OpenACC cache Directive: Opportunities and Optimizations

Ahmad Lashgar
Electrical and Computer Engineering Department
University of Victoria
lashgar@uvic.ca

Amirali Baniasadi
Electrical and Computer Engineering Department
University of Victoria
amiralib@ece.uvic.ca

Abstract*—OpenACC's programming model presents a simple interface to programmers, offering a trade-off between performance and development effort. OpenACC relies on compiler technologies to generate efficient code and optimize for performance. Among the difficult to implement directives, is the *cache* directive. The *cache* directive allows the programmer to utilize accelerator's hardware- or software-managed caches by passing hints to the compiler. In this paper, we investigate the implementation aspect of *cache* directive under NVIDIA-like GPUs and propose optimizations for the CUDA backend. We use CUDA's shared memory as the software-managed cache space. We first show that a straightforward implementation can be very inefficient, and downgrade performance. We investigate the differences between this implementation and hand-written CUDA alternatives and introduce the following optimizations to bridge the performance gap between the two: i) improving occupancy by sharing the cache among several parallel threads and ii) optimizing cache fetch and write routines via parallelization and minimizing control flow. We present compiler passes to apply these optimizations. Investigating three test cases, we show that the best cache directive implementation can perform very close to hand-written CUDA equivalent and improve performance up to 2.18X (compared to the baseline OpenACC.)

***Index Terms*—OpenACC, Cache memory, CUDA, Software-managed cache, Performance.**

I. Introduction

The OpenACC standard introduces directives, API, and the environment for developing applications for accelerators. Most of OpenACC directives and clauses map to API calls of low-level accelerator programming models, like CUDA[1]. OpenACC can be viewed as a high-level programming layer over low-level accelerator programming models, simplifying accelerators' software interface. Compared to low-level programming models, OpenACC reduces development effort significantly, as measured up to 11.9X in terms of words of code by a previous work [1]. On the other hand, OpenACC applications can run much slower than the CUDA versions. This is because CUDA programmers can harness all accelerator resources and apply advanced optimizations. Examples of these optimizations are exploiting CUDA shared memory as a fast on-chip cache for inter- thread block communication [2] and CUDA texture or constant cache for improving memory bandwidth. OpenACC, however, mainly relies on the compiler to apply low-level optimizations. This is due to the fact that programmers are limited by the notation of OpenACC, which centers around expressing parallelism. Therefore, for OpenACC to be competitive with CUDA in high-performance computing, developing compiler optimizations are crucial.

In this work, we investigate the compiler aspect of implementing the *cache* directive. We study various implementations and optimization opportunities. We start with presenting ineffectiveness of a straightforward implementation. We show the mapping of parallel loop iterations to CUDA threads can be configured to share the cache among several loop iterations. This, in respect, improves cache utilization and accelerator occupancy, yielding a significant speedup. We also present optimizations for cache fetch routine and cache write policies. We apply our optimizations and implement a cache directive, performing close to the hand-written CUDA version. In summary, we make the following contributions:

- To the best of our knowledge, this is the first work investigating the implementation aspect of the *cache* directive. We show that a naïve implementation hardly improves performance (presented in Section II). We provide better understanding regarding implementation challenges and list compile-time optimizations and opportunities to enhance performance (presented in Section IV).
- We introduce three methods for implementing the *cache* directive (presented in Section III). One of the implementations emulates hardware cache. The other two cache a range of values. Methods differ in cache utilization and access overhead. Employing all suggested optimizations on top of our best solution delivers performance comparable to that provided by the hand-written CUDA equivalent.
- We introduce microbenchmarking to understand the performance of shared memory in CUDA-capable GPUs (presented in Section V-A). We show that the shared memory layout (2D or flattened) has minor impact on performance. Also we present how using a small padding in shared memory allocation can vastly resolve bank conflicts. We use our findings in optimizing the *cache* directive implementation.
- We evaluate our suggested implementations under three

[1]While we focus on CUDA in this paper, most of the discussions apply to OpenCL as well.

benchmarks (presented in Section V-B): matrix-matrix multiplication, N-Body simulation, and Jacobi iterative method. For each benchmark we compare performance of the proposed *cache* directive implementations to baseline OpenACC and hand-written CUDA. We also estimate development effort of OpenACC and CUDA versions. We improve the performance of OpenACC up to 2.18X, and almost match that of CUDA (while reducing the development effort by 24%).

II. Background and Motivation

OpenACC API is designed to program various accelerators with possibly different cache/memory hierarchies. Generally, the compiler is responsible for generating an efficient code to take advantage of the hierarchies. Static compiler passes can figure out specific variables or subarrays with an opportunity for caching. However, as static passes are limited, OpenACC API also offers a directive, allowing programmers to hint the compiler. The *cache* directive is provided to facilitate such compiler hints. The directive is not accelerator-specific and is abstracted in a general form. These hints specify the range of data showing strong locality within individual iterations of the outer parallel loop, which might benefit from caching.

The *cache* directive is used within a *parallel* or *kernels* region. The directive usually associates with a *for* loop (where the locality is formed) and can be used over or in the loop. The line below shows the syntax of the directive in C/C++:

#pragma acc cache(var-list)

{

\\ *cache region*

}

var-list passes the list of variables and subarrays. Subarray specifies a particular range from an array with the following syntax:

arr[lower:length]

lower specifies the start index and *length* specifies the number of elements that should also be cached. *lower* is derived from constant and loop invariant symbols. This can also be an offset of the *for* loop induction variable. *length* is constant. *Cache region* is the code range where data should be cached.

Listing 1. cache directive; one-dimensional stencil.

```
1  #pragma acc data copy(a[0:LEN],b[0:LEN])
2  for(n=0; n<K; ++n){
3   #pragma acc parallel loop
4   for(i=1; i<LEN-1; ++i){
5     int lower = i-1, upper = i+1;
6     float sum = 0;
7     #pragma acc cache(a[(i-1):3])
8     for(j=lower; j<=upper; ++j){
9       sum += a[j];
10    }
11    b[i] = sum/(upper-lower+1);
12  }
13  float *tmp=a; a=b; b=tmp;
14 }
```

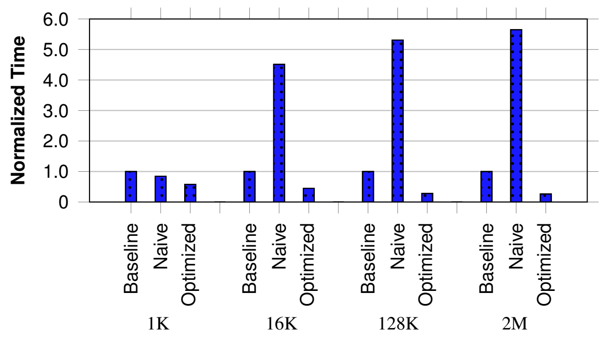

Fig. 1. Comparing naïve and optimized cache implementations under 1D stencil kernel listed in Listing 1 (30-element radius, 1K, 16K, 128K, and 2M elements.)

