporary data on-chip, the on-chip storage required is low as it is proportional to the number of time steps, T, computed at a time. In addition, the on-chip storage required is reused circularly. Earlier implementations of circular queue algorithms on platforms of X86[1] and Cell[7] have resulted in good performance. These earlier efforts show that these algorithms can be flexibly implemented via indirect memory addressing, which requires the underlying memory to be indirectly addressable, as is the case for the main memory in X86 and the local memory in Cell.

Recently, circular queue methods have also been implemented on GPU, which has become an important parallel computing architecture. A naive solution can be obtained by implementing a circular queue using only its on-chip shared memory, which is indirectly addressable. However, this will not be efficient for GPU since its large number of (on-chip) registers is not utilized. Compared to shared memory, the register file is larger and no extra instructions to access are needed. Therefore, it is possible to accelerate the performance of iterative stencil computations further if registers can also be used in implementing circular queues. Furthermore, registers and shared memory should be used in a balanced manner to increase thread occupancy and

the length of tiles can be arbitrarily long. For 3D Jacobi, a grid is decomposed into cubes, each of which is assigned to a thread block. Within each thread block, each thread is responsible for executing a small tile in a plane. A thread block executes a cube plane by plane along the Z-direction. For the same reasons, cubes can be arbitrarily long in the Z-direction. However, in the X- and Y-directions, their lengths are determined by thread block size and thread granularity.

Circular Queue with Multiple Time Steps. The circular queue method can compute multiple time steps in a row between loading data from and storing results back to off-chip DRAM. The parameter T describes the the number of time steps computed between the loading and storing. Using a large T means computing more time steps between loading from and storing to off-chip global memory, which increases the ratio of computation to memory access. On the other hand, more redundant computations arise due to large ghost zones used[8]. Therefore, a choice of T involves tradeoffs between reducing memory traffic and tolerating computation redundancy.

Data-Flow. Fig.2 shows how stencil computations are performed for each thread block for a 2D stencil, in which Figs. 2(a)∼2(d) show the pipeline setup and Figs. 2(e)∼2(f) show the steady state. By steady state, we mean the computations of full skewed columns, where final results are computed. Pipeline setup refers to the computation of the few partial skewed columns at the beginning of the whole process. The pyramid-like grid is computed by a thread block. In this case, three consecutive time steps are computed at a time ($T = 3$). The bottom yellow bars are data loaded from global memory at the time step 0. The top red bars are computed results at the time step 3, which are to be stored back to global memory. The middle blue bars represent computed results at the time steps 1 and 2. In each iteration, a skewed column (dark-colored bars in Figs. 2(b)∼2(f), marked by black circles) is computed.

Within each iteration, the computation is performed from bottom-right towards top-left. For each iteration in the steady state (Figs. 2(e) and 2(f)), at least two skewed columns plus one bar of data are needed to perform the computation. The storage space is immediately retrieved for newly loaded or computed data once it is no longer used.

Listing 1 shows the pseudo code of the circular queue method for 2D 5-point stencil implemented with shared memory arrays with $3 \times T$ storage locations[1]. Values computed in the current skewed column (the inner loop computes a skewed column) is used by adjacent threads during the next skewed column, and thus synchronization happens after each skewed column is computed to

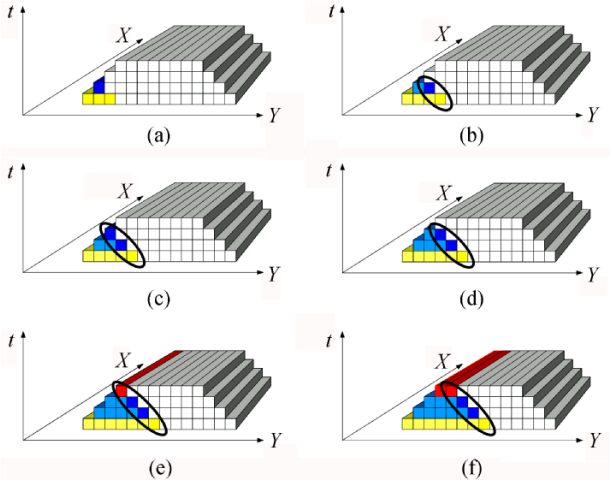

Fig.2. The computation process of a 2D stencil on GPU. (a)∼(d) show the pipeline setup and (e) ∼ (f) show the steady state. The pyramid-like grid is computed by a thread block. The bottom yellow bars are data loaded from global memory, the top red bars are results to be stored back to global memory and the middle blue bars are intermediate values. White bars correspond to data not yet loaded or computed. Bars with darker colors indicate data being loaded or computed in the current iteration. In each subfigure except (a), those dark-colored bars form a skewed column, marked by black circles. Only those bars to the bottom-left, bottom-right and under a dark-colored bar need to be stored on-chip. In each diagram, computations are performed in order from bottom-right to top-left along skewed columns. Storage spaces are recycled for reuse immediately after values are no longer used.

Listing 1. Pseudo Code of the Circular Queue Method Implemented With Shared Memory Arrays with $3 \times T$ Storage Locations

```
1  #define tx threadIdx.x
2  __shared__ float sh[ThreadBlock_Size + 2][3][T];
3  for (y = 2 * T; y < tile_y; y++) {
4      load data from global memory to sh[tx + 1][y%3][0]
5      for (t = 1; t <= T; t++) {
6          result = 1/5* (sh[tx + 1][(y + 2)%3][(t − 1)%T] +
7  sh[tx + 1][y%3][t%T] + sh[tx + 1][(y + 1)%3][(t − 1)%T] +
8  sh[tx][y%3][t%T] + sh[tx + 2][y%3][t%T]);
9          if (t < T)
10             sh[tx + 1][y%3][t%T] = result;
11         else
12             store result to global memory
13     }
14     __syncthreads();
15 }
```

to ensure data exchange.

2.2 Problems with Shared-Memory-Based Circular Queue on GPU

2.2.1 *Unbalanced Usage of On-Chip Storages and the Overhead of Accessing Shared Memory*

For GPU, the concept of occupancy[9] is defined as:

$$Occupancy = \frac{No.\ of\ active\ threads\ per\ SM}{max\ No.\ of\ active\ threads\ per\ SM}. \quad (1)$$

SM in the above formula is short for Streaming Multiprocessor[9]. Higher occupancy indicates more threads to hide latency, which may have great performance impact. According to [10], the number of active threads per SM (NATPS) is determined by

$$NATPS = \min\Big\{8, \frac{registers\ per\ SM}{registers\ per\ thread\ block}, \frac{shared\ memory\ per\ SM}{shared\ memory\ per\ thread\ block}\Big\} \times thread\ Block\ Size. \quad (2)$$

As a result, increasing occupancy requires usage of register file and shared memory to be balanced. In addition, excessive usage of either storage may result in an decrease of occupancy. All current generations of GPU have a much larger per SM register file than shared memory. For each thread block, the data partition should therefore be biased towards registers. However, for circular queues implemented using shared memory only, data are allocated on shared memory. In this case, the occupancy is determined by shared memory usage with many on-chip registers being left unused.

Besides, shared memory takes extra load and store instructions to access. Putting a circular queue data structure entirely in shared memory is therefore not efficient.

2.2.2 Limited Shared Memory Space

GPU's per SM shared memory is only 48 KB even for newly released Fermi[9] and 16KB for all older generations. Shared memory is shared by all active threads on an SM at a given time. It is recommended that at least 192 threads be simultaneously on an SM[9], and thus, for each thread block, the number of time steps, T, cannot be very large, which probably prevents us from obtaining the best performance.

In summary, implementing a circular queue using shared memory only is not efficient. Instead, we should consider leveraging both registers and shared memory.

2.3 Challenges of Implementing Circular Queue in Register File

If we are to move part or all of circular queue from shared memory to registers, the following challenges must be addressed:

• Registers and shared memory are both precious resources. Thus, how can we utilize both effectively?

• Circular queues require storage reuse, which is easy to implement with pointer arithmetics. However, for registers, which are not indirectly addressable, how should we implement a circular queue using registers?

• Stencils require inter-thread communication. If we put some data in registers, how should we handle the data exchange?

• How do we find the best partition of data between registers and shared memory?

We will provide our solutions to address all these issues below.

3 Scalar-Based Circular Queue on GPU

To exploit the potential of registers for circular queue, we must shift from array-based storage to scalar-based storage. In this section, we propose scalar-based circular queue. These scalars variables (SVs) can either reside in registers or shared memory. Two issues need to be figured out for this purpose: 1) The circular queue should be implemented with the least number of SVs; 2) The SVs should be reused explicitly.

To cover the these two issues, we designed the method of reuse pattern. We first discuss it for 2D stencils with each thread computing only one point in the X-dimension, and extend it to coarser thread granularity and higher dimensions later. We refer to the dimension along which the computation moves as the moving dimension. In the 2D case, the moving dimension is the Y-dimension while for the 3D case it is the Z-dimension. We define H as the halo size (the number of neighborhood elements needed in each direction to compute 1 point[8]) of the moving dimension. We assume the halo size along the Y-axis positive and negative directions are the same.

For a two-dimensional stencil with halo size H, to compute T time steps in a row, intuitively, $(2 \times H + 1) \times T$ SVs are needed, where $2 \times H + 1$ SVs are allocated for each time step. Actually, there are two cases where SVs can be reused: 1) The oldest value to compute a point is no longer used after computation, and the SV can be used to hold a newly computed value; 2) The final result is immediately written back to global memory, and thus the SV can be used to hold the value loaded from global memory of the next skewed column. The reuse reduces the number of SVs to $2 \times H \times T + 1$.

