GPU-UniCache: Automatic Code Generation of Spatial Blocking for Stencils on GPUs

Kaixi Hou, Hao Wang, Wu-chun Feng
Department of Computer Science, Virginia Tech
Blacksburg, VA, USA
{kaixihou,hwang121,wfeng}@vt.edu

ABSTRACT

Spatial blocking is a critical memory-access optimization to efficiently exploit the computing resources of parallel processors, such as many-core GPUs. By reusing cache-loaded data over multiple spatial iterations, spatial blocking can significantly lessen the pressure of accessing slow global memory. Stencil computations, for example, can exploit such data reuse via spatial blocking through the memory hierarchy of the GPU to improve performance. However, approaches to take advantage of such blocking require complex and tedious changes to the GPU kernels for different stencils, GPU architectures, and multi-level cached systems.

In this work, we explore the challenges of different spatial blocking strategies over three cache levels of the GPU (i.e., L1 cache, scratchpad memory, and registers) and propose a framework *GPU-UniCache* to *automatically* generate codes to access buffered data in the cached systems of GPUs. Based on the characteristics of spatial blocking over various stencil kernels, we generalize the patterns of data communication, index conversion, and synchronization (with abstracted ISA-friendly interfaces) and map them to different architectures with highly optimized code variants. Our approach greatly simplifies the design of efficient and portable stencil computations across GPUs. Compared to stencil kernels based on hardware-managed memory (L1 cache) and other state-of-the-art GPU benchmarks, the GPU-UniCache can achieve significant improvements.

CCS CONCEPTS

•Computing methodologies → Vector / streaming algorithms;
•Software and its engineering → Domain specific languages;

KEYWORDS

stencil, blocking, SIMD, GPU, AMD, NVIDIA, registers, shuffle, permute, portability, code generation

ACM Reference format:
Kaixi Hou, Hao Wang, Wu-chun Feng. 2017. GPU-UniCache: Automatic Code Generation of Spatial Blocking for Stencils on GPUs. In *Proceedings of CF'17, Siena, Italy, May 15-17, 2017,* 11 pages.
DOI: http://dx.doi.org/10.1145/3075564.3075583

DOI: http://dx.doi.org/10.1145/3075564.3075583

1 INTRODUCTION

Spatial blocking is a critical memory-access optimization that seeks to put spatially reusable data in fast memory (e.g., L1 cache, scratchpad memory, or registers) before actual computation. It has been proven to be effective in utilizing the parallel computing potential of modern accelerators, especially for stencil kernels, where the kernels perform the same computations and data-access patterns over each cell in a multi-dimensional grid. Extensive research efforts have been taken to explore different blocking schemes and develop high-performance stencil programs [7, 26, 30, 31].

In stencil computations, each cell is visited multiple times by its neighbors with the computation sweeping over a spatial grid. Consequently, cache blocking should be done to avoid unnecessary off-chip DRAM loads. Besides global memory, modern GPUs come with multiple low-latency cache levels within each compute unit (CU): (1) *L1 cache*: hardware-managed cache; (2) *scratchpad memory*: fast, programmable memory that is shared by threads assigned to the same CU, but which developers must *explicitly* manage; and (3) *registers*: fastest memory that can be accessed by each thread. In addition, recent GPUs support data exchange between threads in the same wavefront [1] or warp [27].

Different spatial blocking techniques have been proposed for these caches. By using scratchpad memory, one can explicitly load the requisite stencil data into cache from global memory. Then, all the working threads synchronize before doing the actual computation. After the computation, the results are stored back to global memory [26]. With the regularity of access pattern in stencils (based on Cartesian grids), simply relying on the L1 cache can also provide competitive performance [14, 23, 35]. That way, developers only need to focus on the workload partitioning and thread organization. In addition, the advent of register-based data exchange between threads enables each thread to load data into its individual registers and then directly communicate with the threads who own its neighboring data [2, 7, 11].

However, optimizing stencil kernels via spatial blocking introduces three major challenges. *First*, writing blocking code requires substantial coding effort – especially when using registers, as developers must handle the complex and convoluted data communication patterns amongst threads. For stencils with different dimensionalities, where communication patterns must change accordingly, developers must possess extensive coding expertise to reorganize the threads and recalculate the data exchange patterns. *Second*, different GPU architectures have different ISAs, specifications, and run-time configurations – all of which impact the communication patterns, and in turn, lead to rewriting of the stencil codes. For example, the sizes of hardware scheduling unit (e.g., wavefront) and data exchange instructions differ between AMD and NVIDIA

GPUs, causing issues with code portability for the stencil kernels. *Third*, even when a selected stencil is mapped onto a selected GPU, the redesign of the kernel still requires changes in the target cache levels (e.g., scratchpad memory or registers) or blocking strategies (e.g., 2D, 2.5D, or 3D blocking schemes).

While existing stencil frameworks for parallel code generation and performance auto-tuning focus on mapping an entire stencil computation onto an accelerator with dedicated blocking optimizations [24, 34], we focus on a *cross-platform framework* called GPU-UniCache that automatically generates spatial blocking codes for different stencils, GPU architectures, and cache levels, while still allowing developers the option to change their desired stencils. That is, GPU-UniCache analyzes the characteristic parameters of both stencils and GPUs as input and generates highly-optimized blocking codes for the designated cache level. For example, for register-based methods, the GPU-UniCache framework handles the distribution of grid data to minimize register conflict and realizes the communication patterns of given blocking strategies by minimizing the number of permute/shuffle instructions.

The contributions of our work include the following: (1) GPU-UniCache, a framework to automatically generate spatial blocking codes for stencil kernels on GPUs, and (2) a comprehensive evaluation of the GPU-UniCache framework on AMD and NVIDIA GPUs. GPU-UniCache not only improves programming productivity by unifying the interfaces of spatial blocking for different stencils, GPU architectures, and cache levels; but it also provides high performance by optimizing data distribution, indexing conversion, thread communication, and synchronization to facilitate data access in GPU kernels. Compared to hardware-managed memory (L1 cache), with single-precision arithmetic, our automatically-generated codes deliver up to 1.7-fold and 1.8-fold speedups at the scratchpad memory level and register level, respectively, when running on an AMD GCN3 GPU and up to 1.6-fold and 1.8-fold, respectively, when running on a NVIDIA Maxwell GPU. For double precision, it delivers up to a 1.3-fold speedup on both GPU platforms. Compared to the state-of-the-art benchmarks (incl. Parboil [32], PolyBench [29], SHOC [5]), it can also provide up to 1.5-fold improvement.

2 BACKGROUND AND MOTIVATION

2.1 Stencil Computation

A stencil computation defines the point p in a multi-dimensional grid at time t (stored in v) that is updated based on a function f of surrounding grid points $\mathbb{P}$ at the previous time step $t-1$ (stored in u). It sweeps the stencil computation over all the points at t before moving to the next time step $t+1$ and then the next. The stencil order h defines the distance between the central point p and its farthest neighbor $q \in \mathbb{P}$. The stencil size N is $|p \bigcup \mathbb{P}|$. Eq. (1) shows a stencil computation pattern in a 2-dimensional (i.e., 2D) grid; its h is 1; and N is 5. For brevity, we refer to this stencil as "2D5Pt."

$$v_{i,j} = f(\mathbb{P}) = a_0 u_{i-1,j} + a_1 u_{i+1,j} + a_2 u_{i,j} + a_3 u_{i,j-1} + a_4 u_{i,j+1} \quad (1)$$

Due to the application-specific f, there exists no common libraries for stencils that users can directly use without defining the specific stencil patterns. Thus, to evaluate the potential benefits of our GPU-UniCache library framework, we collect a benchmark of stencils representing different dimensionalities and memory-access patterns, as noted in Table 1. Although we distinguish between low and high data-reuse kernels for each dimensionality, their arithmetic intensities (AI), defined as FLOPS/byte [39], are similar.[1] Additionally, data-access patterns differ in that one-dimensional (i.e., 1D) stencils make unit-stride access, whereas higher-dimensional stencils make non-contiguous access of memory. Irrespective of the access pattern, if the data can be ideally cached and reused, the stencil computation will benefit with respect to performance.