Listing 1 shows an example of the *cache* directive. The example is based on one-dimensional stencil algorithm. 1D stencil smooths the values of array iteratively, repeating for certain number of iterations, here *K* times. In this example, the array length and 1D stencil radius are *LEN* and one element, respectively. The new value of every element is calculated as the average of three elements; the element and right and left neighbors. The programmer can provide a hint to the compiler to highlight this spatial locality within each iteration of the parallel loop. On line #7, the *cache* directive hints the compiler that each iteration of the loop requires *three* elements of *a[]*, starting from *i-1*. Provided with this hint, the compiler can potentially cache this data in registers, software-managed cache, or read-only cache (depending on the target). Also depending on the accelerator-specific optimization strategies, the compiler can ignore the hint, which is not the focus of this study.

Figure 1 compares the performance of two different cache directive implementations (*naïve* and *optimized*) for the code listed in Listing 1. These two implementations are compared to the *baseline* (which does not use the *cache* directive). The *naïve* implementation isolates cache space to each parallel iteration of the loop. The *optimized* implementation is equipped with optimizations later introduced in this paper and exploits the opportunity for sharing cached elements among parallel iterations. Consequently, *optimized* delivers more efficient cache implementation through better occupancy, cache sharing, and initial fetch parallelization. We explain each of these optimizations in the rest of the paper. This figure emphasizes the importance of optimizing cache implementation.

III. Implementations

In this section, we present three *cache* directive implementations for accelerators employing software-managed cache. We discuss methods for the case where the list of variables consists of subarrays[2]. For implementing the *cache* directive, the compiler requires two pieces of information: i) the range of the data to be cached and ii) the array accesses (within the cache region) that their array index value falls within the

[2]Simplified version of presented methods are applicable for scalar variables.

subarray range[3]. Using the information provided through the directive, the compiler knows the subarray; data that should be cached. To gather the second piece of information, the compiler must examine the index of every array access in the cache region. If the compiler could statically assure that the index falls within the cache range, the array access might simply be replaced by a cache access in the code. Otherwise, the compiler should generate a code to decide to fetch from the cache or global memory on-the-fly. Therefore, depending on the code, the compiler may generate a different control flow. The methods that we discuss differ in how they make this dynamic decision. All implementations guarantee to fetch the data from cache, as long as the value of indexes falls within the specified range.

The first method is an emulation of hardware-managed cache through software-managed cache. To this end, data and tag arrays are maintained in the software-managed cache. Operations of hardware cache is emulated using these two arrays. The second and third methods are range-based caching. The second method stores the lower and length specifiers and checks if the value of the index falls within this range. The third method assumes all indexes fall into the fetched range and uses a simple operation to map array indexes to cache locations. Below we elaborate on these methods.

A. Emulating Hardware Cache (EHC)

Overview. Two arrays are allocated in the software-managed cache; data and tag. Data array stores the elements of the subarray. Tag array stores the indexes of subarray elements that are currently cached. Tag array can be direct-mapped, set-associative, or fully-associative to allow caching the entire or part of the subarray transparently. The decision depends on the subarray size and accelerator capabilities.

Pros and cons. The main advantage of this method is the ability to adapt to the available cache size. If the *cache* directive demands a large space and the accelerator's cache size is small, this method allows storing only portion of the subarray (other methods might ignore the directive in this case). There are two disadvantages with this method though. First, storing the tag array in the software-managed cache lowers the occupancy of the accelerator and limits concurrent threads. Second, at least two cache accesses (tag plus data) are made for every array access, increasing the read/write delay significantly. In terms of operations, each global memory access is replaced by two cache accesses and few other logical/arithmetic and control operations. This significant overhead impairs the performance advantages as the total latency of the cache hit can exceed the global memory latency (depending on the accelerator's design).

B. Range-based Conservative (RBC)

Overview. One array and two pointers are allocated in the software-managed cache. The array stores the subarray. Two pointers keep the range of indexes stored in the cache. One of the pointers points to the start index and the other points to the end index (or the offset from the start). To check if the array index falls within the subarray range or not, the index is checked against the range kept in pointers. Two comparisons evaluate this; index $\geq$ start && index > end. If the condition holds, data is fetched from the cache, otherwise from global memory. Moreover, if the condition holds, the index should be mapped from global memory to cache space. The operation for this mapping is a subtraction (index - start).

Pros and cons. The *cache* directive always points to a stride of data. This method exploits the fact that elements of subarray are a row of consequent elements from the original array and minimizes the overhead for maintaining the track of the cached data (compared to EHC). The method stores two pointers pointing to the start and end of the stride. The method can be extended to multi-dimensional subarrays by storing a pair of pointers per dimension. The only disadvantage of this method is the performance overhead of the control flow statement generated for checking whether the index falls within the range of stride or not. This control statement might be an expensive operation for multi-dimensional subarrays ($2+1$ logical ops. for 1D, $4+3$ logical ops. for 2D, etc.).

Listing 2. Implementation of Emulating Hardware Cache (EHC) in CUDA.
```
__device__ void __cache_fetch(PTRTYPE* g_ptr,
  PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx,
  unsigned* ctag_ptr){
    for(unsigned i=st_idx; i<en_idx; i++){
        unsigned cache_idx=acc_idx&0x0ff; // direct map
        c_ptr[cache_idx]=g_ptr[i];
        ctag_ptr[cache_idx]=i;
    }
}
__device__ PTRTYPE __cache_read(PTRTYPE* g_ptr,
  PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx,
  unsigned acc_idx, unsigned* ctag_ptr){
    unsigned cache_idx=acc_idx&0x0ff; // direct map
    if(ctag_ptr[cache_idx]==acc_idx){
        return c_ptr[cache_idx];
    }else{
        c_ptr[cache_idx]=g_ptr[acc_idx];
        ctag_ptr[cache_idx]=acc_idx;
                return c_ptr[cache_idx];
    }
}
__device__ void __cache_write(PTRTYPE* g_ptr,
  PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx,
  unsigned acc_idx, PTRTYPE value, unsigned* ctag_ptr){
    unsigned cache_idx=acc_idx&0x0ff; // direct map
    if(ctag_ptr[cache_idx]!=acc_idx)
        ctag_ptr[cache_idx]=acc_idx;
    g_ptr[acc_idx] =c_ptr[cache_idx] =value;
}
```

C. Range-based Intelligent (RBI)

Overview. This method improves RBC one step further and assumes array indexes always fall within the subarray range. This avoids the costly control flow statements for evaluating whether the data is in the cache or not. The compiler may use

[3]We assume pointer aliasing is not the case and pointers are declared as restricted type in the accelerator region, using C's *restrict* keyword.

this method if the compiler passes are able to find the range of values of the index.