We name the SV holding the newly loaded value *BaseSV*, shown as circled node 0 in Fig.3(a)-1. We could start from *BaseSV*, allocate SVs to values as in Fig.3(a)-1: in each level, if the SV holding a value is SV_r, then, the SV holding the value to its left is:

$$SV_l = SV_{((r-1+(2\times H\times T+1)) \bmod (2\times H\times T+1))}. \quad (3)$$

After $2 \times H \times T + 1$ skewed columns (in the case of

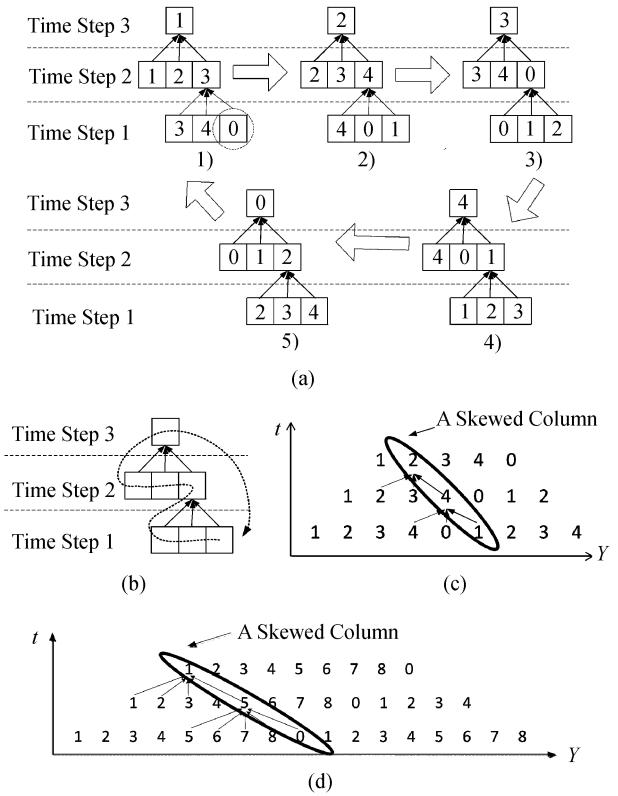

Fig.3. SV reuse and reuse pattern for $T = 2$ and $H = 1$. (a) and (b) shows how SVs are used. (c) summarizes the reuse pattern for $T = 2$ and $H = 1$. (d) shows the reuse pattern for $T = 2$ and $H = 2$.

Fig.3(a), five skewed columns), the allocation pattern goes back to Fig.3(a)-1, forming a repetitive allocation pattern. We call this repetitive allocation pattern the steady state of the reuse pattern. The reuse of SVs are reflected in that: 1) The leftmost value and the newly computed value in upper level are assigned the same SV; 2) The final result and the newly loaded value of the next skewed column are assigned the same SV. For example, final result of Fig.3(a)-5 and the newly loaded value of Fig.3(a)-1 are assigned to the same SV-node 0. Fig.3(b) shows how each SV iterates over the steady state, and Fig.3(c) summarizes the reuse pattern when $T = 2$ and $H = 1$. Note that, while SVs iterate over the steady state, the time steps of the values they are holding also change, as shown in Figs. 3(a) and 3(b). Fig. 3(d) shows the reuse pattern of a 2D 5-point stencil with bigger halo size. For stencils with bigger halo size, skewed columns are skewed more towards the horizontal line, and steady states are longer.

The prolog contains $2 \times H \times T$ partial skewed columns, which can be divided into T consecutive groups, each containing $2 \times H$ partial skewed columns of the same length. Each partial skewed columns in the first group only contains a load from global memory, each column in the next group contains one load followed by one computation, etc. Skewed columns in the next group has one more computation than those in the previous group.

From the above analysis, we could summarize a reuse pattern for a stencil given T and H as follows.

• The steady state of the pattern involves exactly $2 \times H \times T + 1$ skewed columns for the pattern to repeat, which means the loop body contains $2 \times H \times T + 1$ skewed columns.

• The prolog of the pattern involves exactly $2 \times H \times T$ partial skewed columns. The length of i-th partial skewed columns is $\lceil i/(2 \times H) \rceil$.

• To compute a new value stored in SV_k, the SVs needed are SV_k, $SV_{(k+1) \bmod (2 \times H \times T+1)}$, $\cdots$, $SV_{(k+2 \times H) \bmod (2 \times H \times T+1)}$.

• The SV holding the next value to compute in the skewed column immediately after the computation of SV_k is:

$$SV_{next} = SV_{((k-2 \times H+(2 \times H \times T+1)) \bmod (2 \times H \times T+1))}. \quad (4)$$

Once the SV used in the first partial skewed column of the prolog is set, we could derive the reuse pattern. Given T and H, code can be automatically generated with the above rules. Some of the SVs will reside in registers, while the rest reside in shared memory, which will be addressed in Subsection 5.1.

It is easy to extend to cases with coarser thread granularity. If each thread computes k points in the X-direction, the k points can be viewed as a super-SV, which can be treated as one plain SV in our method. Therefore we need to allocate $2 \times H \times T + 1$ super-SVs, with each super-SV consisting of k points, resulting in a total of $k \times (2 \times H \times T + 1)$ SVs. Extending to stencils with higher dimensions is similar. For example, in a 3D 7-point stencil, each thread operates on small 2D tiles of data of size $m \times n$, we allocate $m \times n \times (2 \times H \times T + 1)$ SVs, which are organized as $m \times n$ super-SVs with each consisting of $2 \times H \times T + 1$ points. Inside each super-SV, computations of different plain SVs can be performed in any order, since Jacobi-like stencils have no order constraints. The reuse pattern can extend to cases where halo size along Y-axis positive and negative directions are different. In this case, we just need to replace $2 \times H$ in the above formulae with $H_p + H_n$.

4 Inter-thread Communication

Within a thread block, each thread needs data from neighboring threads to perform a stencil computation. In this section, we discuss how to exchange data through shared memory, and how to reuse shared memory space. Again, we start from 2D stencils with one

point per thread, and then we extend it to more general cases.

In Fig.4(a), we divide a stencil pattern into three areas: left, right and self. Each point has an index for the Y-dimension. In the case that each thread only computes one point, points in the left and right parts together reflect what data are needed by adjacent threads. Y-indices of these points form a subset, which indicate data to be broadcasted. We divide stencils into two categories according to the size of the subset needed, as shown in Figs. 4(b) and 4(c):

• Type A: all the points in the left and right areas have the same Y-index;

• Type B: points in the left and right areas have different Y-index.

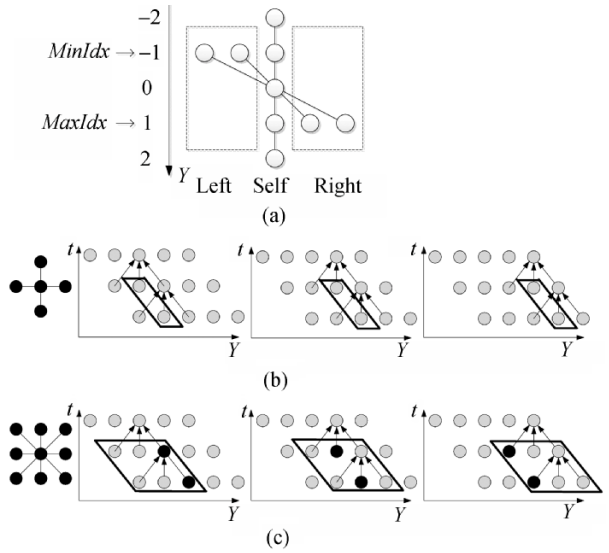

Fig.4. Our division of stencil types. (a) shows our anatomy of a 2D stencil patterns. (b) and (c) use 2D 5-point and 2D 9-point stencils as examples respectively to demonstrate Type A and Type B stencils we defined. Each circle represents a loaded or computed data. The parallelogram boxes show data needed by adjacent threads to compute the skewed column indicated by arrows. In (c), as the dark circles show, each data is used by adjacent threads 3 times.

The difference between these two types is that for Type B, as computation moves forward, the data broadcasted has temporal reuse in the next few skewed columns, while for Type A, it is only used once. For Type B, we define *NIdx* as the number of different Y-indices in the left area and the right area, *MaxIdx* and *MinIdx* as the biggest and smallest among these indices. Note that between *MaxIdx* and *MinIdx*, there may be indices that do not appear in the left and right parts.

For Type A, to compute a skewed column, each thread needs to broadcast T values to its adjacent threads. Thus, each thread needs no more than T locations for communication. To use less shared memory space, we could allocate $\lceil T/RD \rceil$ locations ($1 < RD \leqslant T$), in which case communication space is reused within a skewed column. We name this scheme single column reuse (SCR), and RD as reuse degree of communication space. The bigger RD is, the less shared memory space for communication we need, which is good for occupancy. However, since this storage reuse introduces write-after-read hazard, extra synchronizations are needed, which brings overhead.

For Type B, similar to Type A, we could adopt SCR, allocating $\lceil T/RD \rceil \times NIdx$ locations ($1 \leqslant RD \leqslant T$). In this manner, each thread needs to write *NIdx* values for each single computation. Alternatively, we could exploit the temporal reuse, so that each value only needs to be written once, instead of *NIdx* times for SCR. We propose multiple column reuse (MCR), in which once a value is written to shared memory for communication, it is not overwritten until it is not used anymore. For each single stencil computation, data in the range between *MaxIdx* and *MinIdx* should be kept in shared memory. To save space, locations holding values not used anymore are immediately reused to hold newly broadcasted value. The reuse of communication locations follow the reuse pattern with parameter $H_p + H_n$ being equal to $MaxIdx + MinIdx$. Thus, $(MaxIdx - MinIdx) \times T + 1$ locations are needed. When $MaxIdx + MinIdx$ does not equal $2 \times H$, reuse pattern for SVs and communication spaces are different, and thus have different length of steady state. We may need to unroll the original code to their least common multiple.