Table 1: Summary of the stencil computations

Name	jacobi-1d [29]	gaussian X7 [41]	jacobi-2d [29]	seidel-2d [29]	heat-3d [29]	jacobi-3d [6]
Stencil	1D3Pt	1D7Pt	2D5Pt	2D9Pt	3D7Pt	3D27Pt
h	1	3	1	1	1	1
N	3	7	5	9	7	27
#FLOPS	5	13	9	17	13	53
Bytes	12	28	20	36	28	108
AI	0.42	0.46	0.45	0.47	0.46	0.49

2.2 Spatial Blocking Schemes

In spatial cache-blocking optimizations, one needs to load data into the cache, and then do the stencil computation using the cached data before the results are stored back to global memory. Fig. 1 shows examples of different blocking schemes. A 2D stencil can be optimized by using 2D tiles. Likewise, for 3D stencils, a 3D block is a natural way to buffer data for high reuse. Alternatively, one can use a 2D-slice layout, allowing stencil computations to be carried out from the bottom to the top (i.e., 2.5D blocking). In addition, temporal blocking [26], consisting of multiple rounds of spatial blocking within the cache, can also be used.

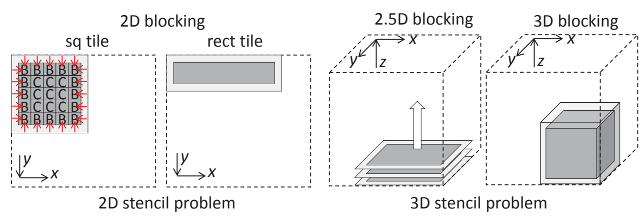


Figure 1: Blocking schemes for 2D and 3D stencils

Fig. 1 shows which data domains are loaded into the cache. However, when designing real GPU kernels, one must explore implementation details, such as how to load domain data. As shown in the figure, when loading a 2D-square tile, the task of loading boundary points (Bs) outside the current tile is assigned to point Bs rather than Cs. This method introduces branch divergence to the GPU kernels. Alternatively, with an (additional) amount of remapping calculation, the data can be evenly assigned to threads (not shown in the figure). In addition, when loading data, one must decide on either a square tile for high data reuse or a rectangle tile for more regular memory access. All the above choices will affect the later realization of fetching data from caches, which in turn, produces significant performance differences (as captured in Fig. 3b).

On the other hand, the temporal cache-blocking essentially adds another dimension (i.e. time) to the spatial blocking by conducting multiple rounds of computations over reusable data (loaded in

[1] We use the *Roofline* model with emphasis on loading data from memory of a machine model without cache.

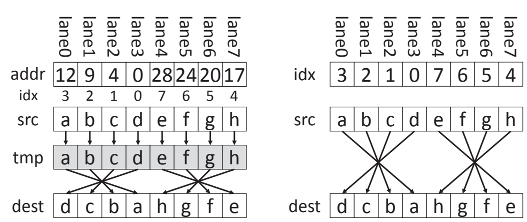
cache). This procedure also follows a fixed or predictable pattern, which matches the idea of our GPU-UniCache framework. However, considering that the spatial blocking is more fundamental and essential in blocking techniques, we focus on analyzing the patterns in spatial blocking for stencils in this paper. Our idea is general and can be used to construct temporal blocking as reported in previous research [26, 31].

2.3 GPU Programming Model and Memory Hierarchy

We use two platforms from different GPU vendors. The first one is HCC[2] for AMD GCN3 (Graphics Core Next) architecture. In the AMD GCN3 GPU, the basic execution unit is called a wavefront (or wave, for brevity) and has 64 lanes. Thus, each thread assigned to a lane ranges from 0 to 63. A wave is assigned to a 16-wide SIMD unit, where each operation takes 4 cycles to finish. The second platform is CUDA (Compute Unified Device Architecture) for NVIDIA GPU, where the basic scheduling unit is a 32-thread warp.

To hide the high latency of off-chip DRAM memory access, modern GPUs possess a cached memory hierarchy. For AMD GCN3, each CU has 16-kB vector L1 cache and 64-kB LDS (Local Data Share) as scratchpad memory. In contrast, each NVIDIA Maxwell streaming multiprocessor (SMM) has a 96-kB scratchpad memory (i.e., shared memory), and its L1 cache is unified with texture cache. Developers use `-Xptxas -dlcm=ca` at compile time to enable L1 cache.

In considering registers as cache, both AMD's CU and NVIDIA's SMM possess 256-kB register files that support cross-lane data sharing. For AMD, this is realized with "permute" instructions, e.g., backward ("ds_bpermute_b32") permute. In contrast, NVIDIA supports a shuffle instruction, i.e., "__shlf". Both "ds_bpermute_b32" and "__shfl" exhibit "pull" semantics, where each thread must read a register value from a target lane. Currently, the GPUs support built-in 32-bit data sharing, while for 64-bit data, one needs to split the data into two values, perform two rounds of data sharing, and then concatenate the results.



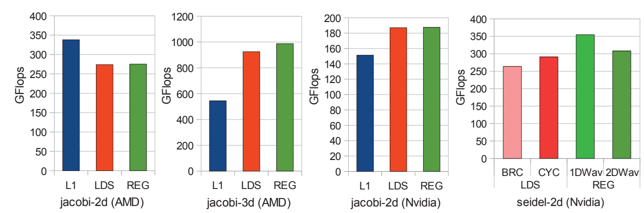
(a) ***ds_bpermute_b32*** in AMD GCN3 (b) ***_shfl*** in Nvidia GPU

Figure 2: Data permute/shuffle in AMD and NVIDIA GPUs. Values in white are stored in registers and the temporary buffer (tmp) is in gray. 8 threads in a wave is for illustration only.

The hardware implementation of data sharing and addressing are different for the two platforms. In AMD GCN3 GPU, LDS is used to route data between the 64 lanes of a wave, but no actual LDS space is consumed [1]. Fig. 2a shows the target values of *src* that will first be put into a *tmp* buffer. Then, the indices are deduced by ignoring the least two significant bits in *addr*, which are used later to select data from *tmp*. In NVIDIA GPU, threads can directly "read" data from another thread that is actively participating in the "__shfl". Fig. 2b shows that each thread can directly access data in another thread based on the given index.

2.4 Challenges

2.4.1 Performance. It is critical to take advantage of the cached memory hierarchy in a GPU via blocking optimizations. Though modern GPUs provide different options, such as L1 cache, scratchpad memory, and registers, it is still unclear where data should be cached for the different stencils. Fig. 3a shows two types of stencils (i.e., jacobi-2d and jacobi-3d) that prefer *different* cache options for the same platform. On the other hand, for the different blocking strategies, developers need to adjust the optimizations to achieve best performance. Fig. 3b shows the diversified performance of "seidel-2d" stencil on two types of caches (i.e., LDS and registers), for each of which we use different loading styles. These simple examples illustrate the challenges encountered by programmers when implementing stencil codes on GPUs. They also demonstrate that choosing a "one-size-fits-all" optimization strategy for any kind of stencil or GPU would be ineffective.



(a) Diff Stencils/GPUs **(b) Diff Variants**

Figure 3: Diversified performance of stencils under different situations. BRC and CYC refer to different loading modes, while 1DWav and 2DWav mean different wave layouts (discussed in Sec. 5).