Pros and cons. This method has significant performance advantage over RBC as it avoids the costly control statements for checking if the data exists in the cache or not. Assuring that the index always falls within the fetched stride was not a trivial compiler pass in the past. The restrictions added in the latest OpenACC version have addressed this by limiting the subarray references. Accordingly, the latest version of OpenACC (2.5 released in November 2015) adds a restriction to *cache* directive requiring all references to the subarray lie within the region being cached[3]. This essentially means RBI can be used with all applications that follow OpenACC 2.5.

Listing 3. Implementation of Range-based Conservative (RBC) in CUDA.

```
__device__ void __cache_fetch(PTRTYPE* g_ptr,
  PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx){
    for(unsigned i=st_idx; i<en_idx; i++)
        c_ptr[i-st_idx]=g_ptr[i];
}
__device__ PTRTYPE __cache_read(PTRTYPE* g_ptr,
  PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx,
  unsigned acc_idx){
    if(acc_idx>=st_idx && acc_idx<en_idx){
        unsigned cache_idx=acc_idx-st_idx;
        return c_ptr[cache_idx];
    }else
        return g_ptr[acc_idx];
}
__device__ void __cache_write(PTRTYPE* g_ptr,
  PTRTYPE* c_ptr, unsigned st_idx, unsigned en_idx,
  unsigned acc_idx, PTRTYPE value){
    if(acc_idx>=st_idx && acc_idx<en_idx){
        unsigned cache_idx=acc_idx-st_idx;
        c_ptr[cache_idx]=value;
    }
    g_ptr[acc_idx]=value;
}
```

D. Example

Listing 2 and 3 show the CUDA implementations of the methods explained above. Three procedures are implemented for each method: i) *__cache_fetch()*, ii) *__cache_read()*, and iii) *__cache_write()*[4]. During generating the accelerator code, *__cache_fetch()* is called early before the cache region starts. This procedure is responsible for fetching the data into the cache. Within the cache region, the compiler replaces every array read with *__cache_read()* call and array write statement with *__cache_write()* call. For these implementations, we assume a write-through cache.

Listing 2 shows the CUDA implementation of EHC where the tag array models a direct-map cache. For this example, we assume a 256-entry cache. In this case, mapping from global memory indexes to cache space is a single logical operation. Listing 3 shows the CUDA implementation of RBC. RBI implementation is the same as Listing 3, except the control statement in *__cache_read()* and *__cache_write()* is removed as it is always not-taken. In this Listing, the mapping is an arithmetic operation; subtracting index from the start pointer. *__cache_fetch()* routine in all implementation has a *for* loop statement. Later in Section IV, we discuss opportunities to accelerate this loop through parallelization.

IV. Implementation Optimization

Various optimizations can be applied to maximize performance in cache directive implementations. Below we review possible opportunities in optimizing cache fetch routine, cache sharing, cache writes, and minimizing index mapping overhead.

A. Cache Fetch Routine

The cache fetch routine is called before cache region starts. This is done once per parallel instance of the loop which the *cache* directive is associated with[5]. If the cache region has long latency, this routine's performance may not be the limiting factor. Otherwise, if the cache region is short, the performance of this routine is critical to the overall performance.

Performing our evaluations under NVIDIA GPUs, we found that minimizing control flow statements comes with significant performance advantage. The fetch routine has a *for* loop statement (as presented earlier in Section III-D) which imposes control flow overhead. Loop unrolling can be employed to reduce this overhead, as the length of the loop is a compile-time constant (equal to the length of the subarray). Also the compiler can reduce this overhead further by sharing a single *for* loop among multiple subarray fetches. Compiler heuristics can decide if the loop can be shared among multiple subarrays. For example, the compiler can read the *cache* directive and group the subarrays having equal length. Subsequently the grouped subarrays can share the same *for* loop, as the number of iterations for fetching the data is the same for all of them.

Another opportunity to optimize the *for* loop is to parallelize the loop. A number of parallel threads, equal to the size of the thread block, can be employed to fetch the data into the software-managed cache. If the compiler is not using parallel threads for another task, parallel fetch can simply achieve this. However, if parallel threads have already been employed to execute parallel tasks, then the compiler should assure that while threads collaborate for fetching the data, they maintain a separated view of the cache, specially in the case of cache writes. We explain this further in Section IV-B.

B. Cache Sharing

The *cache* directive is located within one (or more) parallel loop(s) (also referred to as *outer* parallel loops) and the cache space should be allocated once per parallel instance. The compiler can optimize cache utilization by unifying the allocations of common data and sharing them among parallel iterations. When it comes to *cache* directive implementations in CUDA, sharing data between parallel iterations is efficiently feasible by mapping parallel iterations (in OpenACC) to threads of

[4]As a performance issue, these procedures are declared inline to avoid procedure calls within the accelerator region.

[5]The fetch routine might be called multiple times, if located in a sequential loop.

the same thread block (in CUDA). Therefore, to seize this opportunity, the *cache* directive implementation closely relates to the mapping of outer parallel loops to CUDA threads. Notice that iterations of parallel loops located in the *cache* directive already share the same data. The challenge is to find data sharing among the iterations of outer parallel loops containing the *cache* directive. Listing 4 clarifies outer and inner loops in an example. Below we discuss different cases where the compiler may find sharing among these iterations and seize the opportunity.

Listing 4. Example of inner and outer parallel loops around cache.

```
#pragma acc parallel loop
for(i=0; i<N; i++){ // OUTER LOOP:
 // depending on X and Y, the subarray
 // may or may not be shared among iterations
 #pragma acc cache(subarray[X:Y])
 { // beginning of cache region
  #pragma acc loop
  for(j=0; j<N; j++){ // INNER LOOP:
   // the subarray is shared among all iterations
  }
 } // end of cache region
}
```

OpenACC API accepts hints from the programmer to explicitly specify the mapping of loop iterations to different thread blocks (*worker* clause) or the same thread block (*vector* clause). Since these clauses affect the code that the compiler injects, to provide better understanding, we study two cases: i) implicit mapping (the option is left to the compiler to decide) and ii) explicit mapping (the programmer suggests specific mapping).

1) Case I: implicit mapping.: If the mapping option is left to the compiler, the task is to map outer parallel loops to thread hierarchies with the constraint of maximizing the subarray overlap among threads of the thread block. This is followed by sharing this data among all threads of the thread block through CUDA shared memory. The problem inputs are i) the number of outer parallel loops, ii) the number of iterations per loop, iii) increment steps of outer parallel loops, iv) specifiers in the *cache* directive specifying the range of subarray, and v) number of dimensions in the subarray. The problem output is the mapping of loop iterations to CUDA threads. The mapping should specify the total number of threads, number of threads in thread block, and tasks per thread (which can be calculated from the first two). From the target mapping, the compiler finds the common data among threads of the thread block and injects code to i) calculate *start* and *end* pointers for each thread (if RBC or RBI is used) and ii) fetch the common data collaboratively in parallel using threads of the thread block. We suggest the following compiler passes as solutions to this problem.