To compare SCR and MCR, we consider shared memory writes and space requirement. For Type B, the number of shared memory writes of MCR is only $1/NIdx$ of the number of SCR. As of space requirement, for a specific stencil, the space requirement of MCR is a fixed value while the space requirement of SCR can be reduced by increasing RD, and thus potentially has higher occupancy. Particularly when $NIdx < MaxIdx - MinIdx + 1$, MCR has bigger disadvantage in space requirement. It is expected that when occupancy plays a more important role than instruction count, SCR will outperform MCR and vice versa.

When *MaxIdx* equals H, newly loaded or computed values are needed in the next computation along the same skewed column. In this case, new values must be broadcasted immediately after its generation. In other cases, shared memory writes can be clustered together before the computation of a skewed column (or a section of a skewed column when $RD > 1$ for SCR).

For cases with higher thread granularity, where each thread computes several points in non-moving dimensions, it is possible that data of some points are only needed by points computed by the same thread, and thus not needed by other threads. We need to find out

the subset of Y-indices that are only needed by other threads. As a result, points within a thread may have different subsets and some points may not need to broadcast data to other threads. For 3D stencils, we need to analyze adjacent threads from both X- and Y-dimensions to decide the subset of needed data, and the rest is similar.

5 Framework for Circular Queue on GPUs

In this section, we describe our framework that automatically generates code and searches for the best partition of SVs between register and shared memory. Our iterative search engine aims at finding the best partition between registers and shared memory as well as communication scheme for given thread block size and T.

For readability, we summarize denotations that will be used:

T: the number of time steps calculated between one load and store.

H: the number of neighborhood elements along the moving dimension needed in each direction to compute one point.

N_R: the number of SVs allocated in registers.

N_S: the number of SVs allocated in shared memory.

5.1 Code Generation

A script which takes H, T, RD and N_R as input is used for code generation. Listing 2 shows the generated code for a 2D 5-point stencil with $H = 1$, $T = 2$, each thread computing one point, with four SVs in registers and one in shared memory. A complete code consists of four parts: declaration (lines 7∼9), prolog (lines 17∼31), steady state (lines 33∼78) and epilog (lines 79∼100).

Listing 2. Code of 2D 5-point Stencil with $T = 2$, $RD = 1$, $H = 1$, and Each Thread Computing one Point in the X-direction

```
1   #define TILE_T 2
2   #define TILE_X (BLK_SIZE -2* TILE_T)
3   #define N_IN_SHARED_MEM 1
4   #define N_FOR_COM 2
5   #define PADDING ((N_IN_SHARED_MEM + N_FOR_COM + \
6           1)%2 ? 0:1)
7   float a1, a2, a3, a4;
8   __shared__ float
9       sh_array[THREAD_BLOCK_SIZE+2][N_IN_SHARED_MEM+
10                                   N_FOR_COM+1+PADDING];
11  int start = (TILE_X * blockIdx.x) & (16 - 1),
12      end = start + TILE_X;
13  int offset = TILE_T - start;
14  int tag = start - TILE_T * 2;
15  int tx = threadIdx.x;
16          ...
17  LOAD_DATA (a1);
18
19  LOAD_DATA (a2);
20
21  LOAD_DATA (a3);
22  __syncthreads();
23  WRITE_SHARED_MEM (a2, 1)
24    __syncthreads();
25  DO_STENCIL(a1, a1, 1, a2, a3);
26
27  LOAD_DATA(a4);
28  __syncthreads();
29  WRITE_SHARED_MEM (a3, 1)
30  __syncthreads();
31  DO_STENCIL(a2, a2, 1, a3, a4);
32
33  for (y = 0; y + 2*TILE_T+1< y_max; y+= 2*TILE_T+1) {
34      LOAD_DATA(sh_array[tx + 1][0]);
35      __syncthreads();
36      WRITE_SHARED_MEM(a4, 1);
37      WRITE_SHARED_MEM(a2, 2);
38      __syncthreads();
39      DO_STENCIL(a3, a3, 1, a4, sh_array[tx + 1][0]);
40      DO_STENCIL(a1, a1, 2, a2, a3);
41      STORE_DATA(a1);
42
43      LOAD_DATA(a1);
44      __syncthreads();
45      WRITE_SHARED_MEM(a3, 1)
46      __syncthreads();
47      DO_STENCIL(a4, a4, 0, sh_array[tx + 1][0], a1);
48      DO_STENCIL(a2, a2, 1, a3, a4);
49      STORE_DATA(a2);
50
51      LOAD_DATA(a2);
52      __syncthreads();
53      WRITE_SHARED_MEM(a1, 1);
54      WRITE_SHARED_MEM(a4, 2);
55      __syncthreads();
56      DO_STENCIL(sh_array[tx + 1][0], sh_array[tx + 1][0],
57                                  1, a1, a2);
58      DO_STENCIL(a3, a3, 2, a4, sh_array[tx + 1][0]);
59      STORE_DATA(a3);
60
61      LOAD_DATA(a3);
62      __syncthreads();
63      WRITE_SHARED_MEM(a2, 1)
64      __syncthreads();
65      DO_STENCIL(a1, a1, 1, a2, a3);
66      DO_STENCIL(a4, a4, 0, sh_array[tx + 1][0], a1);
67      STORE_DATA(a4);
68
69      LOAD_DATA(a4);
70      __syncthreads();
71      WRITE_SHARED_MEM(a3, 1);
72      WRITE_SHARED_MEM(a1, 2);
73      __syncthreads();
74      DO_STENCIL(a2, a2, 1, a3, a4);
75      DO_STENCIL(sh_array[tx + 1][0], sh_array[tx + 1][0],
76                                  2, a1, a2);
77      STORE_DATA(sh_array[tx + 1][0]);
78  }
79  epilog = y_max % (2 * TILE_T + 1);
80  if (epilog == 0)
81      goto endoffunc;
82  LOAD_DATA(sh_array[tx + 1][0]);
83    __syncthreads();
84  WRITE_SHARED_MEM(a4, 1); WRITE_SHARED_MEM(a2, 2);
85    __syncthreads();
86  DO_STENCIL(a3, a3, 1, a4, sh_array[tx + 1][0]);
87  DO_STENCIL(a1, a1, 2, a2, a3);
88  STORE_DATA(a1);
89
90  if (epilog == 1)
91      goto endoffunc;
92  //omitted
93
94  if (epilog == 2)
95      goto endoffunc;
96  //omitted
97
98  if (epilog == 3)
99      goto endoffunc;
100 //omitted
101 endoffunc:
```

Given N_R, we assign the N_R SVs with highest SV numbers to registers. These N_R SVs are declared as automatic variables in the form aX, where X is the SV number (line 7), while the rest are declared aggregately as a shared memory array with SV numbers being array indices. We allocate a two-dimensional shared memory array (for 3D stencil, it would be a 3D array) to accommodate both SVs, communication locations and an

extra space for global memory coalescing (lines 8∼9), and each thread owns one row of it. The row length of the array is padded to odd numbers (lines 5∼6) to avoid bank conflict.

As explained in the previous section, the length of prolog is $2 \times H \times T$, each of which computes a partial skewed column, and the steady state contains $2 \times H \times T + 1$ skewed columns. The epilog handles arbitrary tile size in the slowest changing dimension. In Listing 2, due to limited space, we only show epilog code when $epilog = 1$. With the reuse pattern, the script is able to derive SV numbers involved in each load, computation or store. According to SV number, the script either emits an automatic variable name or a shared memory array element.

For SCR, when RD is bigger than 1, the generated code for a skewed column is strip-mined into several sections, with each section does $\lceil T/RD \rceil$ stencils (except the last section of a skewed column). In each section, data needed by other threads for the following computations are first written to the array, guarded by synchronization statements (lines 35∼38, for example), then computations are performed with data written by adjacent threads (lines 39∼40). Our script examines SV number, and only emits write statements for SVs in registers. For SVs in shared memory, they will be accessed through the array indices of their original storage locations (line 66).

Listing 3. Definition of Macros

```
1  #define LOAD_DATA(a)    { \
2        a = tex1Dfetch(texRef, addr); \
3        addr += ROW_LENGTH_OF_GRID; \
4  }
5  #define STORE_DATA(a)    { \
6        __syncthreads(); \
7        sh_array[tx + 1][N_IN_SHARED_MEM+ \
8              N_FOR_COM+PADDING] = a; \
9        __syncthreads(); \
10       if (threadIdx.x >= start && \
11            threadIdx.x < end ) \
12           (*p_dst) = sh_array[tx + 1 + offset][\
13           N_IN_SHARED_MEM+N_FOR_COM+PADDING]; \
14       if (tag > 0) { \
15           if (threadIdx.x < tag)\
16               *(p_dst+BLK_SIZE) = sh_array[tx + 1+\
17               offset+BLK_SIZE][N_IN_SHARED_
18                    MEM+\ N_FOR_COM+PADDING]; \
19       }\
20       p_dst += ROW_LENGTH_OF_GRID;\
21 }
22 #define WRITE_SHARED_MEM(a, index)  {\
23       sh_array[tx + 1][index] = a;\
24 }
25 #define DO_STENCIL(b, upper, mid_idx, middle, lower) { \
26       b = (upper + sh_array[tx][mid_idx] + middle +\
27           sh_array[tx + 2][mid_idx] + lower)*\
28                (float)0.2;\
29 }
```

The script uses macros as templates for code generation. The macros used are defined in Listing 3. *LOAD_DATA* and *STORE_DATA* load data from and store data back to global memory respectively. *WRITE_SHARED_MEM* writes values in registers to shared memory for communication. The *DO_STENCIL* macro is defined by users to describe how to compute each point of a thread. Except *mid_idx*, all other arguments of this macro are SVs (either in registers or in shared memory) generated by the script. Users use *mid_idx* to index into other threads' data in the shared memory. For 2D stencils, *sh_array* is 2-dimensional, and each thread stores its broadcasted data in the $sh\_array[tx + HALO]$ sub-array. For 3D stencils, *sh_array* is 2-dimensional, and each thread stores its broadcasted data in the $sh\_array[ty + HALO][tx + HALO]$ sub-array. Data of adjacent threads can be accessed by changing offsets of the highest or the highest two dimensions for 2D and 3D stencils respectively.