2.4.2 Programmability. The second challenge encountered by developers is the programmability issue. They might be involved in complex implementation details, where, for example, one needs to figure out how to efficiently organize domain data into individual registers across threads in a wave while using registers as cache. Many other factors can affect how GPU kernel codes are written, including stencil types, GPU architectures, and blocking strategies. To address these issues, we present a framework called GPU-UniCache to automatically generate spatial-blocking codes that manage data reuse within a GPU.

3 GPU-UNICACHE FRAMEWORK

Fig. 4 highlights the major components of our GPU-UniCache framework: (1) feature extraction, (2) code generation, and (3) stencil buffer library. Feature extraction discovers the user-defined conigurations, the stencil types, and the underlying GPU platforms. Code generation automatically produces stencil codes for the different cached systems, i.e. L1 cache, scratchpad memory, and registers. In essence, the codes focus on loading from and storing to the global store, during which GPU-UniCache needs to deal with problems like indexing, synchronization, workload partition, and thread communications. Finally, the stencil buffer library wraps the generated

[2]The Heterogeneous Compute Compiler (HCC) is an open-source C++ compiler for heterogeneous computing. It supports GPUs via HSA-enabled run-time systems and drivers.

codes inside a set of functions with uniform interfaces. We provide details of how our GPU-UniCache library framework works below.

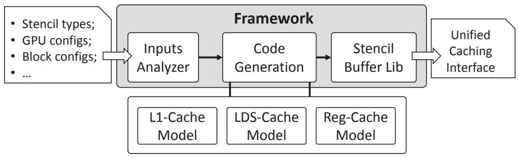


Figure 4: An overview of the GPU-UniCache framework

At the first step, the inputs analyzer component conducts analysis on the user input parameters and some features extracted from the underlying GPU platforms. This information includes stencil types (e.g., stencil order, stencil size), block configs (e.g., block and warp dimensions, blocking strategies), GPU specifications (e.g., built-in warp size, ISAs about data exchange). These parameters assist the framework in realizing the generalized stencil patterns.

The code generation component uses three models on each cache level for given stencils. In L1-cache model, it mainly uses the hardware's capability to access the contiguous data. In scratchpad memory model, it solves the problem of eliminating branches, index conversion under different blocking strategies. In register model, it solves how the data are distributed into registers of each thread, and how the threads communicate with each other to obtain desired neighbors. The generated codes are for three purposes: cache declaration, which allocates required space for scratchpad or registers; cache initialization, which loads central and halo data from global store; cache fetch, which fetches the desired data by using the offsets away from the current point. The codes are finally wrapped into a set of functions by the stencil buffer library component. Developers can call the functions through unified interfaces to design dedicated stencil kernels for efficacy.

3.1 GPU-UniCache API

The GPU-UniCache library provides the operation functions for moving data between on-chip storage and off-chip DRAM memory for stencil computations. Fig. 5 lists the cache classes and their core member functions. The GPU-UniCache API is object oriented. The base class defines interface to initialize the cache, i.e. `init()`, and access the locations with given relative offsets, i.e. `fetch()`. Since all these member functions are executed on GPU devices, we have `__device__` qualifiers for NVIDIA GPU, and `[[hc]]` attribute specifiers for AMD GCN3 GPU. Internally, the classes use `_load()` and `_store()` to access locations in cache using local indices. Subclasses are devised for different cache storage.

```
template<class T>
class GPU_UniCache
{
protected:
  virtual T _load(int z, int y=0, int x=0)=0;
  virtual void _store(T v, int z, int y=0, int x=0)=0;
public:
  virtual void init(T *in, int off, int mode=CYCLIC)=0;
  virtual T fetch(int z, int y, int x, int tc_i=0)=0;
};
// Derived classes
class L1Cache : public GPU_UniCache{ ... };
class LDSCache: public GPU_UniCache{ ... };
class RegCache: public GPU_UniCache{ ... };
```

Figure 5: Interface of GPU-UniCache functions

In Table 2, we list the member functions and corresponding descriptions. Note, all the member functions need location information of the running thread, such as global or local index. For NVIDIA GPU, no specific arguments need to be transferred to the functions, since CUDA supports built-in constants regarding the thread index. For AMD GCN3 GPU, we need to explicitly transfer such information of `tiled_index` by reference. For brevity, we don't list them in the table. In practice, developers create a inherited GPU-UniCache object (e.g., LDSCache) within a device kernel to declare an empty cache space. After the data have been stored into cache, they can use the object to get data in neighbors. We present a working example to show how the GPU-UniCache API works.

Table 2: GPU-UniCache and its subclass member functions

Function Name	Description
(Constructor)(**int** dz, **int** dy, **int** dx, **int** h, **int** tc)	Constructs a specific cache, initializing its attributes of the stencil domain dimensions(dz, dy, dx), order(h), and thread coarsening factor(tc).
_load(**int** z, **int** y, **int** x)	Loads data from cache using local indices(z, y, x). If only z is set, z is the register index.
_store(**T** v, **int** z, **int** y, **int** x)	Stores data(v) to cache using local indices(z, y, x). If only z is set, z is the register index.
init(**T** *in, **int** off, **int** mode)	Initializes the cache from source *in*. The target domain will be located using info. got from the constructor or user-defined offset *off*. Workload distribution can switch by mode, which currently supports CYCLIC and BRANCH styles.
fetch(**int** z, **int** y, **int** x)	Fetches data using the offsets(z, y, x) away from the central point.

3.2 GPU-UniCache Example

We use an example of 2D5Pt GPU kernel (Eq. 1) in Fig. 6 to illustrate how to use the API. This stencil simply uses a 2D blocking optimization strategy and registers as cache. In ln. 4, the kernel declares a RegCache with thread coarsening factor of 4, which means each thread will perform 4 iterations of stencil computation over 4 points. It is demonstrated that using thread coarsening is useful for stencils [2, 22] and we will discuss it in details in Sec. 4. Then, we fill in the register cache by calling `init()` member function. Here, we use the loading mode as CYCLYC in ln. 5, which means the kernel will distribute all the domain data evenly into each thread in a round-robin fashion. While performing the actual stencil computation (ln. 8 to 12), users only need to provide relative offsets of target neighbors and the `fetch()` will figure out where to get the data.

The GPU-UniCache APIs aim to facilitate the process of accessing cached data in stencils on Cartesian grids and allow GPU programmers to develop efficient kernel codes optimized by different blocking strategies. The codes can be easily changed to work on another cache levels for more efficiency. We can also use multiple types of caches at the same time by declaring different GPU-UniCache objects. This could benefit programs which place significantly high resource pressure on a single type of cache. More importantly, the kernel codes are portable across different GPU platforms. We will cover how the GPU-UniCache framework assists in automatically generating the codes for these functions in Sec. 4.

```
__global__
void kern_2d5pt(float *in, float *out, float a0-4)
{
  RegCache<float> buf(m, n, h, 4);
  buf.init(in, 0, CYCLIC);
  // each thread processes 4 points since csr_fct = 4;
  for (csr_id = 0; csr_id < 4; csr_id++)
    out[/*global_idx w/ offset csr_id*/] = a0 * buf.fetch(-1, 0, csr_id)+
                                          a1 * buf.fetch( 0,-1, csr_id)+
                                          a2 * buf.fetch( 0, 0, csr_id)+
                                          a3 * buf.fetch( 0, 1, csr_id)+
                                          a4 * buf.fetch( 1, 0, csr_id);
}
```

Figure 6: Example of 2D5Pt stencil CUDA kernel using RegCache APIs with thread coarsening factor *csr_fct* of 4, which means each thread will update 4 cell points. The current cell point is located by using *csr_id*.

4 CODE GENERATION

In this section, we put emphasis on the register and scratchpad memory methods, since both methods need to explicitly handle how to access the data. For the member functions of sub-classes in Sec. 3, we generate the real codes based on our generalized code constructs and algorithms.