One-dimensional subarray. Below are the steps of the pass for one-dimensional subarrays and one outer parallel loop:

I) Repeat steps **II** to **V** for each subarray listed in the *cache* directive. **II)** Read the specifiers for the subarray (e.g. *subarr[lower:length]*); *lower* and *length*. **III)** Construct the AST tree of *lower*[6] and label variables that are iterators of the outer parallel loops as *ITR*. **IV)** If the equivalent expression of the AST tree is in the form of $ITR + const$ (*const* is an expression of non-*ITR* variables) and increment step of *ITR* loop is $+1$ or -1, the compiler may use the following mapping to share the subarray[7]: Map every iteration of *ITR* parallel loop to one thread (1st iteration to 1st thread, 2nd iteration to 2nd thread, etc. one task per thread). Thread block shares $length + blockDim.x - 1$ elements of the subarray among the threads of the thread block through the shared memory (the size of thread block can be adjusted to maximize accelerator's occupancy, e.g. use default 256.). For RBC and RBI, *start* and *end* pointers should be calculated. The *start* pointer of subarray for each thread equals to $ITR - threadIdx.x$ (if the loop step is $+1$) or $ITR + threadIdx.x$ (if the loop step is -1). The *end* pointer of subarray for each thread is equal to $start + (length + blockDim.x - 1)$. The *__cache_fetch()* routine, in Listing 2 and 3, can be parallelized by simply setting the loop initial value to $st\_idx + threadIdx.x$ and the loop step to $+ = blockDim.x$. **V)** If the equivalent expression of the AST tree is not in the form mentioned in **IV**, skip the cache sharing optimization.

The pass above can be simply extended to support a one-dimensional subarray and *two* or more outer parallel loops. The difference is to map each outer parallel loop along a unique dimension of the CUDA grid. Also wherever *blockDim* and *threadIdx* is used above, instead of x dimension, the dimension that corresponds to the *ITR* should be used (x, y, or z).

Two-dimensional subarray. Steps of the pass for two-dimensional subarrays and two outer parallel loops is very similar but scales to two dimensions of the subarray and thread block. Below we list the pass:

I) Repeat steps **II** to **V** for each subarray listed in the *cache* directive. **II)** Read the specifiers for the subarray (e.g. *subarr [lower0:length0] [lower1:length1]*); *lower0*, *length0*, *lower1*, and *length1*. **III)** Construct the AST trees of *lower0* and *lower1* and label variables. Variables that are iterators of the outer parallel loops are labeled as *ITR0*, *ITR1*, etc. **IV)** If the equivalent expressions of the AST trees of *lower0* and *lower1* are in the form of $ITR + const$ (*const* is an expression of non-*ITR* variables), increment step of *ITR* loop is $+1$ or -1, and *ITR* variable in *lower0* is different from the one in *lower1*, the compiler may use the following mapping: Map each parallel loop along a dimension of the grid and map every iteration of the loops to one thread (one task per thread). Threads of every thread block share $(length0 + blockDim.x - 1) \times (length1 + blockDim.y - 1)$ elements for the two-dimensional subarray in the shared memory. The size of thread block can be adjusted to maximize accelerator's occupancy, e.g. use default 16 threads along x and 16 threads along y. For RBC and RBI, four pointers should be calculated (*start0*, *end0*, *start1*, and *end1*) to track both dimensions of the

[6]Based on the *cache* directive restrictions, *length* is constant and *lower* is the real parameters left for the compiler to analyze.

[7]Same mapping can be used if the AST tree is in the form of *const*.

subarray. The *start0* pointer of the subarray for each thread is equal to $ITR + const - threadIdx.x$ (if the loop step is +1) or $ITR + const + threadIdx.x$ (if the loop step is -1). Similarly, the *start1* pointer of subarray for each thread is equal to $ITR + const - threadIdx.y$ (if the loop step is +1) or $ITR + const + threadIdx.y$ (if the loop step is -1). The *end0* and *end1* pointers of subarray for each thread are equal to $start0 + (length0 + blockDim.x - 1)$ and $start1 + (length1 + blockDim.y - 1)$, respectively. The *__cache_fetch()* routine, in Listing 2 to 3, should be modified by adding pointers of the second dimension of the subarray in the arguments. Also one additional loop should be added to iterate and fetch the second dimension of the subarray. Two loops in this routine can be parallelized simply by setting the initial values of the loops to $lower0 + threadIdx.x$ and $lower1 + threadIdx.y$, respectively and the loop steps to $+= blockDim.x$ and $+= blockDim.y$, respectively. **V)** If the equivalent expression of the AST trees is not in the form mentioned in **IV**, skip the cache sharing optimization.

The pass above can simply be extended to support two-dimensional subarrays and *three* or more outer parallel loops. The difference is to map each outer parallel loop along one dimension of the CUDA grid and use the respective dimension of *blockDim* and *threadIdx* instead of *x* and *y* accordingly.

2) Case II: explicit mapping.: In this case, the compiler should generate a specific mapping of parallel loops to CUDA thread hierarchies, forced by *vector* and *worker* clauses. This can limit the range of compiler optimizations in sharing the cache space among threads. Generally, as long as the *vector* and *worker* clauses are not conflicting with the compiler passes in Section IV-B1, the compiler proceeds and exploits the sharing opportunity. The conflict mostly occurs when *worker* clause is used. *worker* clause asks the compiler to map each iteration to a thread block. This can conflict with the compiler passes presented above, if the compiler decides to map this loop to threads of the thread block. In the case of conflict, the compiler can limit the sharing range, e.g. sharing only across one dimension of the grid and ignoring the sharing along the *worker* loop, or even ignoring the sharing optimization, in the worst case.

C. Cache Write Policy

Writing to the subarray in the cache region invokes the write routine. We assume two alternative policies for cache write: write-back and write-through. Write-back buffers cache writes and writes final changes back to DRAM at the end of the cache region. Write-through writes every intermediate write to both cache and global memory. Write-back tends to perform better under dense and regular write patterns whereas write-through performs better under sparse irregular write patterns. We compare performance of these two implementations in Section V-C.

If the compiler implements write-back cache, an additional routine should be invoked at the end of the cache region to write the dirty content of the cache to global memory. For tracking the dirty lines, the compiler can decide to i) keep track of the dirty lines through a mask or ii) assume all the lines are dirty. Although keeping track of dirty lines can reduce the total amount of write operations, the compiler can instead use the brute-force write-back on the GPUs for two reasons. First, tracking dirty lines demands extra space from the software-manage cache to store the dirty mask. This, in turn, lowers the occupancy of GPU. Second, the write-back routine can include extra control flow statements to filter out dirty lines. These control flow statements can harm performance (e.g. limiting ILP and loop unrolling). On the other hand, employing a dirty mask is preferred, if the size of the cache is large. In this case, the dirty mask version is more efficient than the brute-force approach. In this paper we assume brute-force write-back cache.