Global memory access must be coalesced[9]. Stencil computations need ghost zones to perform computation. Assuming that there are m threads in the X dimension for each thread block and each thread computes one point, then only the inner $m - 2 \times T$ threads write the final result. When $m-2\times T$ is not a multiple of 16, the alignment condition of coalescing is not satisfied. To make writes aligned, the script generates code that calculates an offset (line 13 of Listing 2). When writing global memory, threads first write data to shared memory, then a selected subset of threads, determined by the offset, read other threads' data with this offset, and write them to global memory (lines 7∼13 of Listing 3). There are cases where the offset is too big and the subset does not have enough threads to store all the results, thus, the script generates additional global memory writes, guarded by if statement, for them (lines 14∼19 of Listing 3). Global memory reads have the same alignment issue, but we could bind the global memory array to a texture[9], and use the read-only texture memory to easily solve this problem (lines 1∼4 of Listing 3).

In this work we do not take boundary conditions into account. For a stencil with halo size H and time blocking factor T, we add $H \times T$ elements to both ends of each dimension. For example, for an $N \times N$ domain, we actually use an array of size $(N+2\times H\times T)\times(H+2\times H\times T)$, in which only the inner $N \times N$ sub-array corresponds to points to be calculated, and the outside are used as ghost zones for blocks at domain boundary. As a result, boundary blocks have the same access pattern as inner blocks. We do this to simplify code generation. Boundary condition handling is orthogonal to the proposed reuse pattern and data partition as reuse pattern and data partition deals with data stored on-chip. Special handling can be added to fix global memory array indices for boundary points. Thus, the proposed method will still work with boundary condition handling code.

5.2 Search for the Best Parameters

As shown in Subsection 3.2, the best partition between registers and shared memory is the result of the complex dynamics among occupancy, synchronization overhead and shared memory access overhead. For example, for two code variants, one with slightly higher occupancy but less data in registers, the other with lower occupancy but more data in registers, it is hard to decide which one is better without running. Similarly, two code variants differing both in RD and occupancy may not be compared directly either. Thus, we resort to running to find the best variant among those not directly comparable. An alternative would be to use analytical performance models to help decide. However, models with higher accuracy than those proposed in [11-12] are required. Luckily, RD and N_R are known during code generation, and occupancy can be obtained after compilation. Thus, only a few variants need to be evaluated through running.

For a stencil code with given parameters of T and thread block size, our framework works as follows.

Firstly, we select candidates for each RD value respectively. We iterate RD from 1 to T. For each RD value, we start from the version that N_S is 0, and iteratively move one SV from register to shared memory. Each version is compiled to get registers and shared memory requirement, which are then fed to CUDA Occupancy Calculator[10] to compute occupancy. During the process, whenever an SV movement causes occupancy to increase, we mark the new version as a candidate, as this version has higher occupancy than previous ones, and among those with the same occupancy, it has the most SVs in register, and thus has the least instruction count among them. The iteration stops until occupancy drops or all SVs are in shared memory, which eliminates code variants with occupancy lower than the starting version. We also include the shared memory version as a candidate, because it needs no communication space, which may have smaller shared memory requirement, and thus higher occupancy than hybrid versions. For Type B, we also add the version with MSR scheme as a candidate.

Secondly, among candidates with different RD values, we eliminate versions known to perform worse than others. We group all candidates according to their occupancy. In each group, we compare each pair of candidates. Note that, candidates in the same group have different RD values. If the code variant with bigger RD value does not have less data in register, then we eliminate it from the candidate set. Therefore, for each RD value, at most one code variant remains.

Finally, we run the remaining code variants, and select the outperforming one under certain T and thread block size.

The process is illustrated in Fig.5.

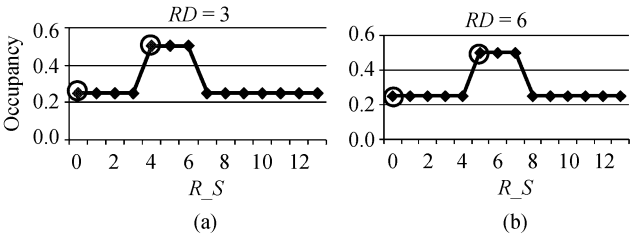

Fig.5. Illustration of the iterative search process. (a) and (b) show how occupancy changes according to R_S for RD being 3 and 6 respectively. 2D 5-point stencil is used, with $T = 6$ on 9800GX2. Code variants in black circles are selected by step 2. The iteration stops when R_S turns 7 and 8 respectively for (a) and (b). In step 3, the two version with $RD = 6$ are eliminated, because compared to corresponding versions with $RD = 3$, they have either the same or less data in registers, which will result in worse performance.

5.3 Discussion

According to the pruning process, the number of candidates left to run is not more than the product of number of different occupancy values and RD values. Since ranges of both quantities are small, we do not expect many candidates are left for the third step. In reality, as our experiment results show, the number of candidates that require running is very small.

It is important to guarantee that the pruned space still contains the point with optimal performance. Since our code generator is on the source code level, interaction with compiler optimizations must be considered. Our pruning process is based on the assumption that, if occupancy stays the same, moving one SV from register to shared memory or increasing RD would impair performance as they increase dynamic instruction count. However, it is not possible to predict how these source-level behaviors would impact compiler optimizations such as instruction selection/scheduling. It is theoretically possible that moving one SV to shared memory causes performance to increase slightly although occupancy stays the same. These anomalies will not keep us from finding very near-optimal points because: 1) If occupancy stays the same, such modification as moving one SV to shared memory will only make tiny differences in performance; 2) By moving SVs to shared memory or increasing RD, the overall trend is that added instructions cause performance to drop. Local anomalies will not compromise the total performance trend. Thus, even if we miss the optimal point, we still will capture points with near-optimal performance.

6 Experimental Results

In this section, we present the experiment results. This section could be divided into 4 parts. In the first part, we fix the value of T, and see how the proposed techniques affect performance under different T values. In the second part, we incorporate T into discussion. For a certain kind of stencil on GPUs, the best T value changes after the application of our techniques. Thus, the purpose of the second part is to evaluate overall performance improvement for a certain kind of stencil. In the third part, we present the results on the 3D 19-point stencil, and compare our results with those presented in Paulius Micikevicius's paper[22] in order to find difference between our implementations. In the last part, we present our findings related to compiler's register allocation, and suggest further refinement. We tested 4 different kinds of stencils: 2D 5-point, 2D 9-point, 2D 13-point (halo size=3), and 3D 7-point stencils, as shown in Fig.6. Our evaluation is based on three different generations of GPUs, with the configurations shown in Table 1 (9800GX2 contains two GPUs, we only use one of them). All the programs are compiled by `nvcc` under `-O3` option, and CUDA Profiler is used to measure execution time and dynamic instruction count. Register and share memory requirements are obtained from the .cubin file generated by `nvcc`. We only measured the execution time on GPU, and CPU-GPU transfer time is excluded.

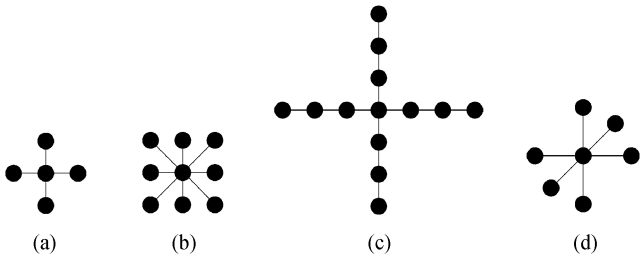

Fig.6. Stencils we used in our experiments. (a) 2D 5-point stencil. (b) 2D 9-point stencil. (c) 2D 13-point stencil. (d) 3D 7-point stencil.

For 2D stencils, we set the tile size in Y-direction 500 points, and tile size in Z-direction 100 points for 3D stencils, so that redundant computation is not too big due to ghost zones[8]. The shared memory versions used $2 \times H \times T + 1$ elements per thread for data storage, and no space for communication is needed. We set the thread granularity in the X-direction to 1 in our study.

6.1 Performance Improvement for Individual T Values

Except 2D 9-point stencil, we present three versions: the shared_mem version with all SVs in shared memory, the hyb_reg version with all SVs in registers and no communication space reuse, which is the start point of our search, and the hyb_search version, which is the best version through search. For 2D 9-point stencil, we used both SCR and MCR. The thread block sizes for the 4 stencils are 192, 192, 256 and 512 (32×16) respectively.

Figs. 7∼9 show the occupancy, dynamic instruction count and performance of various versions under different T values. The instruction counts are normalized to the instruction count of reg versions, and for 2D 9-point stencil, they are normalized to the MCR_reg versions. Table 2 summarizes speedups of search versions over shared-mem and reg versions. For 2D 9-point stencils, data before slashes show speedups of MCR_search versions over shared-mem versions and MCR_search versions over MCR_reg versions while data after slashes show speedups of SCR_search versions over shared-mem versions and SCR_search versions over SCR_reg versions. The circular queue's on-chip resource consumption is proportional to T. Thus, when T increases, the occupancy curves present ladder form fall. As shown in Fig.7, our method enables bigger T values, and for each T value, our search versions provide higher occupancies. Our method generally makes occupancy decrease slower when T increases. Reg versions have all SVs in registers, which eliminates the instructions to access shared memory during computation. Thus, they have the least instruction count. Search versions may have more instructions than reg versions when RD is bigger than 1 or N_S is bigger than 0.

