Li J, Liu L, Wu Y *et al.* Pragma directed shared memory centric optimizations on GPUs. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 31(2): 235–252 Mar. 2016. DOI 10.1007/s11390-016-1624-8

Pragma Directed Shared Memory Centric Optimizations on GPUs

Jing Li [1,2], *Member, CCF*, Lei Liu [1], *Member, CCF*, Yuan Wu [3], Xiang-Hua Liu [3], Yi Gao [3]
Xiao-Bing Feng [1], *Member, CCF, ACM, IEEE*, and Cheng-Yong Wu [1], *Senior Member, CCF, Member, ACM*

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

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

[3] *Beijing Samsung Telecom Research and Development Center, Beijing 100028, China*

E-mail: {lijing01, liulei}@ict.ac.cn; {yuan002.wu, xianghua.liu, yi1980.gao}@samsung.com; {fxb, cwu}@ict.ac.cn

Received January 4, 2015; revised August 25, 2015.

Abstract GPUs become a ubiquitous choice as coprocessors since they have excellent ability in concurrent processing. In GPU architecture, shared memory plays a very important role in system performance as it can largely improve bandwidth utilization and accelerate memory operations. However, even for affine GPU applications that contain regular access patterns, optimizing for shared memory is not an easy work. It often requires programmer expertise and nontrivial parameter selection. Improper shared memory usage might even underutilize GPU resource. Even using state-of-the-art high level programming models (e.g., OpenACC and OpenHMPP), it is still hard to utilize shared memory since they lack inherent support in describing shared memory optimization and selecting suitable parameters, let alone maintaining high resource utilization. Targeting higher productivity for affine applications, we propose a data centric way to shared memory optimization on GPU. We design a pragma extension on OpenACC so as to convey data management hints of programmers to compiler. Meanwhile, we devise a compiler framework to automatically select optimal parameters for shared arrays, using the polyhedral model. We further propose optimization techniques to expose higher memory and instruction level parallelism. The experimental results show that our shared memory centric approaches effectively improve the performance of five typical GPU applications across four widely used platforms by 3.7x on average, and do not burden programmers with lots of pragmas.

Keywords GPU, shared memory, pragma directed, data centric

1 Introduction

The proliferation of GPUs has been witnessed over the past few years for their high processing power. However, the raw computation power of GPUs is often underutilized by slow accesses to off-chip global memory. In contrast, shared memory, which locates inside GPU chip, offers much faster data access with terabyte level throughput and cycle level latency. Acting as a software controlled cache, shared memory facilitates huge performance gain in numerous applications[1-4].

Efficient management of shared memory in GPU is a significant yet challenging problem, as it depends on programmers' knowledge and experiences, which are difficult and impractical to automate. Even for affine loops with relatively regular memory access patterns (which are typical in GPU applications), programmers often suffer from "headache" when trying to optimize for shared memory. With a large number of candidate arrays which exceed shared memory capacity, smart choices need to be made on which arrays and how many array elements are most suitable to load into shared memory. Things will go even worse if shared memory usage triggers bank conflicts and limits parallel thread blocks (TBs), both of which will cause poor resource utilization.

To address the programing challenges for typical GPU applications with affine loops, we should consider the following key factors.

Regular Paper

This work was supported by the National High Technology Research and Development 863 Program of China under Grant No. 2012AA010902, the National Natural Science Foundation of China (NSFC) under Grant No. 61432018, and the Innovation Research Group of NSFC under Grant No. 61221062.

First of all, targeting higher programming productivity, we need a simple way to convey programmer knowledge and experience to compiler, which then takes over and automatically optimizes for shared memory. Taking advice from programmers, pragma directed programming model (e.g., OpenACC① and OpenHMPP②) has been introduced to simplify GPU programming. However, none of these models provides convenient solution to utilize shared memory. *Cache* pragma in OpenACC does not target non-coalescing global accesses, which is a huge performance loss. Besides, it is not the best choice to specify reused elements of each thread as *cache* does, since arrays in shared memory are allocated and shared by each TB. In contrast, OpenHMPP regards shared memory as an advanced optimization, and does not provide additional support for either reuse or coalescing in shared memory. Thus, developers are responsible for managing data between shared and global memory, which is as complex as CUDA/OpenCL programming. From previous analysis, we can conclude that data management hints should include reuse as well as coalescing, and should be mapped to the working set of a TB. Programmers are in the right position to provide this information as arranging parallelism already provides a clear image on data access patterns.

Secondly, in order to relieve programmers from complex parameter selection, we should devise a framework to develop array correlations and choose shared array related parameters automatically. This framework should consider both data reuse and coalescing (see CUDA C Programming Guide③ for more details) to obtain optimal solutions in terms of off-chip traffic. Most of current studies ask programmers to do the work: they expect programmers to explicitly identify all subarrays with reuse potentials, which sometimes might be inconvenient due to complex array access functions. OpenHMPP further requires programmers to infer shared array sizes. Though OpenACC develops shared array sizes automatically, it cannot handle huge shared memory demand, which is bad for parallelism. Therefore, it will be a great relief for developers if the compiler can induce candidate arrays and shared array size automatically.

Moreover, GPU resources, especially memory bandwidth and compute cores, should be carefully managed to avoid underutilization. Bank conflict and partition camping are the most common reasons that serialize memory requests and waste high GPU bandwidth. Previous studies[1-2,5-8] manage to avoid memory level conflicts from different aspects, but none of them provides solutions in all cases. Improving instruction level parallelism (ILP) is believed to be an effective solution to keep GPU cores busy[9-11]. However, few compilers have managed to automatically expose ILP of GPU kernels, not to mention determining the suitable amount of ILP efficiently. In addition, the timing of optimizing for ILP should be evaluated carefully. Since without shared memory optimization, increasing ILP of kernels with inefficient access patterns only aggravates memory stall and degrades performance. Subsection 3.3.2 on ILP and shared memory discusses this in detail.

With the existing factors in mind, our solution to improve shared memory utilization is data centric pragmas and a supporting compiler framework. We make the following contributions. 1) We design a set of pragmas that can convey data management hints of programmers including advice on data partition and desired memory patterns. All candidate arrays related to user pragmas will be analyzed for coalescing and reuse opportunities. It is worth noting that our pragmas provide optimization suggestions from a data centric point of view, which makes our design superior to current pragma directed solutions. 2) We utilize the polyhedral model to choose appropriate arrays and array partition sizes in shared memory. These choices come down to an optimization problem, which targets minimum off-chip traffic. 3) We further explore memory and instruction level parallelism automatically to put GPU resources into full play. These advanced optimizations eliminate bank and channel conflicts and expose parallel instructions, according to underlying architecture.

By extending our pragma to OpenACC, we evaluate our approach with five typical benchmarks across four widely used platforms (e.g., NVIDIA GTX 690, AMD HD7850). The experimental results show that we can achieve an average of 3.7x performance improvement with one simple pragma.

We first explore our data centric pragmas designed for shared memory in Section 2. Section 3 illustrates corresponding compiler framework based on polyhedral model for shared memory utilization, along with advanced optimizations targeting memory and instruction level parallelism. The experimental methodology and

①http://www.openacc-standard.org/, Dec. 2014.

②http://en.wikipedia.org/w/index.php?title=OpenHMPP&oldid=614132944, Oct. 2014.

③http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Dec. 2014.

results are presented in Section 4. Finally, we discuss related work in Section 5 and conclude the paper in Section 6.

2 Shared Memory Oriented Pragmas

An important feature of our work is its capability to convey data management hints of programmers to a compiler. We achieve this feature by shared memory oriented pragmas. In this section, we first present the design of the pragma, and then illustrate its usage by an example before comparing it with mainstream accelerator APIs. We choose to base our approach on OpenACC, which provides good support in extracting parallelism.

2.1 Pragma Design

Our shared memory oriented pragma is designed to describe data partition and optimization hints to assist automatic compiler optimization. Fig.1 shows its syntax written in Backus-Naur Form (BNF). On the whole, a *share* pragma contains a list of array partitions with optimization hints on shared memory.

- sharePragma ::= "#pragma acc share(" arraySpecs")"
- arraySpecs ::= arraySpec {", "arraySpecs}
- arraySpec ::= subArray {":coalesce"}
- subArray ::= arrayName {"["[expr|"∗"]"]"}∗
- expr ::= constant { ["+"| "−"| "×"| "/"] expr }
- constant ::= TB_X | TB_Y | TB_Z | integer

Fig.1. Syntax of shared memory oriented pragma.

Array partitions in *share* pragma represent the working set of a TB. From a data centric point of view, developers describe the desired workload of each TB with array partitions in *share* pragma. In this way, developers can observe and adjust the working set of each TB directly, instead of just a few array elements of each thread. This is reasonable for accelerator-oriented APIs, such as openHMPP, as arrays in shared memory are allocated and accessed in chunks. More importantly, expressing desired memory patterns on a chunk of array is more intuitive than on a few elements.