D. Index Mapping

As we discussed in Section III, mapping global memory indexes to shared memory indexes involves a few operations. To mitigate this overhead, the compiler can allocate a register to store the output of operations for the life time of the cache region, if the value of index is not changing in the cache region. The compiler can also reuse this register for other array accesses, if the array indexes have the same value. This optimization saves register usage and mitigates index mapping overhead.

V. Experimental Results

In this section, we first report the experiments performed to understand shared memory and optimize our implementation on the target GPU. We use IPMACC compiler [4] for compiling OpenACC applications and implementing the *cache* directive. IPMACC framework translates OpenACC to CUDA and uses NVIDIA *nvcc* compiler to generate GPU binaries. Then we study the performance of methods introduced in Section III, under three test cases. Finally, we investigate performance of different cache write policies. We run evaluations under NVIDIA Tesla K20c GPU. The execution time of the kernel is measured by nvprof [5]. Every number is harmonic mean of 30 independent samples. To the best of our knowledge, currently there are no commercial or open source OpenACC compilers that support the cache directive. Therefore, we are unable to compare performance of our implementation to other studies[8].

A. Cache Performance Sensitivity

Software-managed cache in NVIDIA GPUs (also called shared memory) employs multiple banks to deliver high bandwidth. Every generation of NVIDIA GPUs has a certain configuration of shared memory; namely a specific number of banks and the bank line size. A bank conflict occurs once a

[8]We studied several compilers (i.e. PGI and Omni) but found none of them supporting the cache directive. We compiled the kernels with PGI Accelerator compiler 16.1 and found out that the compiler ignores the cache directive and does not generate shared memory CUDA code. We also investigated several open source frameworks, e.g. RoseACC, accULL, and Omni compiler, of which none had an implementation for the cache directive.

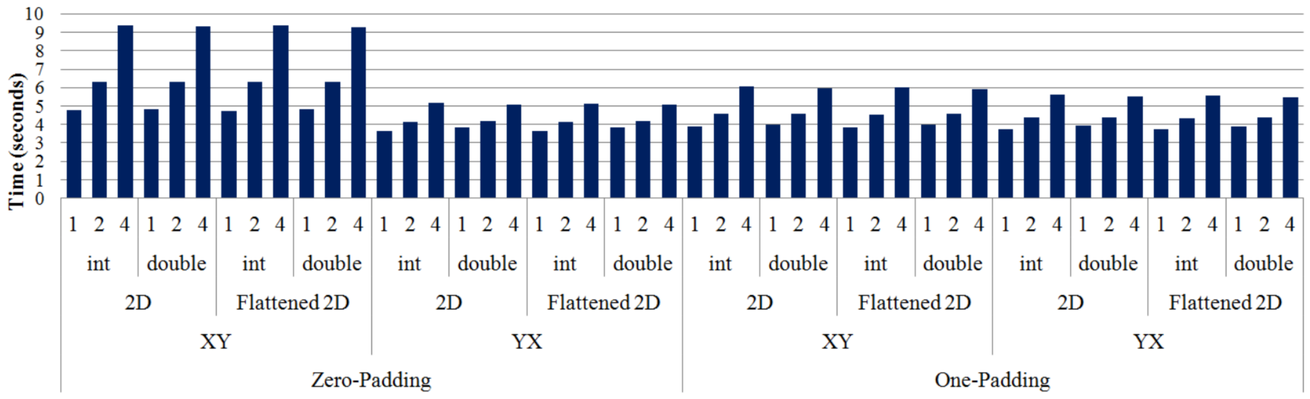

Fig. 2. Comparing execution time of kernel under various shared memory configurations.

warp[9] executes a shared memory instruction and threads of a warp need different rows of the same bank. Bank conflicts cause access serialization if the bank does not have enough read/write ports to deliver data in parallel. We develop a CUDA microbenchmark to evaluate the impact of several parameters on bank conflict. Knowing these impacts delivers deeper insight on optimizing the *cache* directive implementations and enhancing their performance. This test should run separately for every backend supported by the compiler to allow hardware-specific optimizations. Below we first review the microbenchmark structure, followed by presenting results obtained on the GPU of this study. Finally we summarize the findings that help optimizing the cache directive implementation.

1) Microbenchmark setup: We assume one two-dimensional shared memory array per thread block. We also assume two-dimensional thread blocks. We develop a simple kernel in which every thread reads four locations of shared memory and writes one location. These reads/writes are in a loop iterated several times. The code is shown in Listing 5. We report the execution time of this kernel and evaluate the impact of the following parameters in the kernel body:

- **Datatype size (TYPE):** The datatype size of shared memory array is the number of bytes allocated for each element of array. Variations in datatype size impact bank conflict since it determines the layout of array in the shared memory (e.g. one element per bank, two elements per bank, etc.).
- **2D array allocation:** We investigate two alternatives in allocating 2D shared memory: 2D array notation or 1D array notation (flattened notation). 2D array notation is simpler in indexing and code readability. We are also interested to understand whether flattened notation has a different layout in the shared memory from 2D array.
- **Padding (PAD):** When the size of shared memory array is multiple of memory banks, adding a small padding to the array can mitigate the bank conflict. The padding increases the row pitch, spreading the columns of a row across different banks.
- **Access pattern:** Since bank conflict only occurs among the threads of the same warp, it is important to mitigate bank conflict algorithmically. We evaluate the impact of these algorithmic optimizations by mapping threads of the thread block to different dimensions of the shared memory array. Operating in *XY mapping*, threads along the *x* dimension of the thread block are mapped to the first dimension of the shared memory array and threads along the *y* dimension are mapped to the second dimension. *YX mapping* reverses this as threads along *x* and *y* dimensions are mapped to the second and first dimensions of the array, respectively.
- **Iterations (ITER):** Number of iterations of the loop in the kernel body. This number indicates the ratio of shared memory accesses to global memory accesses.

2) Results: Figure 2 reports the execution time of the kernel in Listing 5 under various configurations. Bars report the execution time for three different ITERs (1, 2, and 4), two TYPEs (4-byte integer and 8-byte floating-point), two array allocation schemes (2D and flattened 2D), two shared memory access patterns (XY and YX), and two padding sizes (zero and one).

As shown in the figure, TYPE has modest impact on the execution time. Also the allocation scheme has minor impact on performance. The latter suggests that the layout of 2D array in the shared memory banks is similar to that of the flattened 2D array.