For 2D 5-point stencil, on GTX 285, the search version outperforms the reg version when T is equal to and greater than 5. Actually, when $T = 4$, the search version has higher occupancy, but no performance advantage, which shows that the occupancy level of the reg version is high enough to hide latency. Occupancy plays a more important role when its value is small, which explains the big gap between the search and the reg version when $9 \leqslant T \leqslant 16$. 9800GX2 has a smaller

Table 1. Machine Configuration

	Peak Arithmetic Performance (GFLOPS)	Global Memory Bandwidth (GB/s)	No. SMs	Max No. Threads per SM	Shared Memory per SM (KB)	No. 32-bit Registers per SM	Driver Version	NVCC Version
9800GX2	384	64	16	768	16	8192	190.18	2.3
GTX285	709	141	30	1024	16	16384	190.53	2.3
C2050	1030	144	14	1536	48	32768	260.19	3.2

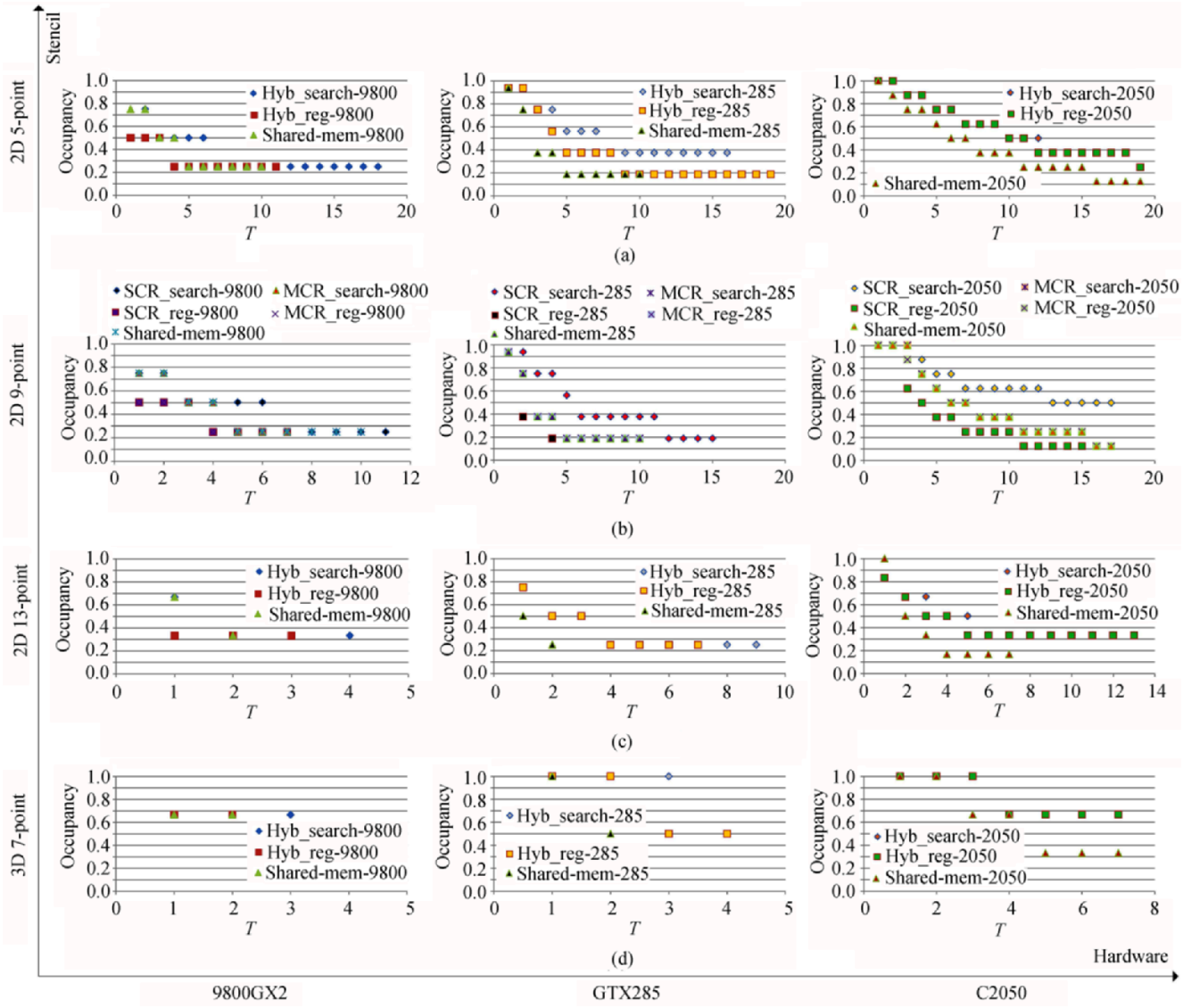

Fig.7. Occupancy of the four stencils on all platforms.

Table 2. Performance Speedups for Individual T Values

	Search/Shared-mem			Search/Reg		
	9800GX2	GTX285	C2050	9800GX2	GTX285	C2050
2D 5-point	1-1.72	1.04-2.10	1-2.93	1-1.59	1-1.42	1-1.11
2D 9-point	1-1.45/1-1.15	1-1.59/1-1.13	1-2.51/1-1.12	1-1.39/1-1.49	1-1.65/1	1-2.6/1-1.01
2D 13-point	1.06-1.08	1.36-1.71	1-2.29	1-1.43	1	1-1.02
3D 7-point	1-1.03	1.11-1.44	1-2.05	1	1	1

register file, thus, the reg version even performs worse than the shared-mem version when $T = 4$, because of the unbalanced usage of register file and shared memory. Tesla C2050's register file and shared memory is significantly larger than its previous generations, and thus, all three versions are able to sustain high occupancy levels even for big T values. The ratio of register and shared memory requirement of the reg version is close to C2050's hardware ratio, and the search version is not able to further improve occupancy. As a result, the curves of the two versions almost overlap. The same happens for 2D 13-point and 3D 7-point stencils on C2050.

For 2D 9-point stencil, on all three machines, MCR search versions have the same occupancies as shared-mem versions, but have less SVs in shared memory than shared-mem versions. Thus, their performance gap shows the effect of instruction reduction. Similar to 2D

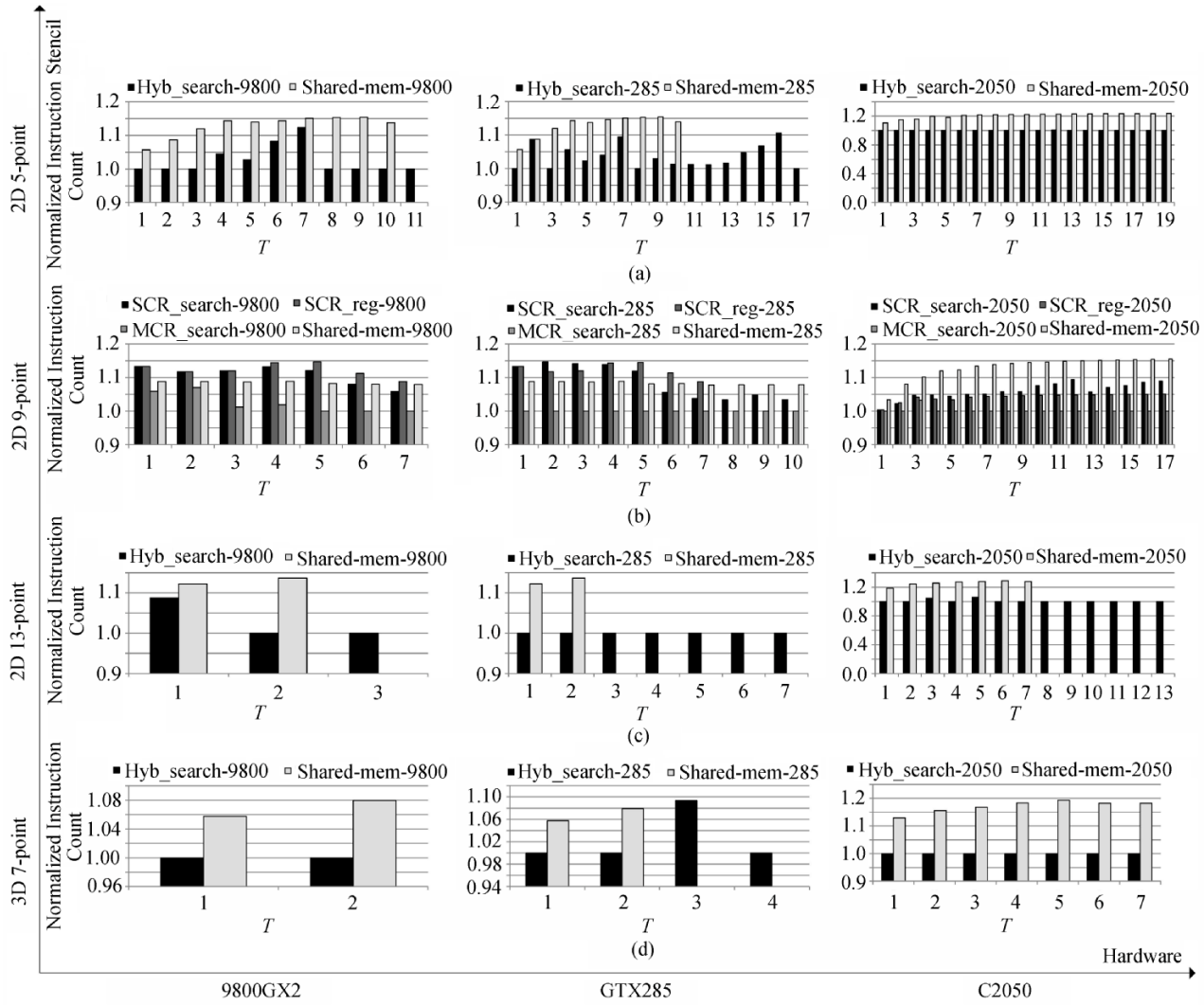

Fig.8. Dynamic instruction count of the four stencils on all platforms, normalized to the reg versions. For 2D 9-point stencil, normalized to MCR_reg version.