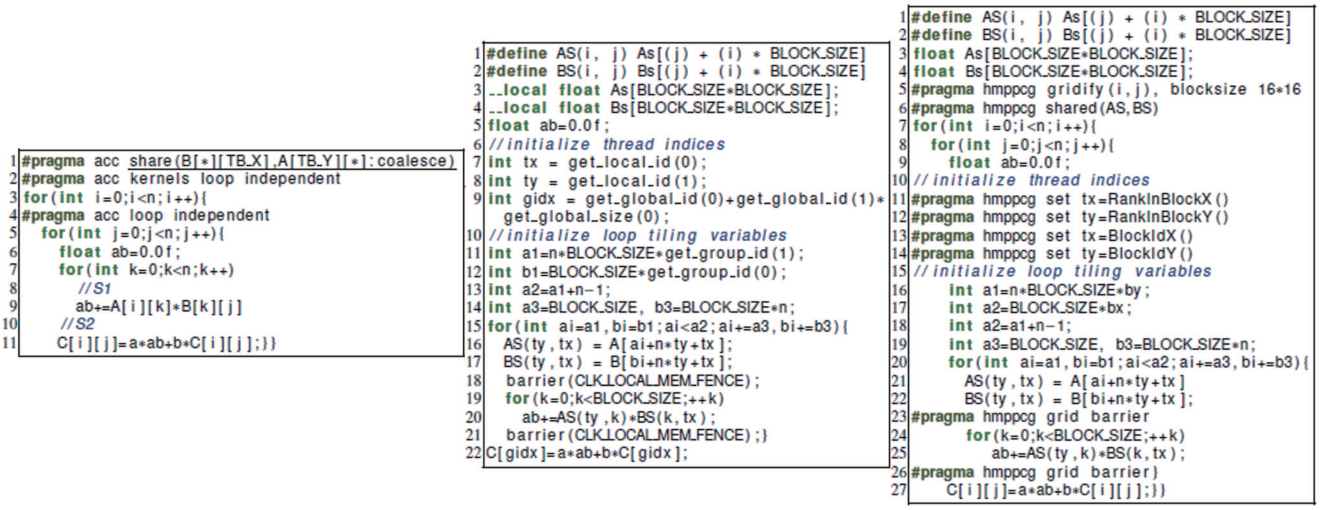
As optimal array partitions depend on several factors (e.g., TB size, problem size, shared memory size), it is challenging for a programmer to provide accurate array partitions. Therefore, we remove this complication for programmers by supporting parametric array partitions in *share*. An array partition is declared by specifying the length on each dimension, which can be a constant number, a TB primitive, an expression or a vague one (represented by "∗"). Constants are positive numbers obtained from previous experience or reuse distance analysis, and they can describe array reuse precisely. Introducing TB primitives (such as TB_X, TB_Y and TB_Z) allows a natural array partitioning since the working set of a TB is usually related to TB size. Expressions are composed of constant numbers and TB primitives, designed to cope with more complex partition plans. The symbol "∗" represents a vague number between zero and the length of the corresponding array dimension, which allows flexible subarray partitioning. Both TB primitives and vague symbol "∗" provide parametric array partitions, which will be determined by our compiler later. It is worth noting that for multiple sections that utilize our pragma, we keep separate copies of all parameters, such as TB primitives, BLOCK_SIZE. Therefore, our compiler framework is able to solve each set of parameters independently, so as to expose maximal potential in all sections.

Programmers can provide *share* pragma without going through much trouble. Having an overall understanding of the target section, programmers can easily identify arrays that are critical to performance. Moreover, based on analysis and experience, programmers can describe the working set of a TB with parametric array partitions, which further simplifies their work.

As for the optimization hints, *share* pragma focuses on two common shared memory utilization scenarios: data reuse and memory coalescing. By default, each array partition in *share* utilizes shared memory to reuse data. They will be read into shared memory and our compiler will ensure maximum reuse on it. Besides, we provide a keyword *coalesce* to deliver global memory coalescing hints to our compiler. When discovering non-coalescing global access, programmers can explicitly require the array partition be loaded into shared memory by adding the keyword *coalesce* to it. Hence, some unnecessary compiler analysis can be avoided with the programmer knowledge on non-coalescing array partitions. It is then left for the compiler to adjust these non-coalescing global references.

2.2 Discussion

To better explain how *share* pragma offers a more convenient way to describe array optimization decisions, we use general matrix multiplication (`gemm`) in Fig.2 as an example.



```
1 #pragma acc share(B[*][TB_X],A[TB_Y][*]:coalesce)
2 #pragma acc kernels loop independent
3 for(int i=0;i<n;i++){
4 #pragma acc loop independent
5   for(int j=0;j<n;j++){
6     float ab=0.0f;
7     for(int k=0;k<n;k++)
8       //S1
9       ab+=A[i][k]*B[k][j]
10    //S2
11    C[i][j]=a*ab+b*C[i][j];}}
```

(a)

```
1 #define AS(i, j) As[(j) + (i) * BLOCK_SIZE]
2 #define BS(i, j) Bs[(j) + (i) * BLOCK_SIZE]
3 __local float As[BLOCK_SIZE*BLOCK_SIZE];
4 __local float Bs[BLOCK_SIZE*BLOCK_SIZE];
5 float ab=0.0f;
6 //initialize thread indices
7 int tx = get_local_id(0);
8 int ty = get_local_id(1);
9 int gidx = get_global_id(0)+get_global_id(1)*
    get_global_size(0);
10 //initialize loop tiling variables
11 int a1=n*BLOCK_SIZE*get_group_id(1);
12 int b1=BLOCK_SIZE*get_group_id(0);
13 int a2=a1+n-1;
14 int a3=BLOCK_SIZE, b3=BLOCK_SIZE*n;
15 for(int ai=a1,bi=b1;ai<a2;ai+=a3,bi+=b3){
16   AS(ty,tx) = A[ai+n*ty+tx];
17   BS(ty,tx) = B[bi+n*ty+tx];
18   barrier(CLK_LOCAL_MEM_FENCE);
19   for(k=0;k<BLOCK_SIZE;++k)
20     ab+=AS(ty,k)*BS(k,tx);
21   barrier(CLK_LOCAL_MEM_FENCE);}
22 C[gidx]=a*ab+b*C[gidx];
```

(b)

```
1 #define AS(i, j) As[(j) + (i) * BLOCK_SIZE]
2 #define BS(i, j) Bs[(j) + (i) * BLOCK_SIZE]
3 float As[BLOCK_SIZE*BLOCK_SIZE];
4 float Bs[BLOCK_SIZE*BLOCK_SIZE];
5 #pragma hmppcg gridify(i,j), blocksize 16*16
6 #pragma hmppcg shared(AS,BS)
7 for(int i=0;i<n;i++){
8   for(int j=0;j<n;j++){
9     float ab=0.0f;
10 // initialize thread indices
11 #pragma hmppcg set tx=RankInBlockX()
12 #pragma hmppcg set ty=RankInBlockY()
13 #pragma hmppcg set tx=BlockIdX()
14 #pragma hmppcg set ty=BlockIdY()
15 // initialize loop tiling variables
16     int a1=n*BLOCK_SIZE*by;
17     int a2=BLOCK_SIZE*bx;
18     int a2=a1+n-1;
19     int a3=BLOCK_SIZE, b3=BLOCK_SIZE*n;
20     for(int ai=a1,bi=b1;ai<a2;ai+=a3,bi+=b3){
21       AS(ty,tx) = A[ai+n*ty+tx];
22       BS(ty,tx) = B[bi+n*ty+tx];
23 #pragma hmppcg grid barrier
24       for(k=0;k<BLOCK_SIZE;++k)
25         ab+=AS(ty,k)*BS(k,tx);
26 #pragma hmppcg grid barrier}
27   C[i][j]=a*ab+b*C[i][j];}}
```

(c)

Fig.2. Code samples of general matrix multiplication using different APIs. (a) OpenACC extended with *share* pragma. (b) Optimized kernel of *share* pragma. (c) OpenHMPP.

To describe our data centric shared memory optimization plan in `gemm`, we add one *share* pragma (line 1 in Fig.2(a)). Being repeatedly read in this loop nest, input arrays $\boldsymbol{A}$ and $\boldsymbol{B}$ are regarded as seed arrays (arrays that are suitable for shared memory optimization). Based on our analysis, both seed arrays contain high order data reuse. Therefore, we partition the working set of TBs with $\boldsymbol{A}[TB_Y][*]$ and $\boldsymbol{B}[*][TB_X]$ to ensure maximal reuse within each TB. In addition, detecting non-coalescing access in array $\boldsymbol{A}$, we add a keyword *coalesce* to inform the compiler. Our compiler then loads array partitions of $\boldsymbol{A}$ into shared memory to avoid non-coalescing references. Fig.2(b) shows the optimized kernel code that our compiler produces. The compiler analysis workflow on `gemm` will be discussed in the next section.

After detailed introduction to our design, we believe it is necessary to compare it with mainstream accelerator APIs. On one hand, *cache* pragma of OpenACC is inferior in expressiveness. First, it does not take global coalescing into consideration. Second, shared memory will not be utilized when reused data exceeds shared memory capacity. Therefore, OpenACC cannot utilize shared memory in matrix multiplication, since *cache* cannot fix non-coalescing reference of $\boldsymbol{A}$ and on chip shared memory is not able to hold the reused data in $\boldsymbol{A}$ and $\boldsymbol{B}$. On the other hand, OpenHMPP programmers are burdened with low level programming details. Thus, programming with OpenHMPP is no less complex than that with CUDA/OpenCL. As shown in Fig.2(c), programmers are responsible of inducing shared array sizes (lines 3 and 4) and copying data into shared memory (lines 20∼22). Besides, programmers need to perform blocking manually to fit all reused data into shared memory (lines 16∼26). Note that the OpenHMPP version is very similar to our optimized kernel, but we do not need complex hand-coding.

3 Shared Memory Centric Optimizing Compiler

3.1 Overview

With data management hints from programmers, our compiler now utilizes this information to generate high performance GPU kernel code. As shown in Fig.3, overall framework of our proposed compiler can be divided to two parts: SM opt. phase in the figure (gray box at the top) is designed to perform polyhedral-based shared memory optimization; advanced opts. phase (gray box at the bottom) carries out advanced optimizations on memory and instruction level parallelism. We elaborate these two processes in Subsection 3.2 and Subsection 3.3 respectively.

We focus on affine loops whose loop bounds and array access functions are affine combinations of outer loop indices and global parameters[12-13]. With this restriction, our compiler is able to analyze access patterns and perform shared memory centric optimizations at compile-time. Though restricted to affine loops, our compiler is still widely applicable as affine loops play a critical role in many computation-intensive pro-

grams, which makes them popular targets to accelerate on GPUs[14].

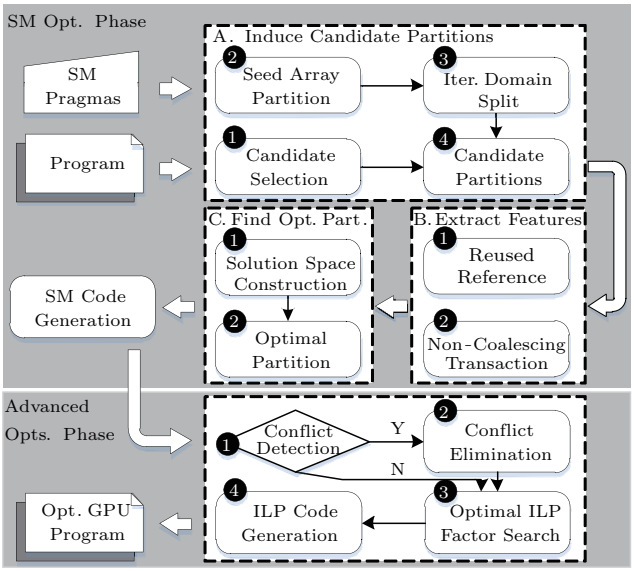


Fig.3. Framework of the proposed compiler.