Access pattern, however, impacts performance significantly. In this benchmark, YX mapping delivers a better performance compared to XY. This is explained by how threads are grouped into warps. Warps are occupied first by the threads along the *x* dimension and then by the threads along *y*. Therefore threads along *x* should access consequent words in order to reduce shared memory bank conflict. This is precisely what YX mapping does.

As shown in the figure, adding a padding to the array can have an impact similar to that of access pattern tunings, lowering the execution time roughly the same amount. Adding

[9]Group of threads executing instructions in lock-step over the SIMD.

a padding to the array can lower the execution time by 57% and 56% under *double* and *int*, respectively. It should be noted that under the cases where the array is padded there is still room for improvement as evidenced by the results. Under one padding, modifying the code algorithmically for reducing bank conflict, as comparing XY to YX shows, can further lower the execution time by 8% (for both *int* and *double*).

Increasing the number of iterations (ITER) increases the importance of the shared memory performance in the overall performance. For larger iterations, the impact of access pattern and padding is more significant. For example, under one iteration, the gap between zero-padding and one-padding is 23%. This gap grows to 37% and 55% under 2 and 4 iterations, respectively.

3) Summary of findings: We make the following conclusions from the findings presented in this section and use them to optimize our implementations. First, the layout of 2D arrays allocated in the shared memory is found to be the same as flattened 2D arrays. Since no performance advantage is found in using flattened 2D arrays, we use multi-dimensional arrays for caching multi-dimensional subarrays to simplify array indexing code generation. Second, our implementation adjusts mapping of parallel loops to x and y dimensions of the thread blocks with the goal of having threads along x accessing consequent bytes. We use a heuristic to map the most inner parallel loop to the x dimension of the grid. This is due to the fact that, intuitively, the inner loop has stronger locality and traverses arrays column-wise. Third, adding a small padding can pay off if other compiler optimizations do not allow mapping inner parallel loops over x dimension.

B. Test Cases

Here we investigate the cache directive under three different benchmarks; matrix-matrix multiplication (GEMM), N-Body simulation, and Jacobi iterative method. For each benchmark, we compare the performance of four implementations[10]: i) OpenACC without *cache* directive, ii) OpenACC plus *cache* directive implemented using RBC, iii) OpenACC plus *cache* directive implemented using RBI, and iv) hand-written CUDA version. All cache-based implementations are optimized with the parallel cache fetch and cache sharing optimizations discussed in Section IV.

We wrote all versions of GEMM and Jacobi. For N-Body Simulation, we used the CUDA version available in GPU Computing SDK [6] and modified the serial version available there to obtain OpenACC versions. We did our best to hand-optimize using the techniques that we are aware of. Table I compares the development effort of GEMM, N-Body, and Jacobi under OpenACC, OpenACC plus cache, and CUDA implementations. Development effort is measured in terms of the number of statements, including declaration, control, loop, return, and assignment statements. Below we compare the performance of these implementations.

[10]We found EHC implementation very slow and hence we avoid further discussion on about it.

Listing 5. CUDA microbenchmark for understanding shared memory.

```
// compiled for  different  TYPE, ITER, PAD, XY
__global__ void kernel(TYPE *GLB, int size){
    __shared__ int SHD[16+PAD] [16+PAD];
    // mapping config to  shared memory
    #ifdef XY
    int row=threadIdx.x,  rows=blockDim.x;
    int col=threadIdx.y,  cols=blockDim.y;
    #else
    int row=threadIdx.y,  rows=blockDim.y;
    int col=threadIdx.x,  cols=blockDim.x;
    #endif
    // fetch
    int index=(threadIdx.x+blockIdx.x*blockDim.x)*size+
               (threadIdx.y+blockIdx.y*blockDim.y);
    SHD[row][col]=GLB[index];
    // computation core
    int S = (row==(rows-1))?row:row+1;
    int N = (row==0)        ?0  :row-1;
    int W = (col==(cols-1))?col:col+1;
    int E = (col==0)        ?0  :col-1;
    int k=0; TYPE sum=0;
    for(k=0; k<ITER; k++){
        sum=(SHD[row][col]+ SHD[S][col]+ SHD[N][col]+
             SHD[row][E]+ SHD[row][W])*0.8;
        __syncthreads(); SHD[row][col]=sum; __syncthreads();
    }
    // write-back
    GLB[index]=SHD[row][col];
}
```

TABLE I
DEVELOPMENT EFFORT OF THE BENCHMARKS UNDER OPENACC, OPENACC PLUS CACHE, AND CUDA IMPLEMENTATIONS.

	OpenACC	OpenACC+cache	CUDA
GEMM	84	94	116
N-Body	81	84	108
Jacobi	145	152	189

1) GEMM: Cache-based implementations (i and ii) iteratively fetch 16×16 tiles of two input matrices into the software-managed cache using the *cache* directive and keep the intermediate results (sum of products) in registers. The CUDA version also implements the same algorithm using *shared memory* notation. Figure 3 compares the performance of these implementations under various square matrix sizes, compared to the baseline OpenACC.

A similar trend can be observed under different input sizes. RBI outperforms OpenACC by 2.18X. The gap between RBI and CUDA is around 8%. RBC, RBI, and CUDA reduce the global memory traffic significantly, compared to OpenACC. By fetching the tiles of input matrices into software-managed cache, these implementations maximize memory access coalescing. Also these implementations exploit the locality among neighbor threads to minimize redundant memory fetches. Using *nvprof* [5], we found that RBI reduces the number of global memory loads by 12X (under 1024x1024 matrices), compared to OpenACC (the very same improvement is observed under RBC and CUDA too).

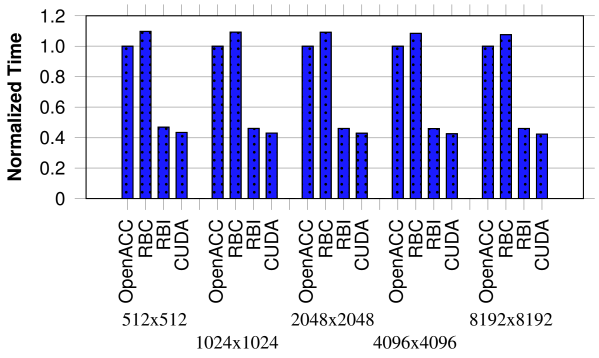

Fig. 3. Comparing performance of four GEMM implementations under different matrix sizes. For each bar group, bars from left to right represent OpenACC without cache directive, OpenACC with cache directive implemented using RBC, OpenACC with cache directive implemented using RBI, and CUDA.