5-point stencil, due to the smaller register file, the MCR reg version on 9800GX2 has lower occupancies than the shared-mem version and thus performs worse. The SCR versions have more instructions than MCR because they cannot take advantage of the reuse of broadcasted data and thus have more shared memory writes for communication. They even have more instructions than the shared-mem version, because the shared-mem version can exploit the reuse. The performance difference of SCR search and MCR search versions is the result of a combat between occupancy and instruction reduction. When T is small, occupancy is less significant and thus the MCR_search version wins as having less shared memory writes. For bigger T values, occupancy is more important and the SCR_search version beats the MCR_search. SCR-reg versions need a large amount of shared memory to store the broadcasted data of a skewed column, even more than shared-mem versions and thus they perform the worst.

For stencils with big halo sizes such as 2D 13-point stencil, number of SVs grows faster with T. As a result, value range of T is smaller. The occupancy is determined by register usage and reusing communication space and partition hardly find any chance to move enough data from registers to shared memory to increase occupancy. Therefore, most of the time, search versions and reg versions are the same. On 9800GX2, due to a smaller register file and the large storage requirement, even the most balanced version cannot improve occupancy.

3D 7-point stencils have small range of T because we choose the thread block size of 512. With such a thread block size, when T increases, storage requirement quickly exceeds the hardware capacity, which makes code variant not runable. For 3D stencils, the ratio of redundant computation increases faster with T

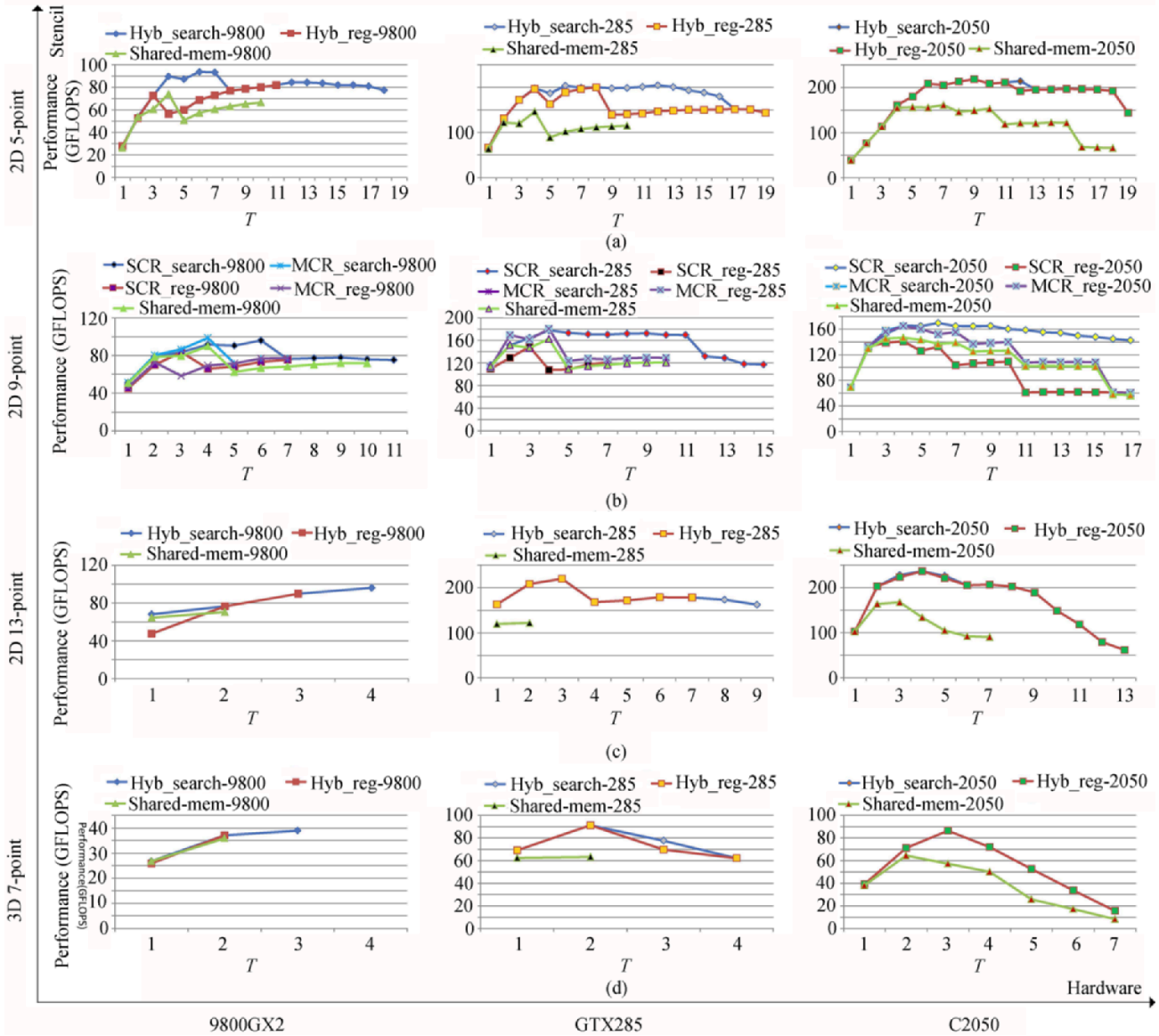

Fig.9. Performance of the four stencils on all platforms.

than 2D stencils and thus a big thread block size is preferred to increase tile size[8]. 9800GX2 can only run at most 768 threads per SM and thus with a thread block size of 512, there is no chance to improve occupancy. The limited speedup when $T = 2$ comes from instruction reduction.

Table 3 shows the effectiveness of the search space pruning. For 9800GX2, T values used for the four stencils are 10, 10, 3, 3 respectively while for the rest two platforms, T values are 15, 15, 8, 4. Thread block sizes used are 192, 192, 256 and 512 (32×16) respectively. As illustrated in the table, only a very small amount of code variants are actually run to find the best one, which shows that our pruning scheme is very effective. For all the four stencils on three platforms, our framework succeeded in capturing the optimal points. Table 4 shows total and average time spent on code generation, compilation and running of the four stencils on C2050 with the above T value and thread block sizes. The CPU of the C2050 system is Intel Xeon E5506 at 2.13 GHz and the main memory is of size 4 GB. Code generation and compilation account for most of the search time. This is due to the fact that: 1) only a small portion of generated and compiled code variants

Table 3. Search Space Pruning

	2D 5-point	2D 9-point	2D 13-point	3D 7-point
9800GX2	2/84	3/76	2/24	3/5
GTX285	3/94	3/182	3/25	2/9
C2050	5/193	6/341	2/166	3/31

Note: In the form a/b, a is the number of candidates actually run in the 3rd step, and b is the total number of runable code variants in the search space.

Table 4. Search Time Breakdown on Tesla C2050

	2D 5-point	2D 9-point	2D 13-point	3D 7-point
Time (seconds)	371(4.9)/743(9.9)/27(5.4)	667(6.2)/4 359(40.7)/34(5.7)	629(7.8)/2 291(28.3)/12(6)	8(0.4)/38(2)/11(3.7)

Note: In the form a(x)/b(y)/c(z), a, b and c are the total time spent on generation of source code, compilation, and running respectively. Numbers in parenthesis show average time spent on generation of source code, compilation and running for each code variant. Running time includes the time of running a kernel for 10 times and CPU-GPU data transfer time.

need running; 2) the average time spent on code generation or compilation is on par or longer than the average running time. The number of code variants generated and compiled are 75, 107, 81 and 19 for the four stencils respectively. The size of the loop body is linear to T and thus when T is big, code size is big. Furthermore, the epilog almost doubles the code size. All these cause the code generation and compilation to slow down. The average code generation and compilation time is small for 3D 7-point stencil because a small T is used. To reduce overall search time, possible solutions are: 1) Reduce the number of code variants that are generated and compiled. For example, moving two SVs from registers to shared memory at a time instead of one or developing models to rule out code variants that are unlikely to have high occupancy. Both ways have the risk of missing optimal configurations. 2) Speed up code generation by implementing it with more efficient languages such as C instead of shell scripts we are using now. As for compilation time, it is currently out of our control. However, if inline assembly is allowed, we could generate code with inline assembly directly, which we believe could save some compilation time. We will consider reducing search time in our future work.

6.2 Interaction With T

In this subsection, we show how the proposed techniques interact with T and compare performance under each versions' best T values. As discussed in [8], the best T value is a tradeoff between global memory traffic reduction and redundant computation brought in. On GPUs, performance is also affected by occupancy. Decrease of occupancy may create sudden drops in the performance curve, which makes the actual best T value smaller. The search versions are also affected by communication space reuse and data partition, which impacts synchronization frequency and instruction count. When T increases, to prevent occupancy falling, RD and N_S tend to increase, and may hurt performance. When occupancy stays the same, performance of reg versions no longer increases from $T = 18$ and $T = 2$ for GTX285 in Figs. 9(a) and 9(b), indicating the theoretical best T value for the two cases. For the rest cases, the theoretical best T values are not yet reached until the curves of reg versions end. 3D stencils have smaller theoretical best T values because redundancy grows faster[8].

Due to the occupancy change, our search versions are able to reach bigger T values with higher performance originally unreachable, such as the cases of 2D 13-point stencil on all three platforms and 3D 7-point stencil on 9800GX2, as shown in Figs. 9(c) and 9(d). In the rest cases, the best T values either stay the same while performance increases or the best T values itself increase.

For each stencil, we tried thread block sizes of 64, 128, 192, 256, 320, 384, 448, and 512. Fig.10 summarizes the speedups of our search versions over shared mem versions under their own best T values.