4.1 Input Parameters

The input parameters are used by the framework to understand the features of target stencil and running environment. Table 3 shows the list of the required parameters in three types. Among them, the crs_fct and crs_dim are used specifically for thread coarsening in RegCache methods. RegCache methods handle the computation based on the unit of wave, whose thread number is usually much smaller than a thread block, meaning we need to load more halo data. In contrast, thread coarsening [22] is an optimization technique to increase the workload of each thread and enable loaded data to be more reused. Therefore, we use thread coarsening to compensate low data-reuse rate in RegCache methods.

Table 3: List of input parameters for the code generation

Parameter Name	Description
User-defined thread layout	
blk_dim[3]	Thread block dimensions in exponent notations (with a base of 2). The least significant dimension is blk_dim[0].
wav_dim[3]	Wave dimensions in exponent notations (with a base of 2). The least significant dimension is wav_dim[0].
crs_fct	Thread coarsening factor. It defines the number of iterative process the wave will conduct.
crs_dim	Thread coarsening occurs along which dimension.
Stencil computation characteristics	
h	Stencil order.
N	Stencil size.
sten_dim	Stencil dimensionality.
GPU architecture characteristics	
blk_sync()	Built-in block-level synchronization barrier.
wav_size	Number of threads in a wave.
wav_shfl(v, id)	Machine-dependent register-level data exchange instruction. Data exchange occurs between calling thread and thread *id* on value *v*.

4.2 RegCache Methods

We first look at a specific example of "2D9Pt" stencil and analyze its data distribution and computation patterns. Fig. 7 shows a wave with thread layout of $2 \times 4 = 8$ (i.e., $wav\_dim[1] = 1$ and $wav\_dim[0] = 2$) loads required grid points of $4 \times 6 = 24$ (i.e., $h = 1$). The 24 points are distributed evenly into registers of each thread in a round-robin fashion, meaning $\lceil 24/8 \rceil = 3$ iterations and registers are needed. To achieve this CYCLIC loading method, we map these threads to assigned points by using $(y, x) = ((i \cdot wav\_size + tid)/(2^{wav\_dim[0]} + 2h), (i \cdot wav\_size + tid)\%(2^{wav\_dim[0]} + 2h))$, where tid is thread index and i is iteration number. Therefore, for example, thread 0 will deal with points (0,0), (1,2), (2,4) and store them in register r, s, t respectively.

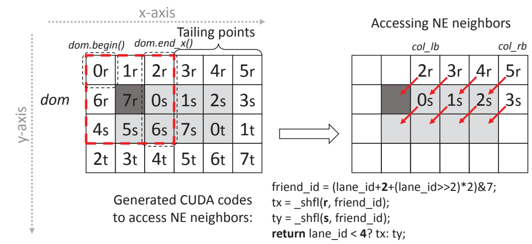


Figure 7: Example of data exchange for "2D9Pt" stencil. The figure illustrates 2 steps of *loading* and *computing*. For *loading*, all the data are distributed across threads in a 2x4 wave. The '2r' in cell, for example, means the corresponding value will be stored in register r of thread 2. For *computing*, the wave updates the gray area. The thread 0 in the wave is in deep gray. The CUDA codes below are to access the NE neighbors.

The destination points (gray area) are updated by fetching registers from their neighbors. However, this raises two further questions: 1, which threads to communicate with; 2, which registers store the desired neighbors. We observe from Fig. 7 that these information can be calculated from neighbors of thread 0 in the wave (located in the red circle). For example, when handling the northeast (NE) neighbors, we need to know the first neighbor is stored in register 0 (r) of thread 2. Then, the other neighbor thread index and register index can be calculated by each thread applying $(tid + 2 + tid/2^{wav\_dim[0]} \cdot 2h)\%wav\_size$ and $0 + (2 + tid/2^{wav\_dim[0]} \cdot 2h + tid)/wav\_size$ respectively. That way, thread 0 will interact with thread 2 on register 0 (r), and simultaneously, thread 4 will fetch value of register 1 (s) of thread 0.

With the variety of stencils and options (e.g., thread coarsening factors and neighbor directions), manually calculating these parameters is a painful task. As size and complexity of the target stencil grow, so does the development cost. Therefore, in our framework, we first generalize the stencil computation in registers by means of code constructs. Then, we calculate the parameters using our proposed formula and algorithm.

Method `init()`: we only support loading method of CYCLIC rather than BRANCH in RegCache. The reasons are two-fold: (1) BRANCH mode will make boundary threads hold too many registers and thus all the other threads in the same wave have to keep same number of "idle" registers, leading to register pressure problem; (2) While accessing neighbors, extra branches are needed to distinguish the meaningful from these "idle" registers. Code constructs in Fig. 8 show how we distribute the DRAM data to registers. The remapping occurs in ln. 3 to 6 and the fetched data are stored to registers (ln. 8).

Method `fetch()`: Fig. 9 exhibits the generalized data exchange code constructs to fetch data in given direction. The neighbor thread index is represented by *friend_id*, which depends on the parameter `F`. The registers of interest are ranged from *reg*`N1` to *reg*`N3`. Parameter `M` is the cut-off marker to select values from different registers. Here, we only use up to three data exchange operations to fetch the data, since this number fits in our benchmark of stencils and

```
// **** CYCLIC ****
int it = _lane_id();
c_0 = (wav_id0<<wav_dim[0]) + it%(2^wav_dim[0] + 2h);
c_1 = (wav_id1<<wav_dim[1]) + it/(2^wav_dim[0] + 2h)
                              %(2^wav_dim[1] + 2h);
c_2 = (wav_id2<<wav_dim[2]) + it%(∏_{k=0}^{1}(2^wav_dim[k] + 2h));
reg_id = 0;
_store(in(off,c_2,c_1,c_0),reg_id++);
it += wav_size;
// repeat for ⌈∏_{k=0}^{2}(2^wav_dim[k] + 2h)/wav_size⌉ times
```

Figure 8: Code constructs for RegCache `init()` method

different wave dimensions. For other stencils with higher stencil order, for example, it is easy to extend the pattern to support more data exchange operations.

```
1 // **** Fetch a given neighbor ****
2 friend_id = (lane_id+F+
3            ((lane_id>>wav_dim[0]*2*h))&(wav_size-1);
4 tx = wav_shfl(regN1, friend_id);
5 ty = wav_shfl(regN2, friend_id);
6 tz = wav_shfl(regN3, friend_id);
7 return ((lane_id < M1)? tx: ((lane_id < M2)? ty: tz));
```

Figure 9: Code constructs for RegCache `fetch()` method

Fig. 10 shows the pseudo code of calculating the parameters based on given inputs from Table 3 (Each direction of neighbors need a set of the parameters). We define a *domain* as a set of points surrounding the first thread in a wave. Since threads might be coarsened by the factor of `crs_fct`, there are multiple domains stored in `dom` (ln. 1 and 17). In the function calculating parameter `F` (ln. 1), the identifier of the starting point in each domain is computed in ln. 8. Then, we sweep all the other points and record the relative order within the wave (ln. 9). The order is the parameter `F`, which can be used later by other threads in the wave to find neighbors towards the same direction (ln. 2 in Fig. 9). Additionally, we record the round number in ln. 10, indicating how many registers we have already used in each thread. Note, the out-of-domain points should be skipped in ln. 12 to 14.

Subsequently, we need to calculate which registers are used to store the target neighbors in the wave and how to select data from these registers. This can be achieved by computing parameters `N` and `M` through the function in ln. 17. The register identifier in ln. 24 indicates the register storing the first value of neighbors toward the given direction. Then, we can calculate the boundaries of neighbors of the entire wave (ln. 27 and 28, also shown in Fig. 7) which will be used to skip other irrelevant points. If an incoming point is identified as using a new register in ln. 36 and it is within the boundary in ln. 38, the new register is recorded with the counter `cnt` showing the cut-off location.