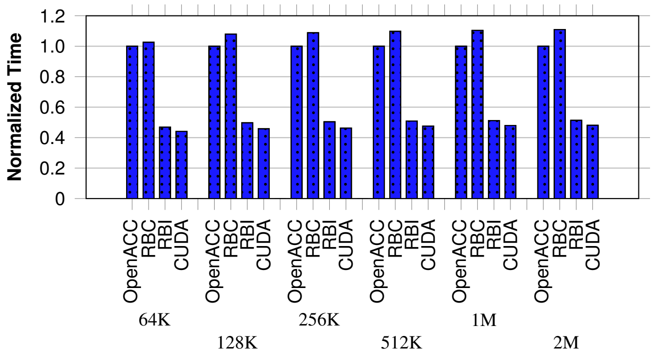

Fig. 4. Comparing performance of four N-Body simulation implementations under different number of bodies.

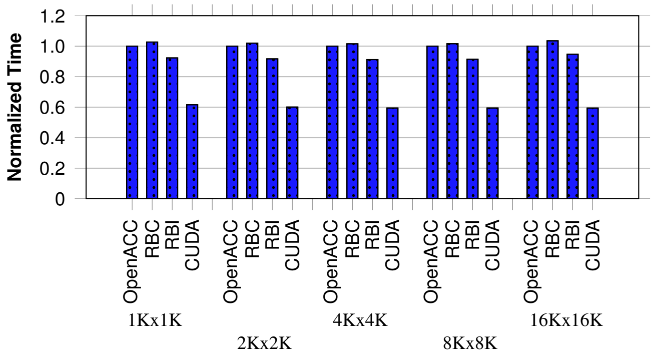

Fig. 5. Comparing performance of four Jacobi iterative method implementations under different matrix sizes.

Using RBC, the compiler generates a code to check the memory addresses dynamically and to find out if the address falls within the subarray range or not. If the address falls within the subarray range, the data is fetched from the cache. Otherwise, the data is fetched from the global memory. Under RBI, however, the compiler static passes assure that dynamic memory accesses always fall in the subarray range (if violated, the program can generate incorrect output). Therefore, dynamic checking for the address range is avoided. This explains why RBI always performs faster than RBC. As shown in Figure 3, RBC is 2.34-2.37X slower than RBI. This gap is caused by RBC's extra logical and control flow instructions per memory access, negating the gain achieved from using the software-managed cache. For the 2D subarray of this benchmark, these extra instructions are one branch, four comparisons, and three ANDs. We discuss this issue further in Section VI.

2) N-Body simulation: Figure 4 compares four implementations of N-Body simulation under different problem sizes. To improve performance using software-managed cache, interaction between masses are computed tile-by-tile. Bodies are grouped into tiles and fetched into software-managed cache one tile at a time. This lowers redundant global memory instructions and DRAM accesses. RBI outperforms baseline OpenACC by 95%-113%. While RBI performs very close to CUDA, there is still a gap between them (nearly 9%). This gap is mainly the result of efficient implementation of the fetch routine in the CUDA version. RBC is unable to improve performance of the baseline OpenACC. This is explained by the overhead for accessing software-managed cache; i.e., assuring the address falls within the range of data existing in the shared memory.

3) Jacobi iterative method: Figure 5 compares four implementations of Jacobi iterative method under different problem sizes. Each thread in Jacobi reads nine neighbor elements (3-by-3 tile) and updates the value of the center element. Considering a two-dimensional matrix, calculations used by neighbor elements share significant amount of input data (four to six elements.) Fetching this data into software-managed cache and sharing data among threads is one way to optimize baseline OpenACC. We employ this in RBC, RBI, and CUDA implementations. Although our analysis shows RBC lowers global memory accesses, RBC harms overall performance when compared to the baseline. This is explained by the overhead (control flow and logical operations) of assuring addresses fall within the range of the data fetched into the shared memory. RBI removes this overhead and improves performance of baseline OpenACC by 6-10%. Despited this we observe a huge gap between RBI and CUDA. CUDA launches thread blocks equal in size to the size of the data being used by the thread block. RBI, however, launches thread blocks equal in size to the size of the computations being performed by the thread block. This results in the CUDA version using slightly larger thread block size than RBI. Here threads at the boarder of thread block are only used for fetching the data. This reduces irregular control flow in the fetch routine. We found that this can be effectively implemented in OpenACC to reduce the gap between RBI and CUDA. However, we do not investigate it further due to the high development effort required (close to CUDA equivalent), which is not desirable for high-level OpenACC.

C. *Cache Write*

We developed two synthetic workloads to investigate performance of write-back and write-through policies. The first workload's write pattern is *dense* and *regular*. The workload is of 1D Stencil type where each parallel work computes an element in the output array, iteratively. In OpenACC terms, all parallel iterations are active (forming the dense pattern) and consequent iterations write consequent words (forming the regular pattern). Every parallel work serially iterates for a certain number of iterations (which is a run parameter) and computes the value of the element iteratively. The second workload is the same as the first, except that only a fraction of threads are active (less than 2%) and only a fraction of serial iterations perform write (less than 2%). This forms the *sparse* pattern.

Parameters of these workloads are parallel iterations (total number of work) and number of serial iterations within the work. The number of serial iterations models the frequency of cache writes. Sweeping this number from 4 to 4096, we measure the performance of write-back and write-through under various cache access frequencies.

Figure 6 compares write-back and write-through under the two synthetic workloads described above (*dense regular* versus *sparse*). Two problem sizes are reported for each workload, 128K and 4K parallel work. We observe a similar trend under both workloads. When parallel work are massive (e.g. 128K work), write-back is faster than write-through (Figure 6b and 6d). This is due to the fact that large amount of threads can perfectly hide the latency of write-back's final write routine. When parallel work are small and write frequency is low (e.g. left side of Figure 6a and 6c), write-through outperforms write-back. For example in Figure 6a, write-through is faster when write frequency is lower than 16. Going beyond 16, write-back starts to catch up with write-through. This can be explained by the higher rate of global memory writes that write-through makes. For large write frequencies (e.g. >64), write-through performs numerous redundant writes to global memory. Write-back, in contrast, buffers intermediate written values (in shared memory) and writes them all to global memory once at the end of cache region. This reduces the total global memory writes compared to write-through and saves performance. As presented, the performance gap between write-back and write-through increases from 7% to 34%, as write frequency increases.

VI. Limitations

In EHC, tag and data arrays should be kept consistent. This limits the parallelism of software-managed cache operations, specially write operations. For instance, if two threads miss different data and want to fetch both into the same location, synchronization is necessary. The synchronization overhead can be significant as the only way to handle such scenarios is to create a critical section or use atomic operations. Because of this limitation, for performance goals, cache sharing optimizations should be avoided on top of EHC. We exclude EHC from evaluations as we did not find it competitive.