Table 5 and Table 6 show the parameters of our best

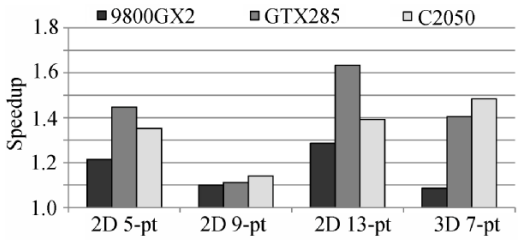

Fig.10. Speedups of best search versions over best shared-mem versions for the four stencils on all platforms. (pt: point)

Table 5. Parameters of the Balanced Versions for 9800GX2, GTX285 and C2050

	9800GX2				GTX285				C2050			
	2D 5-pt	2D 9-pt	2D 13-pt	3D 7-pt	2D 5-pt	2D 9-pt	2D 13-pt	3D 7-pt	2D 5-pt	2D 9-pt	2D 13-pt	3D 7-pt
Thread Block Size	192	192	128	32×16	256	192	256	32×16	192	192	256	32×32
T	6	4	3	3	9	4	3	2	9	6	4	4
Total Number of SVs	13	9	19	7	19	9	19	5	19	13	25	9
SVs in Register	9	7	19	6	18	9	19	5	19	13	25	9
RD	2	MCR	1	1	2	MCR	1	1	1	3	1	1
Total Regs per Thread	21	20	31	16	31	24	31	14	34	27	40	22
Shared Mem (B)	7 020	7 020	2 712	12 276	7 252	7 012	5 264	7 372	8 536	5 432	5 240	23 120
Occupancy	0.5	0.5	0.333	0.667	0.5	0.375	0.5	1	0.625	0.75	0.5	0.667
Reg/Shared mem	2.30	2.19	5.85	2.67	4.38	2.63	6.03	3.89	3.06	4.50	7.80	3.90
Performance (GFLOPS)	93.6	98.6	99.3	39.0	211.0	180.5	219.6	91.3	218.1	169.6	236.3	95.7

Note: pt-point.

Table 6. Parameters of the Best Shared memory Based Versions for 9800GX2, GTX285 and C2050

	9800GX2				GTX285				C2050			
	2D 5-pt	2D 9-pt	2D 13-pt	3D 7-pt	2D 5-pt	2D 9-pt	2D 13-pt	3D 7-pt	2D 5-pt	2D 9-pt	2D 13-pt	3D 7-pt
Thread Block Size	128	192	128	32×16	192	192	128	16×20	256	256	192	32×16
T	7	4	2	2	4	4	2	2	7	4	3	2
Total Number of SVs	15	9	13	5	9	9	13	5	15	9	19	5
Regs per Thread	12	12	12	13	12	12	12	13	34	15	20	18
Shared Mem (B)	7 836	7 020	7 000	12 276	7 012	7 012	6 992	7 948	15 480	9 288	15 048	12 240
Occupancy	0.333	0.5	0.333	0.667	0.375	0.375	0.25	0.625	0.5	0.833	0.375	1
Reg/Shared mem	0.78	1.31	0.88	2.16	1.31	1.31	0.88	2.09	2.25	1.65	1.02	3.01
Performance (GFLOPS)	77.1	89.7	77.2	35.9	145.8	162.5	134.5	64.9	161.3	148.7	169.8	64.5

Note: pt-point.

versions and the best shared memory only versions respectively. For 2D 9-point stencil on GTX285, MCR has the best performance, while on 9800GX2, best performance comes with SCR-1. For 3D stencils, thread block sizes are expressed as $X \times Y$, in which X and Y are dimensionalities of X and Y directions. The maximum numbers of threads per SM are different on all the three platforms, and therefore, the same occupancy on the two platforms indicates different number of active threads on an SM.

6.3 Comparison with Paulius Micikevicius's Results

Paulius Micikevicius presented results of some 3D difference computation on GPUs[22]. His work focused on 3D stencils with big halo sizes. He also combined the registers and shared memory in the implementation of circular queue but only explored the cases when $T = 1$. In this subsection, we present experimental results of 3D 19-point stencil and compare our results with Micikevicius's results in the hope of finding something interesting.

There are two differences between our implementations. 1) In Paulius's implementation, all threads in a thread block compute a final result (they only implemented the $T = 1$ case), and data of the ghost zones are loaded by threads near thread block boundaries. In our implementation, however, only the inner $(BLK\_X - HALO \times T) \times (BLK\_Y - HALO \times T)$ threads of a thread block compute final results. Thus, the ratio of redundant computation of our implementation is substantially higher than that of Paulius's implementation. 2) Paulius's implementation involves register-to-register copies when moving along the moving-dimension while in our implementation, these copies are eliminated through reuse pattern. Table 7 shows the performance of the four versions of 3D 19-point stencil on Tesla C2050: Paulius's version, the shared memory version, our search version and a modified implementation of the search version, in which the inner $(BLK\_X - HALO \times (T-1)) \times (BLK\_Y - HALO \times (T-1))$ in a thread block compute final results. The search version ends up the same as the register version. When $T = 1$, our modified search version has the same ratio of redundant computation as Paulius's version. Thread block sizes are 32×16 and T is 1 for all the four versions. We substitute global memory reads in Paulius's version with texture loads so as to make the comparison fair, which improves the performance by 2 GFLOPS. Both Paulius's version and our modified search version greatly outperform our original search version due to their relatively lower ratio of redundant computation. When $T = 1$, our modified search version slightly outperforms Paulius's, which we believe is due to the elimination of register-to-register copies. When $T = 2$, performance of both our original and modified search versions drop greatly because the reduction in memory traffic does not compensate the performance loss brought by the increased ratio of redundant computation.

Table 7. Performance Results of 3D 19-Point Stencil on C2050

	Micikevicius'	Shared Memory	Search	Search (modified)
$T = 1$	100.1	49.9	70.4	106.0
$T = 2$		17.1	37.4	73.2

Note: Performance comparison of four versions of the 3D 19-point stencil (halo size = 3) on Tesla C2050. Thread block size is 32×16 for all four versions. Performance is measured in GFLOPS.

We can observe great performance improvement of our search version over the shared memory version. The occupancies of the shared mem version are 0.667 and 0.333 respectively when $T = 1$ and $T = 2$ while the occupancies of the search version are 1.0 and 0.667, which is major reason of the performance improvement. Excessive shared memory requirement causes the low occupancies of the shared memory version. Each thread needs a shared memory space of $2 \times HALO \times T + 1$ elements. When $HALO$ is big, it requires large shared memory space while some registers are not utilized. The searcg version (in this case, also the register version), however, is able to achieve a more balanced usage of shared memory and registers.

6.4 Interaction with Register Allocation

The reuse pattern we proposed with minimum

number of SVs requires that an SV be assigned the same hardware register for all of its live ranges. Otherwise, more hardware registers are needed than the minimum. With decuda[13], a third party disassembler, we find that the register allocation is not exactly what we expected, and thus more registers are used. The register allocation part of the tool-chain is not open-sourced, nor do we have control over it. However, we believe it is completely feasible for a compiler to incorporate such a feature as remembering the different live ranges of the same variable in the source code, and assigning them the same hardware register during register allocation. In such a way, the proposed reuse pattern can reach its minimum register requirement.

7 Related Work

Occupancy and Shared Memory/Register Pressure Related GPU Optimizations. A lot of research has tried to increase occupancy to improve performance. [14] used a method based on graph coloring to reuse shared memory space. It focuses on making different user-declared shared memory arrays use the same shared memory partition. Since the data structure of circular queue is usually declared as one array, it will find no optimization opportunity for circular queue. [15] proposed to apply loop fission to loops with multiple independent statements and split kernel with if-else statement into two kernels, where one kernel only executes the if branch and the other only executes the else branch, so as to reduce register pressure. The latter is also adopted by [16]. [17] manually spilled some register variables into shared memory for FFT. Their adjustment of register pressure is ad-hoc, and not as systematic as ours. [18] and [19] separated a complex kernel into several simple kernels, which brings higher occupancy, due to less resource requirement of each simplified kernel. These techniques are either application-specific, or not suitable for stencils.

Temporal Blocking for Iterative Stencil Computations. A lot of research has addressed the issue of data reuse across multiple time steps for iterative stencil computations. In [1-2], Wonnacott proposed time skewing to take advantage of temporal reuse by tiling both space and time dimensions. Song and Li proposed similar techniques in [4]. [3] proposed a cache-oblivious algorithm to compute stencil computations for the same purpose of temporal reuse. [6] proposed circular queue, which uses a special data structure to hold temporary values. [7] implemented circular queue using the local store of the CBEA. Our work is based on circular queue, and extends it to registers.

Stencil on GPU. [8] proposed a model to determine the best ghost zone size for stencil computations on GPU and a related transformation framework. Their work suggested very small best T values for 2D and 3D stencils, because the tile size of their implementation is limited by on-chip storage. By adopting circular queue, tile size can be enlarged, and thus, our experiment resulted in bigger best T values. Their model cannot be applied directly to circular queue, because when T changes, storage requirement changes too, which may result in a change in occupancy. Their model did not take occupancy into consideration. [20] proposed to loose the synchronization constraints for stencils on GPU. Unlike our work, theirs is an alteration of algorithm. [6, 21-22] all adopted circular queue method, but they only explored the case when $T = 1$ for GPU. Besides, [21] only used shared memory. Although [6, 12] combined register file and shared memory, they did not attempt to balance the usage of the two storages as we do, and they both used $3 \times H \times T$ locations for storage. [23] proposed a new parallel algorithm for computing SSOR, a different kind of stencils than we do. Besides, their work was on the algorithmic level.

[6, 24] proposed an autotuning framework for stencils on various platforms including GPU. Our tuning of register and shared memory usage complements their work.

[25] presented new way of tiling as well as ILP-level tuning on both CPUs and GPUs. Our work shared a lot of similarities, but focuses on different aspects. Our work focuses on economizing and balancing on-chip storages. Such techniques as using only $2T + 1$ SVs, communication space reuse and search for the best partition of data were not used in their work. Similarly, we did not tune ILP as they did. Apart from ILP, their implementation is similar to our reg versions, which our search versions outperform in some cases.