After we calculate these parameters, we replace $wav\_shfl()$ with "__shfl()" for NVIDIA GPUs and "amdgcn_bs_bpermute()" for AMD GCN3 GPUs. Note, for AMD GCN3 GPUs, we need to right shift the *friend_id* by 2 (Sec. 2.3). If the datatype is double precision number, we will first split the value into two 32-bit ones, perform two data exchange instructions, and then concatenate the results.

4.3 LDSCache Methods

Method `init()`: The major problem encountered by using scratchpad memory is conditional branching, since the sizes of thread block and working data domain don't match each other. In LDSCache, we

```
1  void calculate_F(domain* dom)
2  { // compute param F used in friend_id formula
3    for(int c = 0; c < csr_fct; c++)
4    {
5      for(auto pt: dom[c]) // each point in domain
6      {
7        if(pt == dom[c].begin())
8          id = c * ∏_{k=0}^{csr_dim-1}(2^wav_dim[k] + 2h);
9        pt.F  = id % wav_size;
10       pt.rid = id / wav_size;
11       id++;
12       if(pt == dom[c].end_x()) // skip tailing points
13         id += 2^wav_dim[0];
14       if(pt == dom[c].end_yx()) //skip tailing lines
15         id += 2^wav_dim[1] · (2^wav_dim[0] + 2h);
16 } } }
17 void calculate_NM(domain* dom)
18 { // compute param N M used in data exchange patterns
19   for(int c = 0; c < csr_fct; c++)
20   {
21     for(auto pt: dom[c]) // each point in domain
22     {
23       int i = 1, j = 1;
24       int reg_id = pt.rid;
25       pt.N[i++] = reg_id;
26       int skipped_pts = pt.F + reg_id * wav_size;
27       int col_lb = skipped_pts % (2^wav_dim[0] + 2h);
28       int col_rb = lb + 2^wav_dim[0];
29       int cnt = 1;
30       bool reg_update = false;
31       while(cnt < wav_size)
32       {
33         skipped_pts++;
34         int col_id = skipped_pts % (2^wav_dim[0] + 2h);
35         int wav_id = skipped_pts % wav_size;
36         if(wav_id == 0) // end of current wave
37           reg_id++, reg_update = true;
38         if(col_lb <= col_id && col_id < col_rb)
39         {
40           if(reg_update) // mark the divergence
41           {
42             pt.N[i++] = reg_id;
43             pt.M[j++] = cnt;
44             reg_update = false;
45 } } } } } }
```

Figure 10: Algorithms to calculate the `F`, `N`, and `M`.

support two loading modes: BRANCH, boundary threads handle more workloads (i.e. halo points); CYCLIC, threads address the data domain in a round-robin fashion by remapping themselves. This way, we can minimize the branches at the expense of more index conversion calculation. The code constructs of BRANCH are comprised of multiple conditional statements to assign additional workloads to boundary threads. The CYCLIC code constructs are similar with RegCache method (in Fig. 8), but replaced with the granularity of *blk_size* rather than *wav_size*. In addition, the destination locations are changed to scratchpad memory. Note, we need to use an explicit synchronization *blk_sync()* at the end of loading.

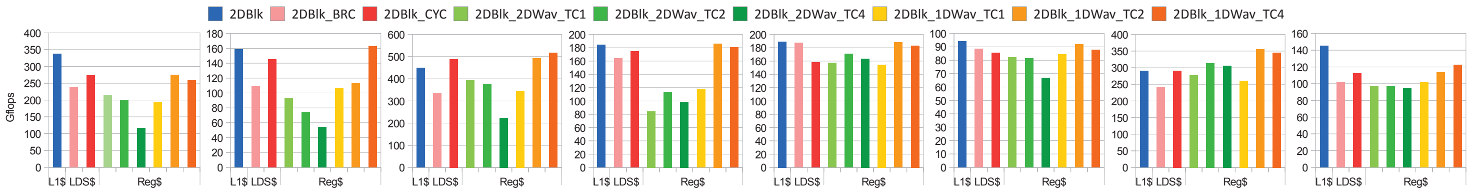
Method `fetch()`: This method is straightforward and we only need to use the thread local index to fetch desired data, since the loaded points follow original data layouts and are same by using BRANCH or CYCLIC mode.

5 EVALUATION

5.1 Experiment Setup

In the section, we evaluate the stencil codes using GPU-UniCache library. The details of the two platforms are listed in Table 4. We conduct the tests using both single precision and double precision numbers.

The benchmark of stencils are listed in Sec. 2.1. We optimize them using the GPU-UniCache APIs with different blocking strategies.



(a) jacobi-2d *l*: single; *r*: double (HCC) (b) seidel-2d *l*: single; *r*: double (HCC) (c) jacobi-2d *l*: single; *r*: double (CUDA) (d) seidel-2d *l*: single; *r*: double (CUDA)

Figure 11: 2D stencils with HCC and CUDA by GPU-UniCache on AMD and NVIDIA GPUs respectively

Table 4: Experiment Testbeds

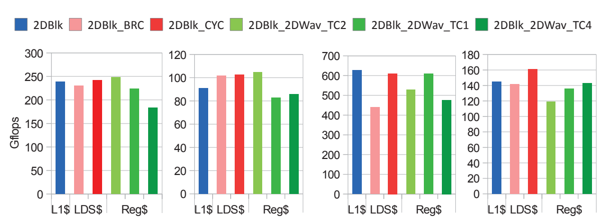
	AMD	NVIDIA
Model	Radeon R9 Nano	GeForce GTX 980
Codename	Fiji XT	GM204(Maxwell)
Cores	4096	2048
Core frequency	1000 MHz	1126 MHz
Register file size	256 kB*	256 kB
L1/LDS/L2	16/64/1024 kB	-/96/2048 kB
Memory bus	HBM	GDDR5
Memory capacity	4096 MB	4096 MB
Memory BW	512 GB/s	224 GB/s
GFLOPS float/double	8192/512	4612/144
Software	HCC/ROCM 1.2	CUDA 7.5

* Each CU has 256 kB vector registers and an additional 8 kB scalar registers.

The blocking strategies used in 1D and 2D stencils are straightforward, while in 3D kernels, we use 2.5D and 3D blocking [26] (labeled as 1DBlk, 2DBlk, 2.5DBlk, and 3DBlk). For LDSCache version, we try both loading modes: BRANCH and CYCLIC (as BRC and CYC), while for RegCache, we vary dimensionalities of wave: 1D and 2D (as 1DWav and 2DWav). The 1DWav is 64×1 for AMD and 32×1 for NVIDIA, while the 2DWav is 8×8 and 8×4 on the two platforms. The sizes of data set are 2^{25}, $2^{12} \times 2^{12}$, $2^{8} \times 2^{8} \times 2^{8}$ for 1D, 2D, 3D stencils respectively. The test iterates for 100 times. The metric we use is GFLOPS calculated by $(FLOPS \cdot dim2 \cdot dim1 \cdot dim0)/time$.

5.2 AMD GCN3 GPU

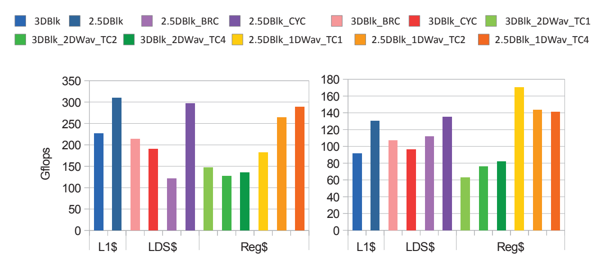
We use the best speedup achievable when the kernel is optimized by RegCache or LDSCache, if not mentioned otherwise. For 1D stencils, the performance numbers are shown in Fig. 12. Different cache levels show very similar performance. Using LDSCache or RegCache do not show significant improvements over L1Cache, because 1D stencil has unit-stride memory access pattern where the data can be effectively put into cache by hardware. The optimal achieved with RegCache leads to 15% improvement; LDSCache achieves up to 13% improvement. We also notice that performance deteriorates with LDSCache in BRANCH mode for gaussianX7 stencil, due to extra loading operations to perform data transfer between L1 cache to scratchpad memory and overhead of branches, which can be offset by using CYCLIC mode.