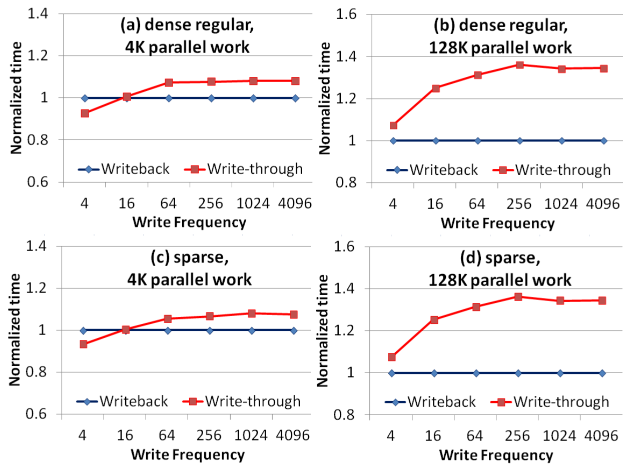

Fig. 6. Comparing execution time of kernel under various shared memory configurations.

In RBC, *__cache_read* routine is the performance limiting factor, listed in Listing 3. Investigating the CUDA assembly of the kernel (in *sass* format), we found that the compiler eliminates branches and instead uses predicates. This, on the positive side, eliminates extra operations for managing the post-dominator stack [7]. On the negative side, all instructions, in both taken and not taken paths of the branch, are at least fetched, decoded, and issued (some are executed as well). The *nvcc* compiler uses a heuristic to employ predicates or generate control flow statements (we describe this in Section 5.4.2 of [8]). For *__cache_read* routine of RBC, the heuristic finds predicate advantageous. However, the overhead of the predicate version is still huge and the routine is translated to 16 machine instructions. This explains why RBC is slow. We believe further optimizations on RBC should be performed at the machine level.

NVIDIA GPUs have alternative on-chip caches that can be used by OpenACC compiler as the target of the cache directive (e.g. constant memory and texture cache) or can be used effortlessly as an alternative to the cache directive (L1 cache and read-only cache). Constant and texture memory are limited to read-only data. If the subarray is written in the cache region, constant and texture memory can not store the latest value nor deliver the latest to subsequent requests. In addition, the precision of the application could be affected if texture memory is used. We evaluated the performance impact of L1 and read-only caches separately. We enforced read-only cache using *const* and *__restrict__* keywords and forced the GPU to cache global accesses through *nvcc* compile flags (*-Xptxas -dlcm=ca*) and found out that performance improvements are less than 2%. This suggests that the advantages of using software-managed cache is not limited to reading/writing data from/to faster cache, but also accessing the data in fewer transactions and in a coalescing-friendly way.

VII. Related Work

Reyes et al. [9] developed accULL to execute OpenACC applications on accelerators. Two major components of accULL are i) source to source compiler and ii) runtime library. The runtime library routines are implemented in both CUDA and OpenCL. Tian et al. [10] presented an OpenACC implementation built in OpenUH [11]. Using OpenUH, they evaluated the performance of several alternatives in mapping loop iterations to GPU parallel threads. Lee and Vetter [12] introduced a framework for compiling, debugging, and profiling OpenACC applications. They also introduced a new directive, *openarc*, mapping OpenACC arrays to CUDA memory spaces. CUDA memory spaces include shared memory and texture memory. Hoshino et al. [13] investigated the impact of memory layout on the performance of NVIDIA Kepler architecture, Intel XeonPhi, and Intel Xeon processors. They limit their study to directive-based programming languages. Their study shows that having structure-of-arrays is much more efficient than array-of-structures for Kepler and XeonPhi, while it has minor impact on the performance of Xeon. They explain this by the cache size of these processors (Kepler, XeonPhi, and Xeon have 110 Bytes, 128 KBytes, and 1048 KBytes of cache per hardware thread, respectively). They also introduced a new directive for changing the data layout of multi-dimensional arrays. Herdman et al. [1] compared the performance of *parallel* and *kernels* constructs under various implementations of OpenACC. They found that most vendors focus on one of these constructs. Comparing the quickest construct of the vendors, their performance variations found to be below 13%.

VIII. Conclusions

In this paper, we studied and addressed the challenges facing the OpenACC *cache* directive in NVIDIA GPUs. We used CUDA shared memory as the software-managed cache space for implementing the directive. We presented three different methods and several performance optimizations for implementing the *cache* directive, among which sharing the cache space among multiple threads and parallelizing cache fetch and write routines are the most critical. Our results also show that a) sharing the cache among several parallel threads is essential to have a robust performance and b) write-back cache outperforms write-through policy for the majority of memory patterns. We also presented a CUDA shared memory test to understand structural hazards and performance bottlenecks of the shared memory. Evaluating under matrix-matrix multiplication, N-Body simulation, and Jacobi method iteration test cases, we presented an implementation that can perform close to hand-written CUDA.

References

[1] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. Jarvis, “Accelerating hydrocodes with openacc, opecl and cuda,” in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, Nov 2012, pp. 465–471.

[2] A. Lashgar and A. Baniasadi, “Employing software-managed caches in openacc: Opportunities and benefits,” *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 1, no. 1, pp. 2:1–2:34, Feb. 2016. [Online]. Available: http://doi.acm.org/10.1145/2798724

[3] “The OpenACC Application Programming Interface,” *Available: http://www.openacc.org/content/openacc-2-5-final-specification*.

[4] A. Lashgar, A. Majidi, and A. Baniasadi, “IPMACC: open source openacc to cuda/opencl translator,” *CoRR*, vol. abs/1412.1127, 2014. [Online]. Available: http://arxiv.org/abs/1412.1127

[5] N. Corp., “Profiler’s user guide: nvprof,” *Available: http://docs.nvidia.com/cuda/profiler-users-guide/#nvprof-overview*.

[6] NVIDIA Corp., “CUDA Downloads,” *Available: https://developer.nvidia.com/cuda-downloads Last visited: 24 Mar. 2016.*

[7] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, “Dynamic warp formation and scheduling for efficient gpu control flow,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 407–420. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.12

[8] NVIDIA Corp., “CUDA C Programming Guide,” *Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ Last visited: 24 Mar. 2016.*

[9] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, “accull: An openacc implementation with cuda and opencl support,” in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 871–882. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_86

[10] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman, “Compiling a high-level directive-based programming model for gpgpus,” in *Proceedings of the 26th International Workshop on Languages and Compilers for High Performance Computing*, ser. LCPC 2013, 2013.

[11] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, “Openuh: An optimizing, portable openmp compiler: Research articles,” *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 18, pp. 2317–2332, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1002/cpe.v19:18

[12] S. Lee and J. S. Vetter, “OpenARC: Extensible OpenACC Compiler Framework for Directive-based Accelerator Programming Study,” in *Proceedings of the First Workshop on Accelerator Programming Using Directives*, ser. WACCPD ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/WACCPD.2014.7

[13] T. Hoshino, N. Maruyama, and S. Matsuoka, “An OpenACC Extension for Data Layout Transformation,” in *Proceedings of the First Workshop on Accelerator Programming Using Directives*, ser. WACCPD ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 12–18. [Online]. Available: http://dx.doi.org/10.1109/WACCPD.2014.12