Optimization Space Pruning. Search is a powerful method to find the optimization combination, sequence and parameters of the best performing version. However, exploring the full optimization space incurs prohibitive overhead. Lots of research has been done to prune the search space. On traditional platforms, some use analytical model to eliminate unnecessary candidates, such as [26-29]. These works focus on a collection of traditional compiler optimizations, and use a model to reduce search candidates. Some adopt method from artificial intelligence to guide search process, such as [30-33]. On a massively parallel architecture such as GPU, [34] proposed utilization and efficiency as metrics and combined them to prune optimization space. Their method is general and requires that global memory bandwidth is not the bottleneck for performance. In [35], a new compressed format is proposed for sparse matrix on GPU, and search is needed to determine certain parameters of the format. They proposed an

analytical model specific to SpMV to eliminate search candidates. Our iterative search algorithm is designed specifically for the circular queue method on GPU. It tries to find a balance between proposed technique of register/shared memory partition and communication space reusing. It prunes the search space by leveraging knowledge specific to GPUs and our proposed techniques.

Other Related Work. [36] proposed modulo variable expansion (MVE), which assigns multiple registers to a variable in a loop, to eliminate anti- and output-dependence in registers across successive iterations for software pipelining. Both our works used unrolling to expose a reuse pattern. However, MVE was proposed to eliminate anti- and output-dependence. Our work, on the other hand, leverages this reuse pattern to minimize storage requirement. Besides, MVE is an instruction-level compiler optimization, while our work is a source code level optimization that is well aware of the data flow of the algorithm and specific to circular queue. [37] proposed scalar replacement to keep some subscripted variables in registers to expose reuse. Our work tries to move data originally in lower level of memory into registers. However, our work is fundamentally different in that scalar replacement does not change where a variable is originally stored. It just keeps a copy in a register. Our work is more of a data partition problem that may change the storage location of a variable.

8 Conclusions and Future Work

In this paper, we addressed the issue of balancing the usage of registers and shared-memory for the circular queue method on GPUs. We proposed a reuse pattern that enables the usage of registers in the circular queue method. Also, we proposed a framework that can generate circular queue code for stencils on GPUs, and can search for the best data partition. The four different kinds of stencils we studied showed up to 2.93X speedup over shared-memory based versions. For future work, we will 1) develop a model to guide the search process to reduce search time, and 2) extend our discussion of on-chip storage allocation on GPUs to a broader range of applications, and develop a more general framework to help programmers optimize their code.

References

[1] Wonnacott D. Achieving scalable locality with time skewing. *Int. J. Parallel Program*, 2002, 30(3): 181-221.

[2] Mccalpin J, Wonnacott D. Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Department of Computer Science, Rugers University. 1999.

[3] Strzodka R, Shaheen M, Pajak D *et al.* Cache oblivious parallelograms in iterative stencil computations. In *Proc. the 24th ACM Int. Conf. Supercomputing*, Tsukuba, Japan, Jun. 1-4, 2010, pp.49-59.

[4] Song Y, Li Z. New tiling techniques to improve cache temporal locality. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, USA, May 1-4, 1999, pp.215-228.

[5] Jin G, Mellor-Crummey J, Fowler R. Increasing temporal locality with skewing and recursive blocking. In *Proc. ACM/IEEE Conference on Supercomputing*, Denver, USA, Nov. 10-16, 2001, pp.43-43.

[6] Datta K, Murphy M, Volkov V *et al.* Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proc. ACM/IEEE Conference on Supercomputing*, Austin, USA, Nov.15-21, 2008, Article 4.

[7] Williams S, Shalf J, Oliker L *et al.* Scientific computing Kernels on the cell processor. *Int. J. Parallel Program*, 2007, 35(3): 263-298.

[8] Meng J, Skadron K. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proc. the 23rd International Conference on Supercomputing*, Yorktown Heights, USA, Jun. 8-12, 2009, pp.256-265.

[9] NVIDIA. NVIDIA CUDA programming guide 3.0, http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide-pdf, 2010.

[10] NVIDIA Corp. CUDA Occupancy Calculator, 2010.

[11] Hong S, Kim H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. the 36th Annual Int. Symp. Computer Architecture*, Austin, USA, Jun. 20-24, 2009, pp.152-163.

[12] Baghsorkhi S S, Delahaye M, Patel S J, Gropp W D, Hwu W W. An adaptive performance modeling tool for GPU architectures. In *Proc. the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 9-14, 2010, pp.105-114.

[13] van der Laan W J. Decuda. http://wiki.github.com/laanwj/decuda/, Sept., 2010.

[14] Moazeni M, Bui A, Sarrafzadeh M. A memory optimization technique for software-managed scratchpad memory in GPUs. In *Proc. the 7th IEEE Symposium on Application Specific Processors*, San Francisco, USA, Jul. 27-28, 2009, pp.43-49.

[15] Carrillo S, Siegel J, Li X. A control-structure splitting optimization for GPGPU. In *Proc. the 6th ACM Conf. Computing Frontiers*, Ischia, Italy, May 18-20, 2009, pp.147-150.

[16] Park S J, Ross J, Shires D, Richie D, Henz B, Nguyen L. Hybrid core acceleration of UWB SIRE radar signal processing. *IEEE Trans. Parallel Distrib. Syst*, 2011, 22(1): 46-57.

[17] Gu L, Li X, Siegel J. An empirically tuned 2D and 3D FFT library on CUDA GPU. In *Proc. the 24th ACM Int. Conf. Supercomputing*, Tsukuba, Japan, Jun. 1-4, 2010, pp.305-314.

[18] Goorts P, Rogmans S, Bekaert P. Optimal data distribution for versatile finite impulse response filtering on next-generation graphics hardware using CUDA. In *Proc. the 15th International Conference on Parallel and Distributed Systems*, Shenzhen, China, Dec. 9-11, 2009, pp.300-307.

[19] Bailey P, Myre J, Walsh S D C, Lilja D J, Saar M O. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In *Proc. International Conference on Parallel Processing*, Vienna, Austria, Sep. 22-25, 2009, pp.550-557.

[20] Venkatasubramanian S, Vuduc R W. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proc. the 23rd International Conference on Supercomputing*, Yorktown Heights, USA, Jun. 8-12, 2009, pp.244-255.

[21] Christen M, Schenk O, Neufeld E *et al.* Parallel data-locality aware stencil computations on modern micro-architectures. In *Proc. IEEE Int. Symp. Parallel & Distributed Processing*, Rome, Italy, May 23-29, 2009, pp.1-10.

[22] Micikevicius P. 3D finite difference computation on GPUs using CUDA. In *Proc. the 2nd Workshop on General Purpose Processing on Graphics Processing Units*, Washington, USA, Mar. 8, 2009, pp.79-84.

[23] Di P, Wan Q, Zhang X *et al.* Toward harnessing DOACROSS parallelism for multi-GPGPUs. In *Proc. the 39th Int. Conf. Parallel Processing*, San Diego, USA, Sep. 13-16, 2010, pp.40-50.

[24] Christen M, Schenk O, Burkhart H. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proc. IEEE International Parallel & Distributed Processing Symposium*, Anchorage, USA, May 16-20, 2011, pp.676-687.

[25] Nguyen A, Satish N, Chhugani J, Kim C, Dubey P. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proc. ACM/IEEE Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, New Orleans, USA, Nov. 13-19, 2010, pp.1-13.

[26] Qasem A, Kennedy K. Profitable loop fusion and tiling using model-driven empirical search. In *Proc. the 20th Annual International Conference on Supercomputing*, Cairns, Australia, Jun. 28-Jul. 1, 2006, pp.249-258.

[27] Knijnenburg P M W, Kisuki T, Gallivan K *et al.* The effect of cache models on iterative compilation for combined tiling and unrolling: Research articles. *Concurrency and Computation: Practice & Experience*, 2004, 16(2-3): 247-270.

[28] Yotov K, Li X, Ren G *et al.* A comparison of empirical and model-driven optimization. In *Proc. the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, USA, Jun. 8-11, 2003, pp.63-76.

[29] Chen C, Chame J, Hall M. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proc. Int. Symp. Code Generation and Optimization*, San Jose, USA, Mar. 20-23, 2005, pp.111-122.

[30] Epshteyn A, Garzaran M, DeJong G *et al.* Analytical models and empirical search: A hybrid approach to code optimization. In *Proc. the 18th International Workshop on Languages and Compilers for Parallel Computing*, Hawthorne, USA, Oct. 20-22, 2005, pp.259-273.

[31] Agakov F, Bonilla E, Cavazos J *et al.* Using machine learning to focus iterative optimization. In *Proc. Int. Symp. Code Generation and Optimization*, New York, USA, Mar. 26-29, 2006, pp.295-305.

[32] Almagor L, Cooper K D, Grosul A, Harvey T J, Reeves S W, Subramanian D, Torczon L, Waterman T. Finding effective compilation sequences. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, Washington, USA, Jun. 11-13, 2004, pp.231-239.

[33] Vaswani K, Thazhuthaveetil M J, Srikant Y N *et al.* Microarchitecture sensitive empirical models for compiler optimizations. In *Proc. Int, Symp, Code Generation and Optimization*, San Jose, USA, Mar. 11-14, 2007, pp.131-143.

[34] Ryoo S, Rodrigues C I, Stone S S *et al.* Program optimization space pruning for a multithreaded gpu. In *Proc. the 6th Annual IEEE/ACM Int. Symp. Code Generation and Optimization*, Boston, USA, Apr. 6-9, 2008, pp.195-204.

[35] Choi J W, Singh A, Vuduc R W. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proc. the 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 9-14, 2010, pp.115-126.

[36] Lam M. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, USA, Jun. 20-24, 1988, pp.318-328.

[37] Callahan D, Carr S, Kennedy K. Improving register allocation for subscripted variables. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, White Plains, USA, Jun. 20-22, 1990, pp.53-65.