3.2 Shared Memory Optimization Based on Polyhedral Model

To relieve programmers from exploring all array partition schemes and parameters, we design our polyhedral-based compiler analysis to develop array correlations and select array partitions automatically.

For each combination of an input program and a parameterized *share* pragma, our compiler needs to induce all array partition plans, extracts kernel features (data reuse and coalescing that can be achieved by utilizing shared memory), and finally decides an optimal partition. We use the polyhedral model to assist candidate partition and feature extraction, thanks to its ability to capture array references and iteration domains.

Terminologies. Before discussing our parameter selection, we first introduce some terminologies. Arrays specified by *share* pragma are called seed arrays. In affine loop nests, iteration domain $\boldsymbol{I}$ of a statement can be described as a polytope bounded by loop indexes and global parameters. For each affine array access, we use an access function $\boldsymbol{F}(\boldsymbol{I})$ to map from iteration space $\boldsymbol{I}$ to array data space. In order to parallelize a loop for GPU, we use schedule $\boldsymbol{\phi}(\boldsymbol{I})$ and placement $\boldsymbol{Q}(\boldsymbol{I})$ to assign a new time stamp and an owner processor to each iteration instance. Detailed definitions can be found in [12-13, 15-16].

3.2.1 Inducing Candidate Partitions

As shared memory optimizations are performed on each array partition, obtaining parametric candidate partitions is necessary. To this end, our compiler first selects candidate arrays that might benefit from shared memory in input program. Then, we partition seed arrays according to programmer specification, which is in turn used to split iteration domain. Given the iteration domain splits and candidate arrays, our compiler can induce a set of candidate partitions. The dotted box A in Fig.3 illustrates above workflow.

Candidate Selection. Seed arrays are believed to be the primary focus of programmers, but chances are that there exist other candidate arrays that call for shared memory optimization. Due to two practical considerations, we find it is necessary to induce a final candidate set based on seed arrays. First, programmers might fail to identify all arrays that benefit from shared memory since they do not care about arrays beyond their interests. Second, complex array correlations make it even more difficult to specify array partitions.

Based on the consensus that developers have insight into target programs, a candidate set will be populated with the given seed arrays. The problem of candidate selection can be solved by mapping it to an equivalent problem of finding the transitive closure of seed arrays in an undirected graph, which is created with vertices representing each array reference in the target loop nest and an edge exists between two vertices in presence of a dependence. It is worth noting that only arrays that share dependence with seed arrays in *share* pragma are brought into the candidate set. As far as the programmer is concerned, this candidate set is still the key to the target section. Besides, without *share* pragma, all arrays in the target section would be considered as candidates. Hence, our compiler would take a lot more time to analyze.

Consider `gemm` in Fig.2(a). After applying data dependence analysis, we add array $\boldsymbol{C}$ to the candidate set as it depends on both $\boldsymbol{A}$ and $\boldsymbol{B}$.

Seed Array Partition. Candidate partition is not that straightforward as it can only be induced indirectly through iteration domains. Candidate partitions should conform with seed partitions (they should be accessed in the same iterations), and iteration subspace is the one thing that links them.

We can infer a partition matrix of a seed array from user pragmas such that each row represents a partition hyperplane and the size of each row vector represents the partition length. For example, partition matrices

of array $\boldsymbol{A}$ and $\boldsymbol{B}$ in Fig.2(a) demonstrate parametric partitions of size $TB_Y \times k_1$ and $k_2 \times TB_X$, where k_1, k_2 represent the vague numbers "$*$" that fall between 0 and n:

$$\boldsymbol{P_A} = \begin{pmatrix} TB_Y & 0 \\ 0 & k_1 \end{pmatrix}, \quad \boldsymbol{P_B} = \begin{pmatrix} k_2 & 0 \\ 0 & TB_X \end{pmatrix}.$$

Iteration Domain Split. By reversing affine access functions, we can induce an iteration subspace $\boldsymbol{IS}$ for each combination of seed partition and array reference, which constitutes a set of iteration subspaces *ISSet*. $\boldsymbol{IS_{Akl}}$ is the iteration domain partition induced from the l-th affine reference to array $\boldsymbol{A}$ in the k-th statement.

$$\boldsymbol{IS_{Akl}} = \boldsymbol{F}^{-1}_{\boldsymbol{Akl}}(\boldsymbol{DS_A}), \quad 1 \leqslant l \leqslant p, 1 \leqslant k \leqslant q,$$

where q is the amount of statements that access $\boldsymbol{A}$, p is the total references to $\boldsymbol{A}$ in the k-th statement, $\boldsymbol{F_{Akl}}$ represents the affine access function, and $\boldsymbol{DS_A}$ is the data space of $\boldsymbol{A}$ induced from $\boldsymbol{P_A}$.

Though provided by programmers, seed arrays are not guaranteed to induce identical iteration subspaces. In order to maximize the difference between iteration subspaces of different seeds, we "intersect" all "compatible" iteration subspaces in *ISSet*: two iteration subspaces are considered compatible if 1) they are identical in all dimensions or 2) they differ in a dimension that either subspace utilizes the vague symbol "$*$". Compatible subspaces can be intersected as a new iteration subspace that combines all compatible dimensions.

For example in Fig.2(a), the affine access functions of the references to all arrays are represented as:

$$\boldsymbol{F_{A11}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{I}_1 \\ n \\ 1 \end{pmatrix},$$

$$\boldsymbol{F_{B11}} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{I}_1 \\ n \\ 1 \end{pmatrix},$$

$$\boldsymbol{F_{C11}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{I}_2 \\ n \\ 1 \end{pmatrix},$$

where $\boldsymbol{I}_1 = \begin{pmatrix} i \\ j \\ k \end{pmatrix}$ and $\boldsymbol{I}_2 = \begin{pmatrix} i \\ j \end{pmatrix}$ are the iteration vectors. Therefore, we can easily induce the iteration subspaces $\boldsymbol{IS_A} = i : [0 : n : TB_Y], k : [0 : n : *]$ and $\boldsymbol{IS_B} = j : [0 : n : TB_X], k : [0 : n : *]$ by applying the inverse access functions to seed partitions $\boldsymbol{DS_A}$ and $\boldsymbol{DS_B}$. We can further combine $\boldsymbol{IS_A}$ and $\boldsymbol{IS_B}$ as they are identical on the dimension k and do not interfere with each other on dimension i and dimension j. Intersecting $\boldsymbol{IS_A}$ and $\boldsymbol{IS_B}$ produces a new iteration subspace $\boldsymbol{IS_{AB}} = i : [0 : n : TB_Y], j : [0 : n : TB_X], k : [0 : n : *]$.

Candidate Partition. Given the iteration domain splits and candidate arrays, we can induce candidate partitions for each $\boldsymbol{IS}_m$ in *ISSet*. Take array $\boldsymbol{B}$ for example, we transform the iteration domain split to a candidate partition for each affine reference of $\boldsymbol{B}$, before combining these candidate partitions to a rectangle hull ((1)).

$$\boldsymbol{DS_{B}}_m = \{RectHull(\boldsymbol{F_{B}}_{kl}(\boldsymbol{IS}_m)) \mid 1 \leqslant l \leqslant p, 1 \leqslant k \leqslant q\}. \quad (1)$$

Depending on the number of affine references to the same array, we obtain candidate partition of different shapes. With a single affine reference, as $\boldsymbol{IS}_m$ splits the iteration domain into non-overlapping polytopes, applying an affine transformation on it results in non-overlapping candidate partition; but for multiple affine accesses, we take the rectangle hull of all candidate partitions to obtain a maximal coverage on all partitions. This leads to the redundant copy of array elements in different partitions. For instance, with an iteration domain split $\boldsymbol{IS} = i : [0 : N : 8]$, the candidate partitions for array references $\boldsymbol{B}[i+1]$ and $\boldsymbol{B}[i-1]$ are $\boldsymbol{B}[-1 : N-1 : 8]$ and $\boldsymbol{B}[1 : N : 8]$, respectively. Neighboring rectangle hulls of the candidate partitions (e.g., $\boldsymbol{B}[-1 : 9]$ and $\boldsymbol{B}[8 : 17]$) overlap on two elements. It is worth noting that sharing array elements between partitions does not violate program correctness, as the target section is free of loop carried dependences. Otherwise, it is not suitable for parallel execution on GPU. Therefore, dependences on these duplicated elements can be preserved.

Considering the loop in Fig.2(a), it is now straightforward to partition candidate arrays according to the iteration split $\boldsymbol{IS_{AB}}$. Specifically, we obtain $\boldsymbol{DS'_A} = [0 : n : TB_Y][0 : n : *]$, $\boldsymbol{DS'_B} = [0 : n : *][0 : n : TB_X]$ and $\boldsymbol{DS'_C} = [0 : n : TB_Y][0 : n : TB_X]$ as candidate partitions. Note that variables or TB primitives in partition matrices of seed arrays are regarded as parameters, which will propagate to all candidate partitions.

3.2.2 Extracting Features

Since off-chip accesses are costly, we intend to choose an array partition plan that minimizes the off-chip traffic. With this goal in mind, our compiler

extracts the number of reused references and non-coalescing transactions, as shown in the dotted box labeled B in Fig.3, to compose the maximum number of global references that could be omitted if shared memory were used.