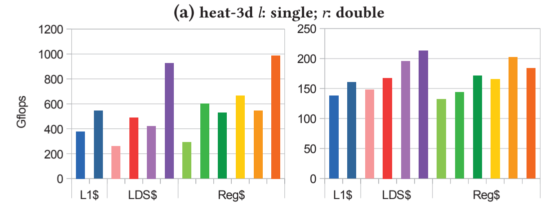
(a) jacobi-1d *l*: single; *r*: double (b) gaussianX7 *l*: single; *r*: double

Figure 12: 1D stencils with HCC by GPU-UniCache on AMD GPU

For 2D stencils, L1Cache methods still exhibit competitive performance on AMD GCN3 GPUs (shown in Fig. 11a and 11b). The maximum speedups with LDSCache and RegCache surpass L1Cache when data reuse grows in seidel-2d stencil. In the LDSCache solution, we first observe that 2D stencils are more sensitive to the loading mode, where CYCLIC mode is generally superior to BRANCH mode averaging 25% better performance, since more branches are needed to load surrounding data in 2D stencils. The maximum improvement of LDSCache over L1Cache is 9%. In RegCache solution, using 1D wave variant is particularly effective over 2D wave. 1D wave have longer dimension while conducting memory access, which can better utilize the hardware bandwidth but at the expense of relatively low data reuse. 2D wave, by contrast, exhibits high data reuse rate. For example, considering the wave size of 64 on AMD GPUs, if we organize the wave threads as 64 by 1, we have to load 66*3=198 elements for the 2D problem with stencil order of 1. However, if we organize them as 8 by 8, we only need to load 9*9=81 elements. On the other hand, the former thread layout can load the data in less memory transactions, leading to its superior performance. If we consider the effect of thread coarsening on performance, 2D wave can barely benefit from it, because the narrowed access stride makes it bound by memory latency. We record the best speedup of RegCache is 15% over L1Cache.



(a) heat-3d *l*: single; *r*: double



(b) jacobi-3d *l*: single; *r*: double

Figure 13: 3D stencils with HCC by GPU-UniCache on AMD GPU

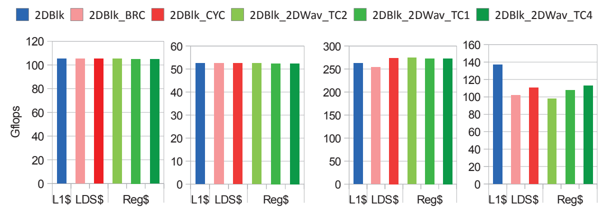
Fig. 13 shows more significant and diversified speedups with RegCache and LDSCache. Differences in the performance are first reflected in the speedups of the 2.5D and 3D blocking. 2.5D blocking gives a speedup of 1.58x over 3D blocking on average. This is because 3D blocking has smaller dimensions for the block than 2.5D blocking if we assume the blocks have same number of threads. That way, uncoalesced memory access would occur even though it has better data reuse rate. In the low data-reuse kernels (heat-3d), L1Cache solution is similar with LDSCache, while the additional gain is achieved from using RegCache, resulting in up to 30% improvement. This is mainly because of the elimination of explicit synchronization of RegCache in this iterative 2D method. For high data-reuse kernels (jacobi-3d), LDSCache or RegCache are critical to get optimal performance. The best improvements are 1.70x for LDSCache and 1.81x for RegCache over their L1Cache counterpart. Moreover, we prefer CYCLIC mode in LDSCache in high data-reuse kernels, observing that the overhead of branches is significantly high, because, for example, the jacobi-3d stencil has nearly 4 times more halo elements to load than the heat-3d stencil. For RegCache solutions, we only record the performance of 2D wave in 3D blocking, because using 1D wave instead would be equivalent to the 2.5D blocking with 1D wave. Also, we only show the results of 1D wave for 2.5D blocking, since this strategy is preferable and has been demonstrated. Similarly, 3D blocking encounters higher memory latency, making itself benefit little from thread coarsening. As a contrast, 2.5D blocking with 1D wave improves significant after applying thread coarsening.

The speedups given by thread coarsening in the cases of double precision numbers are less consistent, where the optimal thread coarsening factors are only 1 or 2, since operating doubles requires more space from register files and register pressure would be more easily reached. Furthermore, because the built-in data exchange of 64-bit data is not supported, we need more operations to achieve the same functionality, i.e. split the data into two 32-bit data, do two permutes, and concatenate the two data together.

5.3 NVIDIA Maxwell GPU

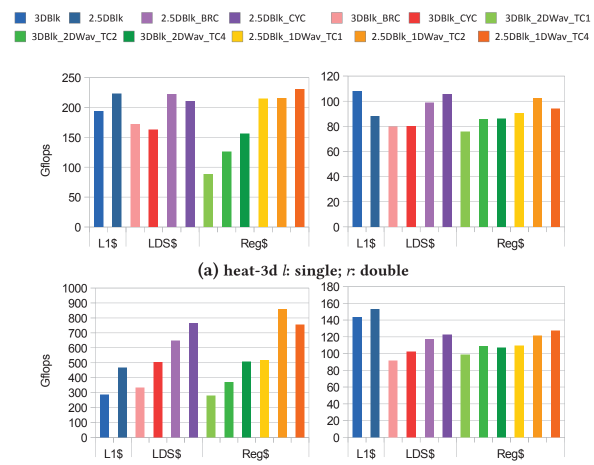
On NVIDIA GPU, Fig. 14, 11c and 11d show the performance of 1D and 2D stencils on Maxwell GPU. For low data reuse kernels, the optimal is achieved by simply using L1Cache. This demonstrates the need to "opt-in" to enable the global caching in the Maxwell GPU (Sec. 2.3), which is particularly effective for solving 1D and 2D arrays. The benefits of using LDSCache or RegCache become obvious when there are high data reuse, achieving up to 5% and 20% improvements for gaussianX7 and seidel-2d stencils respectively. However, for double datatype, we observe a slowdown experienced by LDSCache and RegCache. For LDSCache, since the shared memory banking in Maxwell only supports 4 bytes width per bank, overhead of accessing 8-byte data is accordingly higher; for RegCache, more instructions are needed to conduct every data exchange operations for 8-byte data. Similar to 2D stencils on GCN3 GPU, CYCLIC mode is of necessity in seidel-2d kernels and 1D wave is preferable because all the threads in the same wave are able to access consecutive locations to achieve a coalesced memory transaction.

The performance of 3D stencils shown in Fig. 15 shows diversified speedups after applying different cache levels. Speedups of using LDSCache or RegCache range from a few percent on the low data reuse kernels up to 1.64x and 1.83x for high data reuse kernels with LDSCache and RegCache respectively. The 2.5D blocking is still preferred in the 3D stencils and for float datatype, we record 4% to 12% improvements of the best RegCache over LDSCache. 2.5D blocking needs to iteratively load a 2D slice before conducting actual computation, which will result in overhead of block-level synchronization. In contrast, RegCache can eliminate this explicit synchronization, leading to better performance. For double datatype, using our L1Cache interface can provide competitive performance, mainly because the overhead of operating doubles in RegCache and LDSCache is relatively high in Maxwell.



(a) jacobi-1d *l*: single; *r*: double (b) gaussianX7 *l*: single; *r*: double

Figure 14: 1D stencils with CUDA by GPU-UniCache on NVIDIA GPU



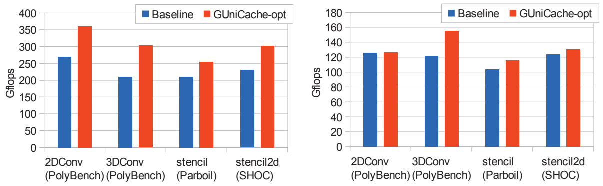
(a) heat-3d *l*: single; *r*: double

(b) jacobi-3d *l*: single; *r*: double

Figure 15: 3D stencils with CUDA by GPU-UniCache on NVIDIA GPU

5.4 Speedups to Existing Benchmarks

In the section, we optimize third-party benchmarks by using GPU-UniCache. They have been optimized via different spatial blocking strategies: *2DConv* and *3DConv* (PloyBench [29]) use 2D and 3D blocking with L1 cache respectively; *stencil* (Parboil [32]) is a "3D7Pt" stencil optimized by 2.5D blocking with shared memory; *stencil2d* (SHOC [5]) adopts 2D blocking with shared memory. We optimize these kernels by using GPU-UniCache and only report the best performance. Fig. 16 presents the results of the comparisons on NVIDIA GPU (There are no equivalent benchmarks using HCC yet). For single datatype, all the optimal GPU-UniCache codes are using RegCache and can outperform the baselines for up to 1.5x. For double datatype, GPU-UniCache selects L1Cache for 2D stencils and LDSCache for 3D stencils, mainly because the overhead of register shuffle on double grows. The best improvement is as high as 1.3x speedup.



(a) single datatype **(b) double datatype**

Figure 16: GPU-UniCache optimized codes vs. existing stencil benchmarks optimized by spatial blocking on NVIDIA Maxwell GPU

5.5 Discussion

Running Parameters In the experiments, we use the same settings for the kernels to evaluate the performance for two main reasons. First, we can limit the variables to the options of cache levels and focus on the correlation between performance and different GPU-UniCache functions. One exception is that we need to shrink the total number of threads as the thread coarsening factor grows up in RegCache kernels. Second, the GPU-UniCache APIs are also designed to enable GPU programmers access to the different caches simultaneously, especially when the programs encounter high pressure on one single type of resource. Therefore, we need to test the APIs under the same circumstances. Despite of this, we still observe the diversified speedups, indicating an auto-tuning framework is of necessity [10, 21]. We leave this as our future work.

Register Pressure Using too many registers in GPU programs could reduce the active waves per CU. Table 5 shows the profiling numbers of register usage from the jacobi-3d stencil which exhibits the highest data reuse rate. First, as the coarsening factor doubles, the number of registers increases logarithmically, because coarsening technique can improve data reuse. Additionally, 2.5D blocking generally uses more registers than 3D blocking as we discussed in Sec. 5.2. The kernel of 2.5D blocking with the best performance can attain 40% occupancy on both GPU platforms. However, considering its better memory access and high FLOPs, the active waves can still effectively utilize the computing resources.

Table 5: Register usage of jacobi-3d stencil*

GPU	L1Cache		LDSCache				RegCache					
	3DBlk	2.5DBlk	3DBlk BRC	3DBlk CYC	2.5DBlk BRC	2.5DBlk CYC	3DBlk 2DWav TC1	3DBlk 2DWav TC2	3DBlk 2DWav TC4	2.5DBlk 1DWav TC1	2.5DBlk 1DWav TC2	2.5DBlk 1DWav TC4
AMD	37	101	21	18	40	31	19	31	48	48	59	74
NVIDIA	32	32	30	28	31	31	32	40	56	42	56	80

* Collected by CodeXL 2.2 for AMD GPU and nvprof tool for NVIDIA GPU

6 RELATED WORK

Since GPU has been a part of general-purpose accelerators, there are many efforts to optimize stencil computation for high performance. Memory access is a key problem, which was addressed by many research endeavors. Nguyen *et al.* [26] focus on a novel 3.5D blocking optimization(temporal reuse with 2.5D blocking). Rawat *et al.* [30] propose an effective tiling strategy to utilize both scratchpad memory and registers for 2D/3D stencils. Vizitiu *et al.* [35] locate reusable data to constant cache, shared memory, etc., to explore performance variance between different NVIDIA GPUs. Falch and Elster [7] optimize 1D/2D stencils using registers as buffer with manually written shuffle operations. For large-scaled problems, the key challenge shifts to communication across compute nodes [25, 28, 38]. The focus of our work is different, as we are providing a framework for accessing neighboring data on different cache levels. It utilizes the knowledge of patterns to automatically conduct data distribution, synchronization, and communication for different stencils.

In order to enable the parallel codes cross-platform, developers prefer to use OpenCL and OpenACC for GPU programming [15, 29]. However, our framework is not based on them, because the current versions of OpenCL in both AMD and NVIDIA GPUs fall short from supporting some of the latest features, such as shuffle/permute (nor in OpenACC). In contrast, these features are well supported by AMD and NVIDIA's own programming languages (i.e., HCC and CUDA). In our GPU-UniCache, we rely on them to design our RegCache methods. For other methods (i.e. L1Cache and LDSCache), although they can be ported to OpenCL using local memory or to OpenACC using tile-clause, one has to explicitly change the logic of kernel codes or loop structures to switch between different cache hierarchies.

Another way to achieve performance portability is based on code generation from parallel patterns and DSLs, such as [12, 13]. Unlike the irregular algorithms (due to either dataset properties [17–19, 37] or algorithms themselves [40, 42]), the target kernels usually exhibit fixed or predictable computational motif. Apparently, stencils belong to this category. Krishnamoorthy *et al.* [16] propose an approach to automatically parallelize stencils codes with emphasis on loop skewing to handle the load imbalance issues. Cui *et al.* [4] focus on directive-based solution to overload computation and communication in GPU stencils. Luo and Tan [20] propose a tool to apply locality optimization on stencil loops to utilize computing resources. Many DSLs are used to describe stencil computations [3, 21]. With DSLs, there are code generation frameworks and specialized compilers aiming at generating efficient parallel codes on GPUs [8, 9, 30, 33, 34]. The knowledge is also extended to large-scaled clusters. Wahib and Maruyama [36] devise framework to transform CUDA kernels to large-scaled GPU clusters. Physis [24] is a programming framework for supercomputers with emphasis on computation and communication overlapping. Furthermore, considering the large set of stencils and architectures, different auto-tuning frameworks are proposed to search for the best optimizations for given stencils [21, 43]. Overall, the key distinctive aspects of our work are: 1) moving the abstraction to a lower level with emphasis on designing a unified and portable interface to efficiently access simulation cells in GPU cached systems, 2) exploiting new data shuffle/permute instructions, 3) using knowledge of patterns of different spatial blocking. By using GPU-UniCache generated codes, developers are still in tight control of designing specific stencil kernels.

7 CONCLUSION

In the paper, we propose a framework GPU-UniCache to automatically generate the library codes to access cached data L1, scratchpad memory, and registers of the spatial blocking optimizations for stencils computations. The codes to achieve these functionalities are automatically generated by our GPU-UniCache framework based on the information of stencils and underlying architectures. The GPU-UniCache has facilitated efficiently accessing cache-loaded data without a tedious code rewrite, a major advantage in designing different stencil codebases. The evaluation demonstrates that we can get up to 1.8x improvements by only changing the GPU-UniCache API calls on different GPU platforms.

8 ACKNOWLEDGEMENTS

First and foremost, we thank Jonathan Gallmeier and Shuai Che for their stimulating discussion and feedback on this work. Thanks also to the anonymous reviewers for their comments and feedback. The authors also acknowledge Advanced Research Computing (ARC) at Virginia Tech for providing leading-edge computational resources.