Reused Reference. Reusing data in shared memory contributes to less off-chip traffic. To quantize potential benefit of bringing an array partition into shared memory, we evaluate the volumes of reused references in shared memory, which can be obtained by intersecting data spaces of all references to the same array ((2)) (*Duplicate* returns the number of duplicate points in a polytope). In addition, we utilize (3) to collect candidate arrays with significant data reuse into a *ReuseSet*. A threshold θ_r is introduced to filter candidate arrays with insignificant data reuse (i.e., the amount of reused data is negligible compared with the size of the data subspace), as they not only provide limited benefit for memory reduction, but also complicate our optimal partition search and code generation processes. Based on our experiments and experience, θ_r usually falls between 1.4 and 2, depending on shared memory size on a platform. Currently, we fix θ_r that contributes to optimal performance on each platform, but a more automatic parameter selection will be available in the future.

$$ReuseSize_{\boldsymbol{A}} = Duplicate(\boldsymbol{F}_{\boldsymbol{A}kl}(\boldsymbol{IS}_k)), \quad 1 \leqslant l \leqslant p \wedge 1 \leqslant k \leqslant q, \tag{2}$$

$$ReuseSet = \{\boldsymbol{A} \mid ReuseSize_{\boldsymbol{A}} \geqslant \theta_r \times |\boldsymbol{DS}_{\boldsymbol{A}}|, \boldsymbol{A} \in Candidates\}. \tag{3}$$

Both array $\boldsymbol{A}$ and array $\boldsymbol{B}$ in Fig.2(a) have high order data reuse. As $A[i][k]$ is repeatedly referenced in loop j, the reuse size of $\boldsymbol{A}$ sums up to $TB_Y \times n \times n$, which is n times of its data space size $|\boldsymbol{DS'_A}|$. Similarly, every element in array $\boldsymbol{DS'_B}$ is accessed n times. On the contrary, no reuse exists for $C[i][j]$ as the reuse set is exactly the same size as $\boldsymbol{DS'_C}$.

Non-Coalescing Transaction. Coalescing global transactions through shared memory greatly accelerates off-chip accesses. Therefore, we need to first identify inefficient global references in candidate arrays and then estimate global transactions that can be accelerated by our compiler.

Locating non-coalescing transactions demands a close understanding of the memory access pattern of each statement instance. We thereby borrow the definition of adjacency constraints on schedule and placement from [14].

Time schedule adjacency constraint ((4)) requires two statement instances that access adjacent elements of an array to be executed at the same time instance.

Space placement adjacency constraint ((5)) requires two statement instances that access adjacent elements of an array to be mapped to adjacent processors.

$$\boldsymbol{F}(\boldsymbol{x}_m) + (0 \ldots 1)^{\mathrm{T}} = \boldsymbol{F}(\boldsymbol{y}_m), \boldsymbol{x}_m, \boldsymbol{y}_m \in \boldsymbol{IS}_m,$$

$$\boldsymbol{\phi}(\boldsymbol{x}_m) = \boldsymbol{\phi}(\boldsymbol{y}_m), \tag{4}$$

$$\boldsymbol{Q}(\boldsymbol{x}_m) = \boldsymbol{Q}(\boldsymbol{y}_m). \tag{5}$$

Those array references that violate either constraint fall into *CoalesceSet* ((6)). For each non-coalescing reference in *CoalesceSet*, we obtain the global transactions required by recording unique transaction IDs ((7)). Notice that for arrays inferred by *coalesce* in our pragma, our compiler skips the check for non-coalescing references and directly evaluates this global transaction size.

$$CoalesceSet = \{\boldsymbol{A} \mid \boldsymbol{\phi}(\boldsymbol{x}_m) \neq \boldsymbol{\phi}(\boldsymbol{y}_m) | \boldsymbol{Q}(\boldsymbol{x}_m) \neq \boldsymbol{Q}(\boldsymbol{y}_m)\}, \tag{6}$$

$$CoalesceTran_{\boldsymbol{A}} = \sum_{\boldsymbol{x} \in \boldsymbol{IS}_m} | Unique(\frac{Offset(\boldsymbol{F}_{\boldsymbol{A}}(\boldsymbol{x}))}{TranSize}) |, \tag{7}$$

where *TranSize* is the global transaction size on current platform, *Offset* represents the offset of a reference in global memory, and *Unique* returns a unique base address for each array reference.

As indicated by OpenACC pragmas in Fig.2(a), our compiler induces the following time schedule and space placement of $S1$ and $S2$ on GPU:

$$\boldsymbol{\phi}(\boldsymbol{I}_{S1}) = k \text{ and } \boldsymbol{Q}(\boldsymbol{I}_{S1}) = i \times TB_X + j,$$

$$\boldsymbol{\phi}(\boldsymbol{I}_{S2}) = n \text{ and } \boldsymbol{Q}(\boldsymbol{I}_{S2}) = i \times TB_X + j.$$

Hence, we conclude that $\boldsymbol{A}$ in Fig.2(a) suffers from non-coalescing global reference as it violates the time schedule constraint. For the iteration instances that access adjacent elements in $A[i][k]$ (i.e., $\boldsymbol{x} = (i, j, k)^{\mathrm{T}}$ and $\boldsymbol{y} = (i, j, k+1)^{\mathrm{T}}$), they do not execute at the same time as $\boldsymbol{\phi}(\boldsymbol{x}) = k$ but $\boldsymbol{\phi}(\boldsymbol{y}) = k + 1$. According to (7), for each concurrent access to $A[i][k]$, a global transaction that starts at $\boldsymbol{A} + (i \times n + k) \times 4$ is returned. As a result, a total of $TB_Y \times n$ global transactions are required to read the data space of $\boldsymbol{A}$. However, accesses to $\boldsymbol{B}$ and $\boldsymbol{C}$ comply with both constraints and therefore do not belong to *CoalesceSet*.

3.2.3 Finding Optimal Partition

As presented in the dotted box named C of Fig.3, our compiler takes features of all candidate partition

plans as input, and builds a solution space for each partition plan based on restrictions of GPU resources. By evaluating all solution spaces, we solve an optimal candidate partition plan and corresponding parameters which maximize the benefit of shared memory.

Solution Space Construction. Evaluating a candidate partition plan can be reduced to a confined optimization problem, which targets maximal *Profit* under the resource constraints on GPUs. For each candidate partition plan ($\boldsymbol{IS}_m$), the solution space of this constrained profit maximization problem can be modeled as:

$$\text{max. } Profit = \sum_{\boldsymbol{A}\in ReuseSet} ReuseSize_{\boldsymbol{A}} + TranSize \times \sum_{\boldsymbol{B}\in CoalesceSet} CoalesceTran_{\boldsymbol{B}}, \tag{8}$$

$$\text{s.t.} \sum_{\boldsymbol{A}\in ReuseSet|CoalesceSet} \boldsymbol{DS_A} \leqslant \theta_s \times SMSize, \tag{9}$$

$$TBSize \leqslant TBSizeMax, \tag{10}$$

$$TBSize \times \frac{SMSize}{\sum_{\boldsymbol{A}\in ReuseSet|CoalesceSet} \boldsymbol{DS_A}} \leqslant TSizeMax. \tag{11}$$

We assess the potential profit of each partition plan with (8), i.e., the global references that can be reduced with shared memory optimization.

GPUs enforce several hardware restrictions on resources to guarantee high parallelism. The number of processors puts limit on attainable parallelism, such as the maximum number of threads and TBs that can be hosted. Similarly, shared memory size also confines the number of concurrent TBs. Our compiler ties to seek a partition plan that satisfies all hardware restrictions, including shared memory size, maximal amount of TB and threads on each SM ((9)∼(11)). (9) utilizes a threshold θ_s to restrict the shared memory consumption in each TB. It not only prevents excessive shared memory usage to boost TB level parallelism, but also attends to architectural restrictions on shared memory (e.g., though each SM on AMD GCN GPUs is equipped with 64 KB shared memory, only 32 KB is useable for each TB). Therefore, the absence of θ_s tends to result in inefficient (even invalid) code that falls short on parallelism. From our experiments and experience, we pick a value between 0.125 and 0.25 for θ_s to allow for four to eight concurrent TBs on an SM. In the future, we plan to automate the selection of θ_s.

Accordingly, the solution space for `gemm` in Fig.2(a) can be presented by:

$$\begin{aligned}
&\text{max. } Profit \\
&= TB_X \times TB_Y \times n \times 8 + 256 \times TB_Y \times n \\
&= 8 \times n \times TB_Y \times (TB_X + 32), && (12)\\
&\text{s.t.} \quad (TB_X \times k_1 + TB_Y \times k_2) \times 4 \\
&\leqslant 0.25 \times 48 \times 1\,024, && (13)\\
&\quad TB_X \times TB_Y \leqslant 1\,024, && (14)\\
&\quad 32 \leqslant TB_X \leqslant 1\,024, TB_X = 2^{m_1}, \\
&\quad m_1 \in [5, 6, ...], && (15)\\
&\quad 1 \leqslant TB_Y \leqslant 1\,024, TB_Y = 2^{m_2}, \\
&\quad m_2 \in [0, 1, ...], && (16)\\
&\quad TB_X \times TB_Y \times \frac{48 \times 1\,024}{(TB_X \times k_1 + TB_Y \times k_2) \times 4} \\
&\leqslant 2\,048, && (17)\\
&\quad 0 \leqslant k_1 \leqslant n, \\
&\quad 0 \leqslant k_2 \leqslant n,
\end{aligned}$$

where (13) confines the shared memory consumption of a TB, (14)∼(16) ensure the amount of threads in a TB does not become illegal, and (17) requires the total threads in an SM do not break hardware restriction. As a common practice on GPU, we only search for TB size that is a power of 2 in each dimension. Besides, we expect a TB_X larger than 32 ((15)), otherwise the affine reference to $B[k][j]$ would not coalesce.

Optimal Partition. Our compiler designs following rules to choose the optimal iteration subspace and parameters.

If all candidate partitions are non-parametric, the profit of each plan is a positive number. Thus, we can choose the corresponding candidate partitions with the highest profit.