This work was initially supported in part by the Air Force Office of Scientific Research (AFOSR) Basic Research Initiative via Grant No. FA9550-12-1-0442 from the Computational Mathematics Program and then by the NSF BIGDATA program via IIS-1247693.

REFERENCES

[1] AMD. 2016. Graphics Core Next Architecture, Generation 3 Reference Guide. (2016). Revision1.1.

[2] E. Ben-Sasson, M. Hamilis, M. Silberstein, and E. Tromer. 2016. Fast Multiplication in Binary Fields on GPUs via Register Cache. In *ACM Int'l Conf. on Supercomputing (ICS)*.

[3] M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *IEEE Int'l Parallel & Distrib. Process. Symp. (IPDPS)*.

[4] X. Cui, T. RW Scogland, B. R de Supinski, and W.-c. Feng. 2016. Directive-Based Pipelining Extension for OpenMP. In *IEEE Int'l Conf. Cluster Computing (CLUSTER)*.

[5] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *ACM Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*.

[6] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. 2009. Auto-tuning the 27-point Stencil for Multicore. In *Int'l Workshop on Automatic Performance Tuning (iWAPT)*.

[7] T. L. Falch and A. C. Elster. 2014. Register Caching for Stencil Computations on GPUs. In *Int'l Symp. on Symbolic and Numeric Algorithms for Sci. Comp. (SYNASC)*.

[8] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A Stencil Compiler for Short-vector SIMD Architectures. In *ACM Int'l Conf. on Supercomputing (ICS)*.

[9] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. 2012. High-performance Code Generation for Stencil Computations on GPU Architectures. In *ACM Int'l Conf. on Supercomputing (ICS)*.

[10] K. Hou, W.-c. Feng, and S. Che. 2017. Auto-Tuning Strategies for Parallelizing Sparse Matrix-Vector (SpMV) Multiplication on Multi- and Many-Core Processors. In *IEEE Int'l Parallel & Distributed Processing Symp. Workshops (IPDPSW)*.

[11] K. Hou, W. Liu, H. Wang, and W.-c. Feng. 2017. Fast Segmented Sort on GPUs. In *ACM Int'l Conf. on Supercomputing (ICS)*.

[12] K. Hou, H. Wang, and W.-c. Feng. 2015. ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors. In *ACM Int'l Conf. on Supercomputing (ICS)*.

[13] K. Hou, H. Wang, and W.-c. Feng. 2016. AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors. In *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*.

[14] K. Hou, Y. Zhao, J. Huang, and L. Zhang. 2011. Performance Evaluation of the Three-Dimensional Finite-Difference Time-Domain(FDTD) Method on Fermi Architecture GPUs. In *Int'l Conf. on Algorithms and Architectures for Parallel Processing (ICA3PP)*. Springer Berlin Heidelberg.

[15] Q. Jia and H. Zhou. 2016. Tuning Stencil codes in OpenCL for FPGAs. In *IEEE Int'l Conf. on Computer Design (ICCD)*.

[16] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. In *ACM SIGPLAN Conf. on Programming Language Design and Impl. (PLDI)*.

[17] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter. 2016. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In *Int'l Conf. on Euro-Par 2016: Parallel Processing - Volume 9833*. Springer-Verlag New York, Inc.

[18] W. Liu and B. Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ACM Int'l Conf. on Supercomputing (ICS)*.

[19] W. Liu and B. Vinter. 2015. A Framework for General Sparse Matrix-matrix Multiplication on GPUs and Heterogeneous Processors. *J. Parallel Distrib. Comput. (JPDC)* (2015).

[20] Y. Luo and G. Tan. 2014. Optimizing Stencil Code via Locality of Computation. In *ACM Int'l Conf. on Parallel Architectures and Compilation (PACT)*.

[21] Y. Luo, G. Tan, Z. Mo, and N. Sun. 2015. FAST: A Fast Stencil Autotuning Framework Based On An Optimal-solution Space Model. In *ACM on Int'l Conf. on Supercomputing (ICS)*.

[22] A. Magni, C. Dubach, and M. F. P. O'Boyle. 2013. A Large-scale Cross-architecture Evaluation of Thread-coarsening. In *ACM/IEEE Int'l Conf. on High Perf. Computing, Networking, Storage and Analysis (SC)*.

[23] N. Maruyama and T. Aoki. 2014. Optimizing stencil computations for NVIDIA Kepler GPUs. In *Int'l Workshop on High-Performance Stencil Comp. (HiStencils)*.

[24] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. 2011. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers. In *ACM/IEEE Int'l Conf. for High Perf. Computing, Networking, Storage and Analysis (SC)*.

[25] J. E. McClure, H. Wang, J. F. Prins, C. T. Miller, and W.-c. Feng. 2014. Petascale Application of a Coupled CPU-GPU Algorithm for Simulation and Analysis of Multiphase Flow Solutions in Porous Medium Systems. In *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*.

[26] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 2010. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *ACM/IEEE Int'l Conf. for High Perf. Comp., Networking, Storage and Analysis (SC)*.

[27] NVIDIA. 2014. NVIDIA's Next Generation CUDA Compute Architecture Kepler GK110. (2014). v1.0.

[28] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda. 2012. Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication. In *IEEE Int'l Parallel & Distributed Processing Symp. Workshops PhD Forum*.

[29] L.-N. Pouchet. 2015. Polybench: The polyhedral benchmark suite. (2015). http://web.cse.ohio-state.edu/~pouchet/software/polybench.

[30] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, A. Rountev, and P. Sadayappan. 2016. Resource Conscious Reuse-Driven Tiling for GPUs. In *ACM Int'l Conf. on Parallel Architectures and Compilation (PACT)*.

[31] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, and P. Sadayappan. 2016. Effective Resource Management for Enhancing Performance of 2D and 3D Stencils on GPUs. In *ACM Workshop on General Purpose Processing Using Graphics Processing Unit (GPGPU)*.

[32] J. A Stratton, C. Rodrigues, I-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and HPC* (2012).

[33] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. 2011. The Pochoir Stencil Compiler. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*.

[34] D. Unat, X. Cai, and S. B. Baden. 2011. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *ACM Int'l Conf. on Supercomputing (ICS)*.

[35] A. Vizitiu, L. Itu, C. Nifl, and C. Suciu. 2014. Optimized Three-dimensional Stencil Computation on Fermi and Kepler GPUs. In *High Performance Extreme Computing Conf. (HPEC)*.

[36] M. Wahib and N. Maruyama. 2015. Automated GPU Kernel Transformations in Large-Scale Production Stencil Applications. In *ACM Int'l Symp. on High-Performance Parallel and Distributed Computing (HPDC)*.

[37] H. Wang, W. Liu, K. Hou, and W.-c. Feng. 2016. Parallel Transposition of Sparse Data Structures. In *ACM Int'l Conf. on Supercomputing (ICS)*.

[38] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda. 2014. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2014).

[39] S. Williams, A. Waterman, and D. Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Comm. of the ACM* (2009).

[40] X. Yu and M. Becchi. 2013. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *ACM Int'l Conf. on Computing Frontiers (CF)*.

[41] F. Zhang. 2014. Automatic Loop Tuning and Memory Management for Stencil Computations. (2014). http://scholarcommons.sc.edu/etd/3012 (Doctoral Dissertation).

[42] J. Zhang, H. Wang, H. Lin, and W. c. Feng. 2014. cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on a GPU. In *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*.

[43] Y. Zhang and F. Mueller. 2012. Auto-generation and Auto-tuning of 3D Stencil Codes on GPU Clusters. In *ACM Int'l Symp. on Code Generation and Opt. (CGO)*.