In the case of parametric candidate partitions, we are not able to compare profits directly. In practice, the linear programming model we proposed cannot guarantee an optimal partition since the existing restrictions might not be tight enough. However, all partition parameters are bounded by array dimension sizes, which leads to a solution set with limited integers. Therefore, we propose three steps to find an optimal partition. First, for each candidate partition plan, we obtain a legal parameter space by enforcing the restrictions on GPU resources and array bounds. Second, we traverse the legal parameter space and arrive at the optimal parameter combination of each candidate partition. Third, we choose the candidate partition and parameter combination with the highest profit. Hence,

the size of our search space depends on the number of candidate partition plans and the amount of legal parameter combinations, which are both polynomial as presented in (18) and (19). Therefore, theoretically, our approach ends in polynomial time. Notice that by providing seed arrays and parametric array partitions, programmers have effectively cut down the search space for our compiler. Otherwise, we would blindly consider every single array as a seed array and every possible parameter combination for seed arrays.

$$NumCanPar = \sum_{NumSeed} NumAffineReference, \quad (18)$$

$$NumParaComb = \prod_{NumParameter} NumLegalValues. \quad (19)$$

To solve an optimal partition for `gemm` in Fig.2(a), our compiler tries out all legal TB sizes for TB_X and TB_Y. As the profit of utilizing shared memory equals $TB_Y \times (TB_X + 32)$ ((12)), the larger TB_Y is, the higher profit we can achieve. Due to TB size restrictions in (14)∼(16), the maximal TB_Y we can use is 32 (as TB_X is at least 32). Therefore, the optimal TB size is 32×32. In this case, the constraint on shared array size is reduced to $k_1 + k_2 \leqslant 96$. As k_1 and k_2 only affect shared memory usage of a TB, we set both variables to 32 for simplicity.

3.3 Advanced Optimizations on Parallelism

SM opt. phase can reduce off-chip traffic and obtain performance boost through utilizing shared memory. However, memory bandwidth and compute cores remain underutilized due to low memory and instruction level parallelism. Bank conflicts, usually caused by imperfect shared memory access patterns, force memory transactions to delay and thus decrease overall performance[17]. ILP, which reflects GPU core utilization, is another important performance indicator. Many studies tried to improve ILP from different aspects[9-11,17], but none of them manages to choose a proper amount of ILP automatically. Moreover, finding the right timing to perform ILP is also nontrivial. Therefore, our compiler detects and eliminates memory bank conflicts of an input kernel, before choosing and increasing an optimal amount of ILP (as shown in the gray box at the bottom of Fig.3).

3.3.1 Bank Conflict Elimination

Similar to memory in multicore system[18], GPU divides shared memory into equally-sized memory banks that can be accessed simultaneously for high bandwidth. When multiple addresses in the same bank are accessed at the same time, memory requests are serialized. This is known as bank conflict, which hurts memory level parallelism and undermines benefits of shared memory optimization. To make the best of memory bandwidth, it is therefore important to avoid bank conflicts.

Bank conflict is closely related to shared memory bank width and bank number (*bank_width* and *bank_num*), and shared array data type and access pattern (*data_type* and *array_stride*). Hence, avoiding bank conflict is complex for programmers, but can be automated as compilers have easy access to above information.

Conflict Detection. We devise two methods to detect bank conflicts for each shared array access. First, for simple (e.g., constant stride) array accesses, conflict degrees can be inferred by (20) and (21). We utilize bank stride to normalize the difference in array strides and data types. Second, when special conflict rules or variable array strides make above equations inaccurate, low overhead simulations can be used to estimate conflict degrees[19-20].

$$bank\_stride = \frac{array\_stride \times data\_type}{bank\_width}, \quad (20)$$

$$Degree = \begin{cases} |1/bank\_stride|, \\ \quad \text{if } bank\_stride < 1, \\ \text{GCD}(bank\_stride, bank\_num), \\ \quad \text{otherwise.} \end{cases} \quad (21)$$

Conflict Elimination. Targeting bank conflicts in different cases, our compiler provides five transformations to avoid bank conflicts and employs a decision tree to select a suitable one. In general, the transformations we provide minimize bank conflicts from the perspective of data and code: data reorganization targets the most typical scenarios and features a data preprocess stage; code restructure, which applies to more general cases, trades some extra computations and compiler work for conflict free accesses. As shown in Fig.4, our decision tree relies on four code features to select a suitable transformation, i.e., array of structure (AOS), 2D array with strided reference, stride size of the 2D array, and associative operations to combine the conflict references. For each conflict array, our compiler

Table 1. Techniques to Avoid Bank Conflicts

		Data Reorganization			Code Restructure	
		AOS-SOA	Padding	Transpose	Thread Remap	Access Reorder
Application scope	Code feature	AOS	2D, strided array access, $array_{\text{width}}$ mod $SM_{\text{width}} \neq 0$	2D, array accessed by column repeatedly	—	Associative operation
	Typical algorithms	Complex number, tree, graph, neural network	Sort, scan, tree	—	—	—
Properties	Extra space	×	√	×	×	×
	Extra computation	×	×	×	√	√
	Compiler search	×	√	×	√	√
	Introduce conflict	×	×	√	×	×
	Data preprocess	√	√	√	×	×

Note: — indicates that an application scope does not apply to a technique. × indicates that a technique does not present a property. √ indicates that a technique presents a property.

checks for above code features and decides the appropriate transformation accordingly. We summarize our techniques to avoid bank conflicts in Table 1 and highlight three of them.

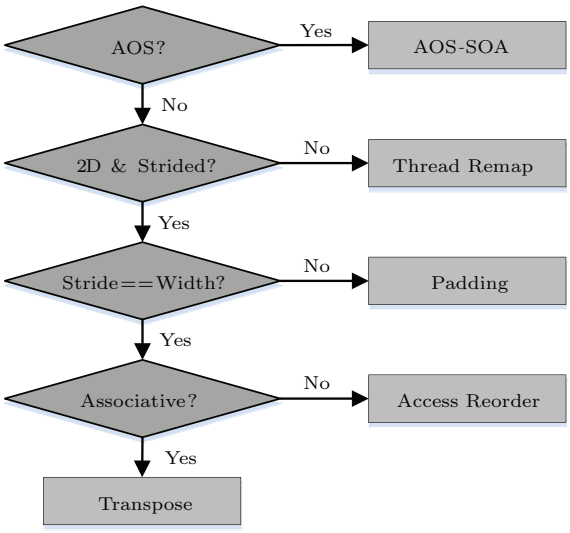


Fig.4. Bank conflict transformation decision tree.

Padding scatters accesses to different banks by adding empty columns to an array. For 2D arrays, strided access and an array width that is not evenly divisible by shared memory width are two common code features that invite padding. Strided access is common in sort, scan, and tree algorithms as the stride doubles at each level of shared memory access. Apart from space cost, padding requires some compiler analysis to pin down an optimal padding size.

Thread remap designs different placement matrices for each parallel loop to change access patterns of consecutive threads. For example, diagonal remapping[1] eliminates bank conflicts by mapping consecutive threads to diagonal work items, which makes use of a placement matrix $\boldsymbol{Q}$, with each value being $Q_{00} = 1$, $Q_{01} = 0$, $Q_{10} = 1$, $Q_{11} = 1$.

Thread remap does not introduce extra space or bank conflicts at the cost of some extra index computation. However, finding the perfect placement matrix requires exhaustive compiler search.

Access reorder schedules shared array accesses and associative operations of consecutive threads to minimize bank conflicts. For example, by varying start offsets of each thread that accesses an array by row, consecutive threads can operate on different banks at the same time[5]. Access reorder trades extra index and offset computation for higher memory level parallelism, without sacrificing shared memory space. However, float point operations should be handled with care to avoid precision errors.

Since the width of shared arrays is often correlated with TB width, which is a power of 2, bank conflicts are common after shared memory optimization. In the same spirit, above detection and solutions can be easily ported to avoid global memory channel conflicts, which share the same features with bank conflicts.

3.3.2 *ILP Enhancement*

Supported by instruction pipelines and high memory bandwidth on GPU, ILP is achieved by scheduling independent instructions for concurrent execution. GPU performance benefits from high ILP in two ways: better latency hiding and better resource utilization. First, long latency instructions can interleave with independent instructions to hide latency. Second, executing

more independent instructions at the same time makes better use of available cores and bandwidth. Meanwhile, ILP is increasingly vital as modern GPUs are offering more cores, schedulers, registers, and higher bandwidth.

However, current accelerator oriented APIs choose to ignore ILP and execute one work item per thread, which results in poor performance. By contrast, we propose to expose ILP by computing N elements per thread, thus offering N times perfectly parallel instructions per thread. Moreover, we believe ILP enhancement is more of an automatic optimization than a manual one, since ILP factor is too complex for a programmer to decide.

ILP and Shared Memory. ILP and shared memory optimization should be applied together as they are mutually beneficial. On one hand, for memory bound kernels with non-coalescing or bank conflict memory requests, additional memory instructions lead to worse bandwidth utilization and delayed memory operations[21]. Fig.5 demonstrates the performance loss of ILP optimization in presence of non-coalescing memory operations. Consequently, ILP enhancement should be carried out after our shared memory and bank optimizations, which eliminate unsatisfactory memory patterns. On the other hand, suffering from poor parallelism, programs with excessive shared memory demand are more likely to benefit from ILP enhancement.

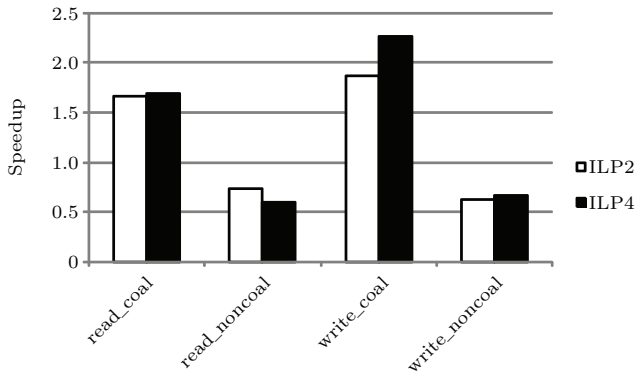


Fig.5. Average speedup of ILP optimization for different memory operations across all platforms.

Optimal ILP Factor Search. ILP factor affects the workload of each thread and the available parallel instructions on a stream multiprocessor. The best ILP factor depends on several parameters: available resources, compute density, TB size, etc.

Though searching an optimal ILP factor seems to be challenging, we can prune the search space to a great extent based on two understandings. First, the ILP factor is usually a power of 2. As it is conventional to set TB size to 2^n for many GPU kernels, an ILP factor that is not a power of 2 cannot be divided by such TB size and thereby results in illegal TB size. In this case, restricted by the upper and the lower bound of a TB size (e.g., 1 024 and 32), there are only five legal ILP factors in each dimension. Second, to preserve memory coalescing in X dimension of a TB, we choose to perform ILP optimizations on Y and Z dimensions, which further reduces legal ILP factors.

Therefore, automatic ILP factor tuning can be achieved by trying out all legal ILP factors of each kernel. The highest speedup identifies the optimal ILP factor on current platform.

ILP Code Generation. We present the code transformation of ILP optimization in Algorithm 1 and highlight two techniques designed to ensure correct translation. As an input kernel is free of dependence in global memory, we focus on dependence induced by shared memory and local variables.

Algorithm 1. Code Transformation of ILP Optimization

Require:
CUDA/OpenCL code before ILP optimization, *code*
direction along which ILP will be enhanced, *dir*
TB size on *dir* direction, *sizedir*
ILP factor, $factor$
Ensure:
ILP optimized CUDA/OpenCL code, *ilp_code*
1: Liveness analysis of variables in *code*
2: **for all** *code_region* separated by barriers in *code* **do**
3: Surround *code_region* with ILP loop: [*ilp_index*, $0{:}sizedir{:}\frac{sizedir}{factor}$]
4: Add all local ID references on *dir* direction with *ilp_index*
5: **for each** variable *live* that lives at *code_region* exit **do**
6: Array expansion to $live[factor]$
7: Replace access to *live* with $live[index]$
8: **end for**
9: Loop invariant hoisting within *code_region*
10: Add transformed *code_region* and barrier into *ilp_code*
11: **end for**
12: **return** *ilp_code*

First, to preserve dependences on shared memory, code replication is performed for each barrier-free code block. As barriers are commonly used to separate read and write phases on shared memory, unrolling an entire kernel incurs premature usage of shared memory values. In contrast, by unrolling read/write phases of shared memory respectively, we are able to preserve dependences and provide independent instructions as well.

Second, array expansion is used to handle dependence on local variables, which are often used to pass

temporary results through barriers. Violating dependences on these local variables will induce polluted temporary values. Array expansion secures these dependences by using disjoint space to hold temporary values.

4 Experiments

In this section, we verify effectiveness of our shared memory centric optimizations on four different GPU platforms. Specifically, we 1) analyze our performance advantage over several mainstream GPU programming models, 2) decompose the performance contribution of each optimization and 3) verify the portability of our optimizations.

4.1 Methodology

Compiler. We choose to implement our optimizations inside Cetus[22], a source-to-source compiler for C programs that inhabits OpenMP to CUDA translation support[23]. Our compiler, which is built on Cetus version 1.3.1, takes in pragma annotated sequential C programs and returns the optimized OpenCL kernel. We use `gcc` version 4.4.7 to compile the optimized OpenCL source code of our compiler. In comparison, we use `PGI` version 14.6 and `CAPS` version 3.4.4 to compile OpenACC and OpenHMPP programs, respectively; and we use `nvcc` 5.0 for CUDA code generation.

Benchmarks. We use five benchmarks: `mt`, `gemm`, `mv`, `gauss` and `hotspot`. In Table 2, we briefly describe the benchmark algorithms and datasets. We also report programming complexity using LOC of the sequential algorithm. Meanwhile, we present LOP of OpenHMPP algorithm and our *share* pragma to reveal the productivity and optimizing efforts. We select these benchmarks, which can be easily ported to GPU platforms, as they require major optimization efforts to obtain high performance. As a typical memory bound application, `mt` features 8 bytes of I/O traffic and 4 arithmetic operations per thread. In addition to heavy I/O traffic, `gemm` and `mv` exhibit high compute intensity and significant data locality. `gauss` and `hotspot` instead embody different degrees of data reuse within a TB. Moreover, the computation intensity of both kernels is relatively higher with dozens of multiply-add operations.

Platforms. We evaluate our compiler on four GPUs: NVIDIA GTX 690, NVIDIA GTX TITAN, AMD HD 7850 and NVIDIA Tesla C2050. As shown in Table 3, substantial differences exist within these four platforms ranging from the number of cores to shared memory size. These differences pose severe challenges to the performance portability of our compiler. For example, GTX TITAN provides as many as 3 072 compute cores, while Tesla C2050 has only 448 cores. Each stream multiprocessor of HD 7850 is equipped with 64 KB shared memory, which is one third more than others.

4.2 Experimental Results

4.2.1 Comparison Against Mainstream Programming Models

To illustrate the effectiveness of our compiler, we compare the performance of our auto-generated kernels against that of several mainstream GPU programming models, i.e., OpenACC, OpenHMPP and CUDA on GTX 690. Fig.6 reveals the performance compar-

Table 2. Benchmarks

Name	Description	Input	Data Type	LOC (*seq*)	LOP (*share*)	LOP (*hmpp*)
`mt`	Matrix transpose	1M∼64M	Float	7	1	17
`gemm`	General matrix multiplication	1M∼64M	Float	14	1	22
`mv`	Matrix vector multiplication	1M∼256M	Float	9	1	17
`gauss`	Blurring images with 5×5 filter	1M∼256M	Unsigned char	29	1	43
`hotspot`	2D thermal simulation kernel	16K (1K∼16K steps)	Float	42	1	84

Note: LOC represents lines of code and pragmas in the input loop, whereas LOP only counts lines of pragmas.

Table 3. Platforms

GPU	Core Number	Clock Rate (MHz)	Memory (GB)	Bandwidth (GB/s)	Shared Memory (KB)	Driver	SDK
NVIDIA GTX 690	3 072	915	4	384.0	48	304.33	CUDA 5.0
NVIDIA GTX TITAN	2 688	837	6	288.4	48	319.37	CUDA 5.5
AMD HD 7850	1 024	860	2	153.6	64	AMD Catalyst 14.1	APP SDK v2.9
NVIDIA Tesla C2050	488	1 150	6	144.0	48	285.05.33	CUDA 4.1

ison, with OpenACC being the baseline. While OpenACC and OpenHMPP achieve 26% and 68% performance against native CUDA code (which is consistent with previous studies[24-26]), our compiler generates efficient kernels and yields over 90% of the performance of hand-crafted CUDA code on average.

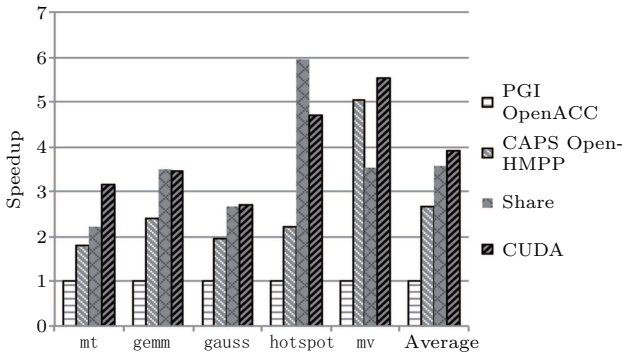


Fig.6. Performance comparison with PGI OpenACC, CAPS OpenHMPP, and NVIDIA CUDA on GTX 690.

We choose PGI OpenACC compiler over CAPS for its better implementation and higher performance. Still, the experimental results show that our compiler achieves consistently better performance than PGI OpenACC. All kernels annotated with OpenHMPP are hand-coded to utilize shared memory, which takes great effort judging from LOP (*hmpp*) in Table 2. Meanwhile, we further improve memory and instruction parallelism automatically, as illustrated in Subsection 3.3. It can be inferred from Fig.6 that our compiler is superior to OpenHMPP in most cases, offering an impressive speedup of 1.3x on average.

We manually applied shared memory optimization to CUDA versions of the benchmarks, in addition to eliminating memory level conflicts and increasing ILP. Fig.6 shows that the performance gap between our auto-generated kernels and the hand-written CUDA codes is less than 9% on average, which is acceptable as CUDA performs at most 30% better than OpenCL for most applications[27-28]. `hotspot` is an exception which our auto-generated OpenCL code presents better performance than CUDA. This results from the difference between OpenCL and CUDA, as they store arrays in different memory spaces for `hotspot`.

4.2.2 Shared Memory Centric Optimizations

In Fig.7, we present our speedups over naive OpenCL implementations of each kernel, and we report the performance decomposition of each optimization on GTX 690 to illustrate their respective contributions. Speedups are averaged over all problem sizes. On average, our compiler achieves 2.8x performance boost: the speedups of benchmarks with serious memory bottlenecks, i.e., `mt`, `gemm`, and `mv`, reach 3.6x; other computation intensive benchmarks (`gauss` and `hotspot`) also reach 1.5x. Besides, all three shared memory centric techniques are proved to be effective. We further analyze their respective performance impact in this subsection.

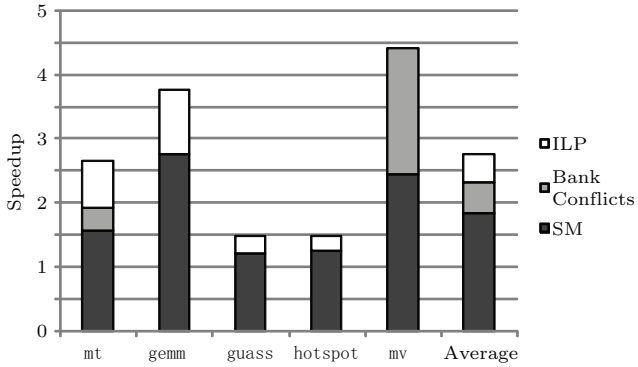


Fig.7. Decomposition of speedups over naive kernels on GTX 690.

Shared Memory Optimization. Overall, shared memory optimization can achieve an average of 1.8x speedup, which contributes to 67% of the total performance gain. Recall that all programmer needs to provide to achieve these results is properly annotated *share* pragmas. Then our compiler solves an optimal partition scheme for arrays and utilizes shared memory to reuse or coalesce off-chip data automatically.

By examining the adjacency constraints, our compiler discovers non-coalescing global references in `mt`, `mv`, and `gemm`. As discussed in Subsection 3.2, array $\boldsymbol{A}$ in `gemm` falls into *CoalesceSet* as the adjacent elements in $\boldsymbol{A}$ are not accessed at the same timestamp. In the same spirit, the input vector in `mv` violates the time constraint, whereas the input matrices in both `mt` and `mv` break the space constraint. Besides, we manage to identify a considerable amount of data reuse in `mv` and `gemm`. Similar to $\boldsymbol{A}$ and $\boldsymbol{B}$ in `gemm`, the input vector from `mv` is read repeatedly, which results in a *ReuseSet* that is many times larger than its candidate partition. By adapting these non-coalescing requests and reused data to shared memory, we achieve an impressive speedup of 2.3x. Though having a great volume of reused data, `gauss` and `hotspot` respond rather modestly to shared memory optimization (1.2x) due to their cache friendly access patterns. Furthermore, the

benefits of reusing data are offset by 55% more ALU instructions introduced by shared memory utilization. Hence, solely relying on shared memory for high performance is insufficient.

Bank Conflict Elimination. Our compiler manages to utilize memory bandwidth by detecting and avoiding bank/channel conflicts after analyzing array access patterns. Among all benchmarks, high degree bank conflicts are detected in `mt` and `mv`, as all threads within a warp request data elements in the same bank. Our compiler chooses to avoid bank conflicts by padding one column to each shared array, which is 2D and has a stride smaller than array width. As a result, we are able to reduce bank conflicts and accelerate memory access by 0.4x and 2.0x, respectively, boosting average performance by 17%.

ILP Enhancement. To exploit performance potential of compute resources, our compiler enhances ILP by filling idle time slots of memory and arithmetic latencies with independent instructions. As shown in Fig.7, enhancing ILP contributes to an average of 0.45x speedup. While populating parallel instructions in compute intensive applications achieves little runtime reduction (0.2x), it is especially effective for I/O bound applications (an average of 0.8x speedup). With much longer latency than arithmetic instructions, memory instructions enable better latency hiding and higher performance speedup. However, `mv` is better off without ILP optimization as utilizing shared memory and eliminating bank conflict already offer sufficient parallelism.

In our framework, we try to improve performance of kernels by utilizing GPU resources. Shared memory, a perfect place to hold off-chip data for reuse or coalescing purposes, promises significant speedup for memory operations. We take in programmers' optimization advice, based on which we explore optimal array partition scheme and produce shared memory optimal code. While the bandwidth of shared/global memory and hundreds of processing cores allow high memory and instruction level parallelism, they might be underutilized in GPU kernels. Consequently, we apply automatic optimization to avoid memory conflicts and increase independent instructions. We show not only that our compiler can fix inefficient array references to a large extent with shared memory but also that it can refine memory and instruction level parallelism.

4.2.3 Performance Portability

In order to evaluate the performance portability of our compiler, we port our benchmarks to four platforms and present their performance results in Fig.8. The performance improvement of each optimization is averaged over all benchmarks. On the whole, our compiler is effective across all platforms, offering an average performance gain of 3.7x. Among all optimizations, utilizing shared memory remains a major performance booster (offering 70% of total performance), with bank conflict avoidance and ILP enhancement each constituting 15% of total performance.

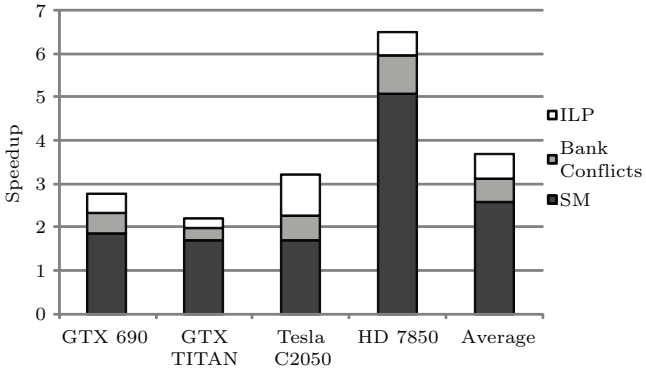


Fig.8. Decomposition of performance improvement across all platforms.

Portable Optimizations. Though our compiler is generally effective on all platforms, we actually handle our optimizations slightly different as platforms exhibit various features as described in Table 3. Our compiler is able to choose similar parameters for GTX TITAN and GTX 690, which are close in design and compute capability. Therefore, we obtain similar performance for each optimization on these platforms. In the mean time, ILP optimization of our compiler prefers GPUs with more registers, which can hold values for more active instructions. Therefore, Tesla C2050 and HD 7850, with 2 times more registers for each processor, enjoy finer latency hiding and higher performance improvement from ILP optimization. Our shared memory utilization contributes to an evidently higher speedup (5.0x) on HD 7850, which offers a larger shared memory space and thus allows more parallel TBs. Moreover, avoiding channel conflicts on HD 7850 improves performance by 2.0x for `mt` and `mv`. Adjacent TBs access data that fall into the same memory channel, thus limiting effective global memory bandwidth. Among all solutions in Table 1, our compiler chooses to apply diagonal remapping and arithmetic reordering to spread concurrent requests, according to the decision tree.

Portable ILP Factor. The ILP factor directly reflects the number of parallel instructions, and should

be carefully chosen as a small one fails to hide latency perfectly and a large one introduces superfluous register pressure. Our compiler manages to find optimal ILP factors in terms of different kernels, platforms, and problem sizes. Table 4 displays the best ILP factors chosen for each benchmark on all platforms. Notice that as all TB sizes in our experiment are powers of 2, we choose ILP factors that are also 2^n to avoid illegal TB size. As can be seen, we select larger ILP factors for memory intensive applications, such as `gemm` and `mt`, since off-chip memory latency needs more parallel instructions to hide. Furthermore, our compiler is aware of higher ILP demand on GTX 690 and GTX TITAN, both of which issue more independent instructions per cycle. We also recommend a higher ILP factor for Tesla C2050 to hide longer latency of memory and computation. In addition, our compiler adapts to parallelism requests for different input sizes. For `mt` on Tesla C2050, our compiler chooses 2 as the optimal factor for a 16M input and 4 for 64M; otherwise, there is a performance difference as large as 0.3x if an identical ILP factor is used.

Table 4. Optimal ILP Factor for All Benchmarks

Platform	mt	gemm	mv	gauss	hotspot
GTX 690	4	8	1	2	2
GTX TITAN	2	8	1	4	1
Tesla C2050	4	8	1	4	2
HD 7850	4	4	2	4	1

In summary, our experimental results show that our optimizing compiler generates high quality code on diverse platforms and often achieves comparable performance, even compared with manually optimized code in OpenHMPP and CUDA.

5 Related Work

5.1 Optimizing for Shared Memory

Manual Optimization. Utilizing shared memory to achieve coalesced memory accesses[1-2] and to act as a software managed cache[3-4] has been studied thoroughly by researchers; however, most of them remain manual optimization. Optimizing for GPU calls for considerable experience and a deep understanding of hardware details[29]. Moreover, optimization techniques need adjustment when ported to a new platform[30]. In comparison, Our compiler simplifies the work of programmers by generating optimized code automatically.

Optimizing Compiler Based on Compiler Analysis. For the sake of programmability, correctness and productivity, optimizing compilers are regarded as a relatively easy and reliable shortcut to high performance[5,7,31]. CUDA-lite[7] is one of the first compilers to coalesce global memory requests with some programmer annotation. CUDA-lite translates a naive kernel annotated with parallelism and array information into a memory coalesced version, using shared memory as a temporary depot. In comparison, our compiler requires only data management hints from programmers. More importantly, our compiler aims at global access coalescing and data reuse alike. Yang *et al.*[5] designed an auto-optimizing source-to-source GPU compiler, which manages to increase memory throughput by employing shared memory. Although utilizing shared memory for similar purposes, we approach this optimization from a data centric perspective. Besides, we trust programmers to provide better insight into the memory pattern and data partition, and take their advice for shared memory utilization.

Optimizing Compiler Based on Polyhedral Model. As the basis for major advances in automatic program optimization and parallelization[12-13], the polyhedral model has been introduced to build efficient GPU compilers. Baskaran *et al.* conducted a series of studies on translating affine loop nests to GPUs[32] and built an automatic C-to-CUDA compiler[15] under the polyhedral framework. They adjusted Pluto to find affine transforms that enable global memory coalescing. While C-to-CUDA always maps all arrays into shared memory, our compiler can solve an optimal combination of arrays and filter out those that do not benefit from shared memory optimization or cannot fit into shared memory. Therefore, we avoid inefficient or invalid optimizations, which are possible in C-to-CUDA. Gpuloc is based on the algorithm proposed by Baghdadi *et al.*[33] It uses a ranking-based technique[34], which targets only data locality, to manage data movement between shared memory and global memory. By contrast, we consider coalescing global memory accesses with shared memory, which is even more significant in improving performance. Moreover, GPUloc utilizes a dual buffer system, which wastes memory and limits parallelism. In comparison, our compiler has a smaller memory request.

Pragma Directed Programming. Portability and productivity motivated the design of accelerator oriented APIs, which aim to ease GPU programming by designing OpenMP like pragmas. In the OpenACC

API, the programmer annotates the sequential code with compiler directives, indicating those code regions susceptible to be executed on GPU. Programmers can have fine control over parallelism and data management. However, shared memory optimization inside OpenACC is not satisfactory. Targeting data reuse, OpenACC requires the programmer to specify the elements that will be reused for each thread. In contrast, we design *share* pragma to handle data reuse as well as data coalescing scenarios. Moreover, from a data centric point of view, we allow a more flexible way to describe data management decisions compared with OpenACC. OpenHMPP distinguishes from OpenACC in its ability to perform low-level optimizations, including shared memory optimization. However, in order to utilize shared memory, programmers are responsible of moving data in and out of shared memory as well as reading data from shared memory, which is as complex as programming in CUDA/OpenCL. On the contrary, our compiler accepts a few hints from programmers and does not bother programmers with code generation.

5.2 Optimizing for ILP

Volkov[11] studied the performance impact of parallel instructions and inspired many studies[9-10] to optimize ILP of GPU applications. He revealed that enhancing ILP facilitates latency hiding and can be an effective performance technique. However, [9-10] fail to notice the performance defect with ILP optimization in case of bad memory access patterns. In contrast, we choose to perform ILP optimization on top of our shared memory utilization to eliminate interference from bad memory patterns. Moreover, our compiler enables automatic ILP code generation and ILP factor selection, and therefore relieves programmers from troublesome coding and cross platform tuning.

A more recent work[9] managed to improve ILP for GPU by scheduling PTX instructions across branches on critical path. Our ILP optimization works on source code level instead. Nevertheless, their work could complement ours and further improve ILP on PTX code level.

6 Conclusions

In this paper, we introduced several shared memory related optimization techniques for better bandwidth utilization and parallelism on GPUs. We leveraged programmer knowledge for pragmas concerning data partition. Then, our compiler tries to harness shared memory for data reuse and coalescing opportunities. Besides, we designed two automatic optimizations on top of shared memory: bank conflict avoidance and ILP enhancement, which target higher memory and instruction parallelism, respectively. A set of compiler techniques was proposed to generate optimized kernel code. Our experimental results showed that the optimized code achieves high performance, often superior to current accelerator oriented APIs.

Our future research will focus on enhancing the applicability of our compiler framework, including automatically selecting the thresholds, and coping with larger applications with multiple affine sections. We will also combine ILP optimization with register tiling in our compiler so as to improve how to select ILP factors.

References

[1] Ruetsch G, Micikevicius P. Optimizing matrix transpose in CUDA. http://www.cs.colostate.edu/∼cs675/MatrixTranspose.pdf, Jan. 2009.

[2] Fujimoto N. Faster matrix-vector multiplication on GeForce 8800GTX. In *Proc. IEEE International Symposium on Parallel and Distributed Processing*, Apr. 2008.

[3] Van Werkhoven B, Maassen J, Bal H E, Seinstra F J. Optimizing convolution operations on GPUs using adaptive tiling. *Future Gener. Comput. Syst.*, 2014, 30: 14-26.

[4] 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. the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010.

[5] Yang Y, Xiang P, Kong J, Zhou H. A GPGPU compiler for memory optimization and parallelism management. In *Proc. the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2010, pp.86-97.

[6] Kandemir M, Kadayif I, Sezer U. Exploiting scratch-pad memory using Presburger formulas. In *Proc. the 14th International Symposium on Systems Synthesis*, Sept. 2001, pp.7-12.

[7] Ueng S Z, Lathara M, Baghsorkhi S, Hwu W. CUDA-Lite: Reducing GPU programming complexity. In *Proc. the Languages and Compilers for Parallel Computing*, July 3-Aug. 2, 2008, pp.1-15.

[8] Yang Y, Xiang P, Mantor M, Rubin N, Zhou H. Shared memory multiplexing: A novel way to improve GPGPU throughput. In *Proc. the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2012, pp.283-292.

[9] Jablin J A, Jablin T B, Mutlu O, Herlihy M. Warp-aware trace scheduling for GPUs. In *Proc. the 23rd International Conference on Parallel Architectures and Compilation*, Aug. 2014, pp.163-174.

[10] Schäfer A, Fey D. High performance stencil code algorithms for GPGPUs. *Procedia Computer Science*, 2011, 4: 2027-2036.

[11] Volkov V. Better performance at lower occupancy. www.cs.berkeley.edu/∼volkov/volkov10-GTC.pdf, Dec. 2014.

[12] Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2008, pp.101-113.

[13] Bastoul C. Code generation in the polyhedral model is easier than you think. In *Proc. the 13th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 29-Oct. 3, 2004, pp.7-16.

[14] Baskaran M M, Bondhugula U, Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proc. the 22nd Annual International Conference on Supercomputing*, Jun. 2008, pp.225-234.

[15] Baskaran M, Ramanujam J, Sadayappan P. Automatic C-to-CUDA code generation for affine programs. In *Proc. the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, Mar. 2010, pp.244-263.

[16] Pouchet L N. Polyhedral compilation foundations. http://web.cs.ucla.edu/∼pouchet/lectures/doc/888.11.2.pdf, Dec. 2014.

[17] Murthy G S, Ravishankar M, Baskaran M M, Sadayappan P. Optimal loop unrolling for GPGPU programs. In *Proc. the 2010 IEEE International Symposium on Parallel & Distributed Processing* (*IPDPS*), Apr. 2010.

[18] Liu L, Li Y, Cui Z, Bao Y, Chen M, Wu C. Going vertical in memory management: Handling multiplicity by multi-policy. In *Proc. the 41st ACM/IEEE International Symposium on Computer Architecture* (*ISCA*), Jun. 2014, pp.169-180

[19] Gao S. Improving GPU shared memory access efficiency [Ph.D. Thesis]. University of Tennessee, 2014.

[20] Gou C, Gaydadjiev G. Addressing GPU on-chip shared memory bank conflicts using elastic pipeline. *International Journal of Parallel Programming*, 2013, 41(3): 400-429.

[21] Ryoo S, Rodrigues C I, Baghsorkhi S S, Stone S S, Kirk D B, Hwu W W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2008, pp.73-82.

[22] Lee S I, Johnson T, Eigenmann R. Cetus — An extensible compiler infrastructure for source-to-source transformation. In *Lecture Notes in Computer Science 2958*, Rauchwerger L (ed.), Springer Berlin Heidelberg, 2004, pp.539-553.

[23] Lee S, Min S, Eigenmann R. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proc. the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2009, pp.101-110.

[24] Wienke S, Springer P, Terboven C, an Mey D. OpenACC — First experiences with real-world applications. In *Lecture Notes in Computer Science 7484*, Kaklamanis C, Papatheodorou T, Spirakis P G (eds.), Springer Berlin Heidelberg, 2012, pp.859-870.

[25] Catanzaro B, Garland M, Keutzer K. Copperhead: Compiling an embedded data parallel language. Technical Report, UCB/EECS-2010-124, EECS Department, University of California, Berkeley, Sept. 2010.

[26] Reyes R, López I, Fumero J, de Sande F. A preliminary evaluation of OpenACC implementations. *The Journal of Supercomputing*, 2013, 65(3): 1063-1075.

[27] Fang J, Varbanescu A, Sips H. A comprehensive performance comparison of CUDA and OpenCL. In *Proc. the International Conference on Parallel Processing*, Sept. 2011, pp.216-225.

[28] Karimi K, Dickson N G, Hamze F. A performance comparison of CUDA and OpenCL. arXiv: 1005.2581, 2010. http://arvix.org/abs/1005.2581, Jan. 2016.

[29] Li C, Yang Y, Dai H, Yan S, Mueller F, Zhou H. Understanding the tradeoffs between software-managed vs. hardware-managed caches in GPUs. In *Proc. the 2014 IEEE International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), Mar. 2014, pp.231-242.

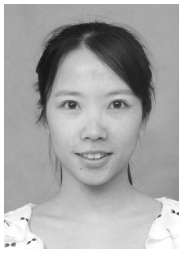
[30] Chen G, Wu B, Li D, Shen X. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proc. the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO*), Dec. 2014, pp.88-100.

[31] van den Braak G, Mesman B, Corporaal H. Compile-time GPU memory access optimizations. In *Proc. the 2010 International Conference on Embedded Computer Systems* (*SAMOS*), Jul. 2010, pp.200-207.

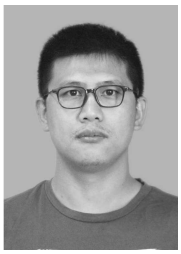
[32] Baskaran M M, Bondhugula U, Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proc. the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2008, pp.1-10.

[33] Baghdadi S, Größlinger A, Cohen A. Putting automatic polyhedral compilation for GPGPU to work. In *Proc. the 15th Workshop Compilers for Parallel Computers*, Jul. 2010.

[34] Größlinger A. Precise management of scratchpad memories for localising array accesses in scientific codes. In *Proc. the 18th International Conference on Compiler Construction*, Mar. 2009, pp.236-250.



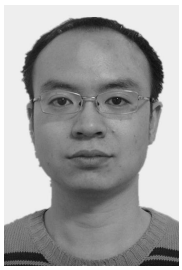
Jing Li received her B.S. degree in software engineering from Wuhan University, Wuhan, in 2012. Currently she is a Ph.D. candidate of Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. Her research interests include programming language and optimization on GPUs.



Lei Liu received his B.S. degree in computer science from Changchun University of Science and Technology, Changchun, in 2001, M.S. degree in computer science from Jilin University, Changchun, in 2004, and Ph.D. degree in computer architecture from ICT, CAS, Beijing, in 2010. He participated in the Advanced Compiler Technology Laboratory (ACT) of ICT, CAS, in 2010, and is now an assistant professor of ICT, CAS. His research interests include programming language and compiler optimization.

Yuan Wu received his B.S. degree in computer science and technology from Zhengzhou University, Zhengzhou, in 2012. He is now a senior engineer of Samsung Electronics, Beijing. His research interests include compiler optimization and performance analysis.

Xiang-Hua Liu received his B.S. degree in electrical engineering from Beijing Institute of Technology, Beijing, in 1998, M.S. degree in electrical engineering from Beijing Institute of Technology, Beijing, in 2001, and Ph.D. degree in electrical and information engineering from Beihang University, Beijing, in 2005. Currently he is a principle engineer of Samsung Electronics, Beijing, working on compiler development and software optimization.



Yi Gao received his B.S. degree in computer software in 2002, and M.S. degree in computer architecture in 2005, both from Peking University, Beijing. He is now a senior engineer of Samsung Electronics, Beijing. His research interests include compiler optimization and performance analysis.

Xiao-Bing Feng received his B.E. degree in computer software from Tianjin University, Tianjin, in 1992, M.S. degree in computer software from Peking University, Beijing, in 1996, and Ph.D. degree in computer architecture from ICT, CAS, Beijing, in 1999. Now he is a professor and Ph.D. supervisor of ICT, CAS. His research interests include compiler optimization and binary translation.

Cheng-Yong Wu received his B.S. degree in mathematics from Fudan University, Shanghai, in 1991, M.S. degree in computer science from Beihang University, Beijing, in 1996, and Ph.D. degree in computer architecture from ICT, CAS, Beijing, in 2000. Now he is a professor and Ph.D. supervisor of ICT, CAS. His research interests include compiler optimization and binary translation.